A Formal Framework for the Analysis of Human-Machine Interactions

Sébastien Combéfis

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

November 20, 2013

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)

Louvain School of Engineering (EPL)

Université catholique de Louvain (UCL)

Louvain-la-Neuve

Belgium

Abstract

There are more and more automated systems and people are led to interact with them everyday. They are also increasingly complex and exhibit more and more "smart" behaviour. One direct consequence is that it becomes harder for the human operators to drive those systems safely for both the system and the user. Due to that increasing complexity, interactions between users and automated systems are more likely to be error-prone. In particular, inadequately designed interactions may result in the user being surprised while interacting with the system. Several accidents are due to such surprising situations, as it can be testified by real accidents, including the *Three Mile Island* nuclear meltdown, the lethal radiation doses administered by the *Therac 25* medical device or the shutdown of the aircraft of the KAL007 flight.

Human-Computer Interaction (HCI) has been studied for many years by researchers from various fields including psychology, human factors and ergonomics. This thesis follows a recent research direction that considers the use of formal methods to analyse the behavioural aspects of HMI. The focus is put on the actions or events exchanged between an operator and the system being used during an interaction. The work of this thesis builds on its initial inspiration from the recent work of Degani and Heymann that addressed the problem of automatically generating an adequate user interfaces for a given system model. In their work, an adequate user interface refers to one ensuring that potential mode confusion is avoided for the operator.

The main contribution of this thesis is an analysis framework supported by formal methods that can be used to assess whether a system model is prone to potential automation surprises when being used by

a human operator. The thesis develops a formalisation of automation surprises. It proposes and precisely characterises the full-control property that captures the fact that interactions between a system and its operator are free of potential automation surprises. It also defines a property, the full-control determinism, that guarantees the existence of a full-control conceptual model for a given system model.

The thesis also defines precisely the minimal full-control conceptual model generation problem. The problem consists in finding a minimal conceptual model of the system model that allows full-control of it, which is only possible for fc-deterministic system models. Such full-control conceptual models can be used to generate artefacts to help the user to better understand them, such as user and training manuals. Three algorithms are proposed to solve the generation problem. The first one is based on Three-Valued Deterministic Finite Automata (3DFA) that are used to characterise the full-control property in terms of traces. The second one is based on a reduction approach inspired by the Paige-Tarjan algorithm that solves coarsest partition problems. Finally, the third one is based on an active learning approach based on the L^* algorithm.

The three proposed algorithms have been analysed for correctness and time complexity considerations. Moreover, the proposed framework, and therefore the proposed algorithms, have also been tested on various examples among which a large case study of an autopilot. That latter case study comes from ADEPT, a toolset to support designers in the early phases of the design of automation interfaces. That case study also shows how the proposed methodology could be integrated with ADEPT.

Acknowledgments

All my research activities, and in particular this thesis, have been carried out in the Computer Science and Engineering Department (INGI) of the Université catholique de Louvain (UCL). More precisely, I got the opportunity to do my research in the *Louvain Verification Lab* (LVL) whose head is my advisor Prof. Charles Pecheur. I am most grateful to Prof. Charles Pecheur for his help and guidance provided for those six years. All that support was necessary to bring me to scientific research and lead this work whose subject was a discovery for him and even more for me. He also provided me with the freedom I needed, installed a confidence relation, and presented me many profitable persons who allowed me to be become part of an outstanding international community.

In particular, I got the opportunity to meet and work with Dr Dimitra Giannakopoulou during two internships that I have done at the NASA Ames research center. I also thank very much Dr Dimitra Giannakopoulou that gave me a big punch, opening new fruitful directions of investigation that contributed to a large part of my thesis.

I also wish to thank the other members of my examination board, Professors Jean Vanderdonckt, Thierry Massart, Philippe Palanque and Peter Van Roy, who accepted to review my thesis and whose comments played a crucial role in the final quality of this thesis.

Moreover, I am also thankful to all the people I got the chance to work with, or talk to, about my research. I would also like to thank all my past and present colleagues from the UCL with whom I worked on various projects related to my research or to the university life. In particular, I thank my two office mates José Vander Meulen and Simon Busard for all the great times we spent together; Matthew Bolton, Michael Feary,

iii

Foreword

The work of this thesis lies mainly between two fields, namely human-computer interaction and formal methods. More specifically, this thesis is about using techniques based on formal methods to analyse human-machine interactions. In such situations, it is not always easy or possible to use a vocabulary common to both fields, each community having their specific words to designate similar objects or concepts. An effort has been made to use terms that refer unambiguously to the same object or concept. However, to help the reader and avoid any confusion, a glossary that gathers the key terms used in this work is provided on page 253.

In this work, the two main concerned entities are the system and the user who is operating the system. To use formal methods to analyse the interaction, models of those two entities are used. Whereas the model of the first entity is always referred to as the system model, the second entity is called mental model or conceptual model in this thesis. Both terms refer to a model of the knowledge of the user about the system that is operated. However, mental model is used to refer to the model that actually lies in the mind of the user, and can evolve over time. The term *mental model* is mainly used up to Chapter 4 since the proposed analyses could be applied to a mental model at a given time. From Chapter 5 the term *conceptual model* is used to distinguish the mental models that are generated by the algorithms proposed in this thesis, from the actual mental model of the users. Conceptual models can be seen as "perfect mental model" that the users should have in their mind.

Finally, another specificity of the human-computer interaction field is that it includes the study of the human, as a user of interactive systems. In this thesis, the pronouns "he", "his" and "him" are used in their epicene sense, to refer to the human user.

Contents

Chapter 1

Introduction

The number of automated interactive systems has been growing rapidly those years. Such systems include consumer electronics, cars, vending machines, medical devices and aircrafts. Moreover, those systems are also increasingly complex and exhibit more and more "intelligent/smart" behaviours. A direct consequence is that it is hard to ensure that the human operators will be able to operate those systems safely and without confusion. The increasing complexity of automated systems has important consequences on the interactions, making them more error-prone. Several accidents are due to wrong interactions between the operator and the system being used, as testified by real accidents [Deg04, Per99].

Critical accidents include the *Three Mile Island* accident which was a partial meltdown of one nuclear reactor that took place in Pennsylvania in the United States on March 28, 1979. This accident included human errors where the operators were confused by wrong indications provided by the user interface on the control panels of the reactor. A valve that was stuck open appeared as closed to the operators that were not trained to handle the particular ambiguous nature of that indicator. With the information shown on the interface, the operator thought the system was in another configuration than the actual one and he drove the system according to his wrong understanding. That confusion lead them to performing inadequate operations with the system.

Another system which is involved in several incidents between 1985 and 1987 and which lead to human death is the *Therac 25*, a radiation therapy medical device. This device can administer two kinds of beams to patients, one of which requires a spreader to be in place. Because of some internal events, invisible to the operator, some patients were administered lethal doses of radiations. This accident is an example of a well-known class of problems called *automation surprises* [SWB97, Pal95].

In such situations, the system reacts in a way that the operator did not expect, causing surprise and confusion. More precisely, such a situation is a *mode confusion* [LPS+97], a particular case of automation surprise where the actual operating mode of the system is not the one expected by the operator, which can induce bad interactions with the system.

A lot of accidents also occurred in the aviation domain. One very impressive example is the *Korean Air Lines Flight 007* whose aircraft was shot down by a Soviet interceptor Sukhoi Su-15 on September 1, 1983. The airplane deviated more than 200 miles from its route inside the prohibited Soviet airspace, but due to a confusion the pilots did not notice the deviation and continued to fly just as if they were on the correct route.

These accident examples, and the many other existing ones, are not only due to a bad system design, but also to the human operator. Neither the system nor the human is at fault on their own, but the problem is located in a faulty interaction. As automated systems continue to become more and more complex, human-machine interaction errors like automation surprises are more likely to appear. That increasing complexity can in turn lead to more and more incidents if no particular attention is paid to those potential errors during the design phases of the systems.

Human-Machine Interaction (HMI) has been extensively studied for several years by researchers from various fields, mainly from psychology, human factors and ergonomics. Initially, the research focused on incidents or accidents that had occurred, in order to understand what went wrong. From those analyses, researchers moved to the development of models of the interaction and of the human cognition. Since the mid-1980s, researchers have been investigating the use of formal methods to analyse behavioural aspects of HMI. Formal methods bring rigorous, systematic and automatic techniques that can be used to help designers for the analysis and design of complex systems involving human interactions.

The first results were obtained by using and applying formal methods techniques to analyse existing accidents. The analysed systems were modelled with a mathematical formalism from which properties were verified by using automatic formal analysis tools. The work then moved to more generic results using theories like graph theory, model checking or theorem proving. The researchers now develop techniques that can

be applied on system models in order to identify whether potential automation surprises can arise when the system is to be used by an operator. Such automatic identification of potential interaction errors can be integrated into the design process of a system

1.1 Research Goals

The objective of this thesis is to demonstrate how techniques based on formal methods can be used to analyse aspects of human-machine interactions (HMI). More precisely, one of the aspect of interest is the detection of potential automation surprise situations. The goal of the work is to define a general formal framework to support the analysis of HMI. The framework has to be based on formal models and on properties to capture precisely the elements that play a role for the desired analyses. The framework must be supported by algorithms. It must also be applied on case studies in order to demonstrate the validity and applicability of the proposed approach. A real-size example must also be analysed in order to assess how the algorithms scale.

The techniques that are developed in this thesis are meant to be used by system designers during the design of a system or by system analysts during any later phase requiring to analyse an existing system. Many accidents that happened were due to a wrong interaction between the user and the system being operated. Those wrong interactions can be the result of an automation surprise. On possible way to analyse formally potential automation surprises is to focus on the behavioural aspect of the interaction. This is precisely the focus of this work which focuses on system whose behaviour can be described such as the VTS example presented hereafter.

This work does not analyse the interaction means or devices, nor the ergonomics and user interface design aspects, such as the modality of the interface or the arrangement of the components of the UI for a software. This work is only concerned by the "event communication" that takes place between the user and the system being used. Correspondingly, it is the software component of the system that is the focus of analysis in the frame of this work.

The following example illustrates the motivation of this work. Degani et al. use a small model of a semi-automatic transmission system of a large vehicle to illustrate the concepts they developed. Figure 1.1 shows a graphical representation of the system model of the *vehicle transmission system* (VTS) example. The system is composed of three operating levels: LOW, MEDIUM and HIGH. The system is composed of eight internal states represented by the rectangular boxes. There are two kinds of actions that are possible with that system, represented as lines linking together two states.

- The user can drive the transmission lever to increase or decrease the operating level of the gearbox (push-up and pull-down). Those actions are under the control of the user who decides when they are performed.
- The gearbox can automatically shift gears as the speed of the engine changes (up and down). Those actions are not controlled by the driver but he can detect their occurrence as he hears the engine speed going up or down.
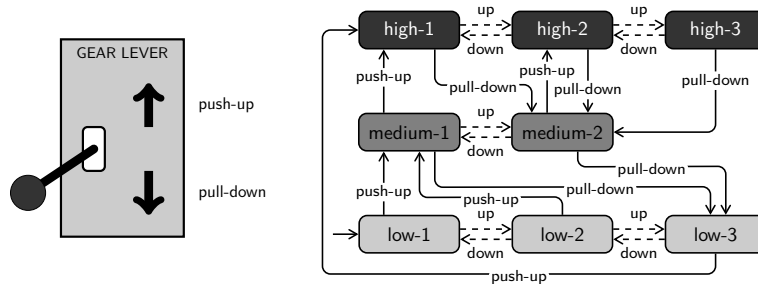
(a) The lever of the VTS (physical device).

(b) A graphical representation of the behaviour of the VTS (model).

Figure 1.1. The Vehicle Transmission System (VTS) (Example taken from [HD07]).

The peculiarity of the VTS example is that pushing up the lever when the system is in the LOW level can lead to two different situations. The gear either moves to the MEDIUM level (when the system is in the low-1 or low-2 state) or to the HIGH level (when the system is in the low-3

state). A direct consequence of that difference is that if the user wants
to be able to predict in which level the gearbox will transition in reaction
of a push-up on the lever, the user must be able to track in which state of
the LOW level the system is.

Such subtleties that may be part of the system model must be taken
into consideration whenever designing a system and analysing it. For
the VTS example, attention has to be paid to the LOW level. Either
the automatic transitions have to be hearable for sure by the driver or
user interface elements such as visual indicators or audible alert must be
provided to indicate the actual state in the LOW level.

The designer of the system may have thought that the system could be
viewed by the user as a three-mode system, one for each level. Applying
the methodology proposed in this thesis can identify that the three-
mode system view is not adequate and may cause potential automation
surprises. Moreover, the developed algorithms can identify that the
operator has to be aware of at least five states about the system in order
to be able to control it while avoiding potential surprises. The states
of the HIGH and MEDIUM levels can be merged together while the three
states of the LOW level are kept separated.

1.2 Overview of the Contributions

The contributions of this thesis are multiple and revolving around the
full-control property, which is the central contribution of this work.

To capture the features that are necessary to model the human-
machine interactions, a generalised version of the automata-based formal-
ism proposed by Degani et al. based on an enriched version of labelled
transition systems (LTS) have been defined. The proposed formalism
includes the distinction between commands performed by the operator
on the system, observations triggered autonomously by the system and
observed by the operator and internal actions not perceivable by the
operator. A formalism mixing event-based and state-based observations
has also been developed to be closer to how system models are designed
by system designers and analysts.

A criterion to characterise one aspect of adequate interactions between
an operator and a system has been formalised and put in the form of

a property called *full-control*. The full-control property captures the fact that for an interaction to be adequate, that is, to avoid potential automation surprises, the operator must always know what commands he can perform on the system and he must expect at least all the observations that may arise. In addition to the characterisation of the full-control property, it has been compared and positioned regarding other existing relations such as trace equivalence and input-output conformance, for example. That comparison highlighted a trace characterisation of the full-control property. Moreover, existence of full-control mental models for a given system is guaranteed by the fc-determinism criterion that is also defined and characterised in this work.

To support the formal framework, three algorithms that can be used to automatically generate a minimal full-control mental model for a given system have been proposed and analysed. The first proposed algorithm is based on the trace characterisation of the full-control property and encodes the full-control mental model candidates into a *Three-Valued Deterministic Finite Automaton* (3DFA) that is then reduced to get one minimal full-control mental model. The second proposed algorithm is based on a *reduction-based* approach similar to the one used to compute the bisimulation reduction. It has been developed based on the procedure proposed by Degani et al. [HD07]. Finally, the third proposed algorithm is based on an active *learning-based* approach where the minimal full-control mental model is built incrementally state by state.

Based on the full-control property and the proposed algorithms, a proposed methodology to analyse a given system model during the design stages is developed and illustrated on existing small but realistic case studies. Those analyses highlight some limitations of the full-control property and some possible variants of the full-control property are proposed, analysed and illustrated with small examples. Those analyses also revealed the necessity to integrate user tasks into the formal analysis. That latter point is sketched in this thesis and precise and further analyses are left for future work.

The proposed methodology and algorithms have also been applied on a larger realistic case study which is a model of an autopilot of a Boeing 777 aircraft. The autopilot model that has been used is initially modelled in ADEPT, a tool dedicated to system designers to help them to prototype user interfaces. To be able to analyse that model, this

thesis proposes a formal semantics for ADEPT models and proposes a systematic way to translate ADEPT models to the models used in this work.

In summary, the contributions of this thesis are as follows:

- Definitions and characterisation of HMI-LTS, HVS and HVM models, along with translation algorithms from HVS and HVM to HMI-LTS;

- Definition and characterisation of the full-control and fc-determinism properties, along with algorithms to check them;

- Three minimal mental model generation algorithms, based on: 3DFAs, a reduction-based approach and a learning-based approach;

- An analysis framework with a proposed method, illustrated by concrete examples and a variant of the full-control property along with intuitive ideas about the integration of user task models;

- A semantic for ADEPT models, with a translation algorithm to HVS and the analysis of an autopilot case study.

1.3 Publications

The work presented in this thesis has already given rise to five publications:

- The definitions of HMI-LTS and of the full-control property have been initially published in *the first ACM SIGCHI Symposium on Engineering Interactive Computing Systems* in July 2009 [CP09]. The paper also describes the reduction-based algorithm for the generation of minimal full-control abstractions. The version presented in the paper is restricted to system models satisfying specific requirements, but the algorithm has been generalised in this thesis to handle any system model.

- The application of the proposed analysis techniques to the human-machine interaction field, and in particular how they can be used in the design process of systems, has been published in a special session of the *IEEE International Conference on Systems, Man, and Cybernetics* in October 2011 [CGPF11a].

- The learning-based algorithm for generating a minimal full-control conceptual model has originally been published in the *International Workshop on Machine Learning Technologies in Software Engineering* in November 2011 [CGPF11b]. This paper also provides the trace characterisation of the full-control property.

- The prototype tool jpf-hmi is described in a paper published in *the Java Pathfinder Workshop* in November 2011 [CGPM11].

- Finally, exploratory work on how user task models can be integrated into the analyses has been presented at the *Third International Workshop on Formal Methods for Interactive Systems* in November 2009 [Com09].

1.4 Organisation of the Thesis

This thesis is organised in eight chapters. The remainder of this thesis is organised as follows:

- Chapter 2 presents fields of the human-machine interaction and formal methods. After reviewing background information about those two fields, it draws up the state of the art of applying formal methods to the analysis of human-machine interactions. Finally, the precise context in which the work presented in this thesis lies is summarised.

- Chapter 3 presents HMI-LTSs which are the enriched labelled transition systems that are used to represent system and mental models. It also presents HVS and HVM, generalised versions of HMI-LTSs that mixes event-based and state-based approaches for observations. Finally, other kinds of formalisms used to model reactive systems are briefly presented and compared to the formalism proposed in this work.

- Chapter 4 presents the full-control property and characterises it. An algorithm to check whether the full-control property is satisfied between a mental model and a given system model is proposed. The fc-determinism property that guarantees the existence of a full-control mental model is presented and characterised. Finally the

last section reviews how properties developed by other researchers can be used to relate mental and system models, and how they compare to the full-control property.

- Chapter 5 formulates formally the problem of generating a minimal full-control mental model for a given system. Based on a trace characterisation of the full-control property, it presents three approaches: the 3DFA-based, the reduction-based and the learning-based approaches. Concrete algorithms are described and the three approaches are compared.

- Chapter 6 follows by presenting how the proposed full-control property and mental model generation algorithms can be used concretely to analyse human-machine interactions that can potentially lead to interaction failures. The proposed methodology is illustrated with small realistic examples. Limitation of the full-control property is also presented, along with a discussion about how user task models can be integrated into the analysis.

- Chapter 7 applies the proposed technique to a real-size concrete example which is a partial model of an autopilot of the Boeing 777 aircraft written in the ADEPT toolset. ADEPT models are first described and a formal semantics for them is proposed. Then, the autopilot model is described and analysed with the methodology proposed in this thesis.

The last chapter concludes the work presented in this thesis and draws up some possible future work.

Chapter 2

Context and State of the Art

The goal of this first chapter is to lay down background information, about both the human-machine interaction and the formal methods fields, that is relevant for the work presented in this thesis. Section 2.1 defines *human-computer interaction* and provides definitions for general concepts and elements used throughout this thesis. Section 2.2 reviews main kinds of analyses that can be performed for human-machine interactions, and which are close to the analyses performed in this thesis. Then, Section 2.3 defines *formal methods* and presents briefly two main verification techniques: model checking and theorem proving. Section 2.4 follows directly with a state of the art that reviews how formal methods has been used for the analysis of human-machine interaction. Finally, Section 2.5 draws up the main underlying hypotheses of this work.

2.1 Human-Computer Interaction

Human-computer interaction (HCI) is a field that studies the interactions between people and computers, and their designs [CMN83, Car97]. This field is sometimes referred to as *human-machine interaction* [Boy11], *human-automation interaction* or *man-machine interaction* in the literature. HCI is an interdisciplinary field which involves and connects together computer science, psychology, system engineering, ergonomics, human factors, cognitive science and interface design [HLP97]. The origins of HCI date back to 1960 in the seminal paper by Joseph Licklider [Lic60] which states the need to have easier interactions between humans and computers.

The main goal of HCI is to work on the design of *interactive systems*, along with their interfaces, to make them usable by their users. The cognitive capabilities of the users, such as what they can observe about

11

the system or their memory capacity, are one important aspect considered by HCI people. They have to be well understood to be able to allow building systems that are usable by their operators. Whereas it is a crucial point of interest for the field in general, it is not really the focus in this thesis.

There is no one unique definition of human-computer interaction. ACM SIGCHI defines it as *"a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them"* in a technical report describing a curricula for HCI [HBC⁺92]. This latter definition only considers computing systems, but more generally, the work presented in this thesis is also applicable to any machine whose behaviour can be described as an automaton. However, the proposed techniques can only be applied on finite automata even though many interactive systems do have an infinite set of states. They will therefore not be considered for analysis, except under suitable abstraction or simplification. Also, looking back to the definition just mentioned, this work is more focused on the evaluation part, and to a lesser extent on the design part of interactive systems, but not much on the implementation part.

To highlight the fact that the work of this thesis is not only concerned with interactive computing systems, the term *human-machine interaction* (HMI) is used in the sequel.

2.1.1 Human-Machine Systems

Human-machine interactions are happening between machines and their operators. But, there are also many other elements involved and of interest for the approach developed in this thesis. Among the involved components, some of them correspond to concrete elements while others are abstractions used for the formal analysis. Those elements, presented in the next section, are inspired from the ones defined by Degani et al. [DH02]. The difference is that they are put in a broader and more general context, such as proposed in [Cac04]. A *human-machine system* (HMS) can be defined as in [DoD84]:

> "A composite, at any level of complexity, of personnel, procedures, materials, tools, equipment, facilities, and software. The elements of

> this composite entity are used together in the intended operational or support environment to perform a given task or achieve a specific production, support, or mission requirement."

There can be more than one system in an HMS, in general. The system can also be decomposed in several subsystems running in parallel. Many architectures are possible in such a setting: either the operator interacts with each system independently or there can be machine-machine interactions. In that latter case, those interactions can be visible or invisible to the operator. One example is a bread production line composed of the succession of several machines. The user has to control and operate the different machines while the breads are progressing through the production line. The different machines may be communicating together and exchanging information in order to adjust their own settings, given a communication protocol.

It is also possible to have more than one user operating the system. In such a situation, the different operators may just act independently on the system, they can cooperate to reach a global goal, or they can be competing against each other. For the two latter cases, there is the need to consider the possible human-human interactions as well. One example of such a situation is an aircraft that is flown by a pilot and his copilot.

The work presented in this thesis only considers HMS with a single machine used by a single operator. The extension of the developed techniques to more general HMSs is left for future work.

Reactive Systems

By definition, a system in an HMS is a *reactive system* [HP85]. A reactive system is viewed at a high level of abstraction as a black box which is provided with inputs and produces outputs. Such systems usually never terminate and are always interacting with their environment. Harel and Pnueli precisely stressed the difference between transformational systems which take inputs and produce outputs in a functional way, and reactive systems which *"are repeatedly prompted by the outside world and their role is to continuously respond to external inputs"* [HP85]. Reactive systems are offering a set of input actions to their environment and, in reaction to these inputs, produce outputs.

It is exactly this dynamic aspect and distinction between inputs and outputs that are of interest for the work of this thesis, that is, the behavioural control aspects of HMI (versus ergonomic and timing aspects, for example). That is the reason why considered systems are all reactive systems. The operator is interacting with the system, by providing inputs and observing outputs, and without any knowledge about the purely internal behaviour of the system, considered as a blackbox by the user. The system is in an ongoing relationship with the environment and the operator. To be more precise, the operator may know precisely how the system works, but during the interaction, it is not possible for him to know what is happening inside the system, except by observing the produced outputs and remembering the provided inputs.

2.1.2 General Overview of Involved Elements

Figure 2.1 shows elements that are involved in a human-machine system [Nor88] and that are of interest in the frame of the work presented in this thesis. The user interacts with the system through an interface. That interaction takes place in a given environment. The user gets his knowledge about the system through user manual or training, for example, and has to perform some tasks. The behaviour of the system is captured by a system model and the knowledge of the user about the system is represented by a mental model. The conceptual model abstracts the system model and is used to generate training material.

System, User and Interface

Three real concrete elements are involved in human-machine interactions. The *system* is the machine that is used, such as a vending machine, an ATM, an autopilot, a car or a software running on a computer. The system may either be a physical device or a computer program that can be used to control a physical device.

The *user* is the operator who is using the system. Different kinds of interactions are possible between the user and the system. For example, the user may be monitoring the system, executing a procedure or trying to achieve a specific task with the system. The user can also just be exploring and discovering the functionalities of a system that he has never used before.
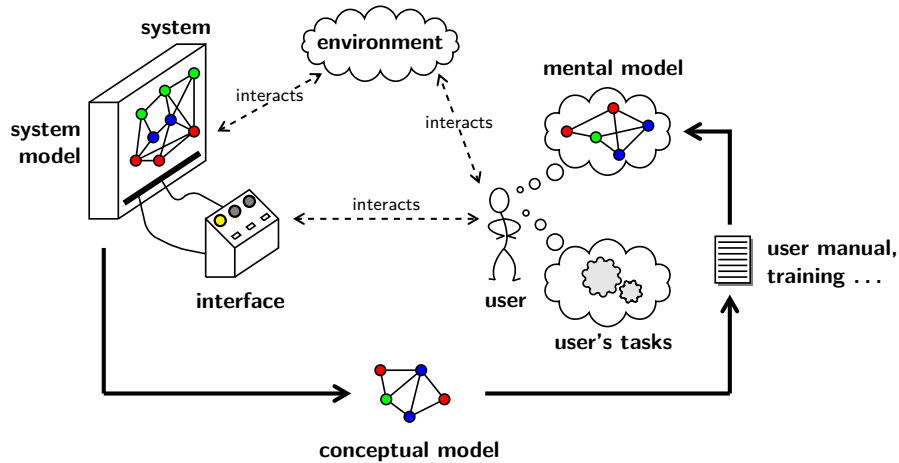
Figure 2.1. General overview of elements involved in human-machine interactions.

The third element is the *interface* which is located between the user and the system and serves as a communication channel. The interface allows the user to send commands to the system. It is also through the interface that the user will be able to observe information provided by the system. Any interaction between the user and the system is done through the interface. The interface can either be a physical or a virtual interface. The interaction with physical interfaces is done through physical devices such as buttons, knobs or levers to execute commands and lights, LCD displays, gauges or alarms to get information about the system. For virtual interfaces, the interaction is done through a command line or through the widgets of the graphical user interface (GUI). The interface is tightly coupled to the system and is most of the time considered as being part of it.

Environment

The interaction between the user and the system does take place inside some environment. The *environment* is everything that has an influence on the interaction, but which is external to the system and the user. It is also referred to as the *socio-technical working conditions* or the *context*. Usually neither the system nor the user have any control on

the environment, but it must be taken into account by both the system and the user to ensure a good interaction. The environment is made of three elements: the actual environment, other operators and the social context [Cac04].

For example, elements belonging to the environment may be physical values such as temperature, pressure or wind speed for physical systems. For a computer running a software system, it could be the operating system or the internet service provider (ISP).

System and Mental Models

In order to be able to analyse human-machine interaction using formal methods, both the system and the user are modelled with a well-defined modelling formalism. The interaction analysis is based on a system model and a mental model which respectively models the system and the user. The *system model*, also referred to as the *implementation model* [CRC07a] when referring to software systems, represents the *behaviour* of the system. It contains all the executions that the system can go through. The *mental model* contains the knowledge that the user has about the system, that is, all the executions that he knows or thinks that are possible on the system [SN93]. The mental model can also be characterised as the representation of how the system works that emerges from its use. Norman [Nor02] defines it as follows:

> "Mental models are models people have of themselves, others, the environment, and the things with which they interact. People form mental models through experience, training, and instruction."

By definition, the mental model is therefore a highly probabilistic model. Moreover, it evolves over time as the user is learning new functionalities or is forgetting others as they are not used frequently enough, for example. Social interactions with other users of the same or similar systems may also alter the mental model.

There are plenty of different formalisms that can be used to model system and mental models. The choice of a formalism depends, among other things, on the kind of analysis to be performed. Moreover, the system model may not represent the exact behaviour of the system, it is often an *abstraction* of it. This is for example the case when a continuous

system is modelled using a discrete formalism. The abstraction level that is chosen may also have an influence on the kinds of analyses that are possible.

User's Tasks

A user is operating a system because he has a goal to achieve, that is, some tasks to be executed on the system. The user's tasks can be of different types. One possible mission for an operator is to monitor a system, and to ensure that it will never reach some forbidden hazardous states. For example, the operator's goal may be to watch the temperature of the cooling system of a nuclear power plant. Another kind of user's tasks is the execution of a procedure on a system, that is, performing a sequence of commands that will depend on the observations that are made during the interaction. For example, a user may want to go to an ATM to withdraw money with his credit card. The interaction will be different if there is no more money left in the ATM, or if the credit card has been blocked by the bank, for example. The user will follow a procedure which starts with the insertion of the card followed by the composition of his PIN code. Of course, user's tasks can be more complex and are usually presented as a hierarchy with a main task decomposed into subtasks. For example, "driving a car" could be the main task, with "monitoring the speed" as one of the subtasks.

User's tasks may also be modelled formally in order to be able to apply formal methods techniques to perform analysis related to user's tasks. Possible analyses include the feasibility of a given set of tasks on a system, or the evaluation of the cognitive complexity for an operator to perform a task on a system. Different elements have be considered in order to have a task model.

Training Material

In the frame of this work, the systems, and consequently the system models, are supposed to be immutable. System models are thus frozen and do not evolve over time. Contrarily, mental models may evolve over time, as previously mentioned. The user has initially no knowledge about a specific machine. The user will learn and grow an initial mental model using information coming from many sources, referred to as *training*

material. Those sources of information include common knowledge
coming from the use of similar machines and knowledge learned through
the reading of training manuals or obtained during practical training
sessions.

Training material are not considered directly in this thesis. They may
play a role in two main situations: the approach used in this work can be
used for the generation of user manuals and existing user manuals can be
checked with the developed technique. Small examples illustrating those
two roles are provided in Chapter 6 and a more thorough development is
left for future work.

Conceptual Model

Norman also defined the notion of *conceptual model* that corresponds
to the mental representation of the design. The conceptual model is
developed by the designer and communicated to the user through the
system design and its interface. Norman [Nor86] distinguishes three
conceptual models: the *design model* is the representation of the system
in the mind of the designer, the *user's model* is the model developed
by the user and finally the *system image* is the perceivable part of the
device. The designer cannot communicate directly with the final user
and transmits his design model through the system image, which can
thus be seen as a materialisation of the design model.

In the remainder of this thesis, the generic term conceptual model is
used to refer to an abstraction that is directly generated from the system
model. In contrast, the term mental model refers to the model that is
actually in the user's mind. The conceptual model can be viewed as a
kind of "perfect/ideal" mental model, with enforced properties.

The work presented in this thesis focuses on the analysis of the
interactions that take place between a user and an operated system. The
first focus is on the relation that should be defined between a system
and a mental model so as to detect potential wrong interactions. The
second focus is on the generation of a conceptual model from the system
model, so that the generated conceptual model is a good abstraction of
the system model, that if used by an operator, avoids potential wrong
interactions. The system and conceptual models are thus at the core
of the analysis approach proposed in this thesis. The other elements,

and their integration, are also discussed in this thesis, but to a lesser extent. Moreover, the system with its interface and the environment are considered as a whole, and the interaction is taking place between the user and that "large system".

2.2 Analysing Human-Machine Interactions

The HCI field tackles many different problems related to the analysis of human-machine interactions. This section reviews briefly the main kinds of analysis and positions this work relative to them. Initially, HCI was called "software psychology" and had as a goal to consider human related aspects to better understand software design, programming and interactive systems [Shn80]. Assessing the *usability* of systems and software is a key analysis to get a better understanding of human-machine interaction.

For the particular case of software development, two categories of properties are of interest for people from HCI [GC96]. External properties are related to the perspective of the user while internal properties are related to the perspective of the software developer. A first kind of questions dealt with by HCI is related to goals and tasks. The task completeness property addresses several questions including the possibility for a user to achieve his goals and to do his tasks on a system and the ability of the system to provide a feedback to the user about the successful realisation of a task. Such questions are not covered in this thesis.

Another kind of analysis that is central in HCI is about *interaction robustness*. A robust interactive system is one that supports the user during the execution of a task so that he does not perform any irreversible mistakes, and allows him to know where in the task progress he is. That is precisely the goal of this thesis, defining a property that minimises the risk of failure during the interaction, namely avoiding automation surprises. As summarised in [GC96], seven properties are related to the interaction robustness. *Observability*, *insistence*, *honesty* and *predictability* are user-dependent properties whereas *access control*, *pace tolerance* and *deviation tolerance* are less user-dependent. The first four ones have to be validated by user testing and the three last ones can be validated by analysing the system.

A system is *observable* if it is possible for its user to observe all the information necessary for him to perform his tasks. A system is *predictable* if it helps the user, with its past and present provided information, to predict the outcome of future interactions. The full-control property, defined and developed in this thesis, addresses both properties. It checks whether the user knows enough about the observable behaviour of a system in order to avoid automation surprises when interacting with it.

HCI covers many other kinds of analysis including the development of design processes for interactive systems such as what is done in the software engineering field. This thesis is not related to design processes although future works discusses how the developed techniques could be integrated in the design loop of systems. Other subjects of interest of HCI include the development of cognitive models for the users, the study of affective aspects of the interaction, the analysis of interface and interaction means (physical or virtual), multimodal interactions, support for design, prototyping and construction of interactive systems, evaluation of interactive devices (manual through surveys or automated) and usability testing [DFAB03, SRP07, CRC07b].

2.3 Formal Methods

Formal methods is a field of computer science in which mathematical techniques are used for the specification, development and verification of software and hardware systems [CW96]. Formal methods have been applied to different fields ranging from software and hardware systems design, such as verification of the design of chips or network protocols.

When talking about formal methods, in particular for automated verification of properties, two main approaches are distinguished: *model checking* and *theorem proving*. Those two methods are used to perform verification of formal specifications on systems. Even if only the exploration capabilities of model checking is used in this work, the next two sections present model checking and theorem proving and discuss the difference between those two approaches, so as to present the state of the art in the domain of formal methods applied on HMI.

2.3.1 Model Checking

Model checking [CGP99] is the problem of exhaustively checking whether
a model of a system meets a given specification, both the model and the
specification being formulated in some precise mathematical formalism.
The model checker exhaustively explores all the states of the model
and checks whether the specification is satisfied for the whole system.
Should the specification be unsatisfied, the model checker provides a
counterexample which is an execution trace within the model which
violates the specification. Figure 2.2 illustrates the working of model
checking.
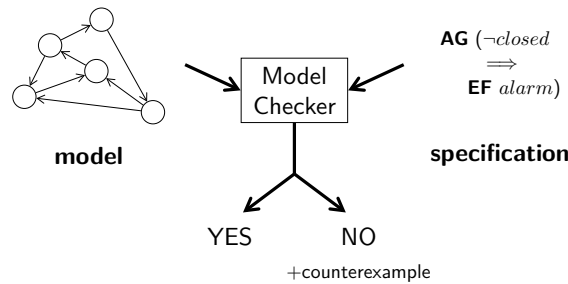
Figure 2.2. Model checking algorithms exhaustively search through all the states of
a model and check whether a given specification is satisfied in all those states. If
the answer is no, the model checker provides a counterexample which is typically an
execution path in the model leading to a state not satisfying the specification.

A *specification* is a set of desired properties each of which representing
an element of the specification. Satisfying the specification means satis-
fying all of its properties. If M denotes a model and S a specification,
the notation $M \models S$ means that the model satisfies the specification.

Different kinds of specifications can be checked with a model checker.
Firstly, a model checker can verify properties expressed in a given logic,
for example *linear temporal logic* (LTL) defined by Pnueli [Pnu77] or
computation tree logic (CTL) defined in parallel by Clarke and Emer-
son [CE81] and in a slightly different form by Queille and Sifakis [QS82].
The specification used in Figure 2.2 consists of a single CTL property,
stating that whenever the door is not closed, an alarm can eventually be
raised.

Besides being able to check whether a given property holds on a model, model checkers can also check whether two given systems are equivalent, according to a chosen equivalence relation. One example of such an equivalence relation is the *trace equivalence*. Two models are trace equivalent if they can perform exactly the same set of observable executions. Model checkers with this possibility include CADP [GLMS13] and FDR2 [For10].

Both those possibilities offered by model checkers have been used in the techniques presented in this thesis. In particular in the algorithm presented in Section 5.5, model checking is intensively used to check whether some forbidden states cannot be reached and to test whether two models are equivalent according to full-control, a property developed in this work.

2.3.2 Theorem Proving

Theorem proving [HV91, Wie06] works differently from model checking by applying inference rules to a specification in order to automatically derive new properties of interest. Given a model of a system and a property to verify, a *theorem prover* outputs a proof stating whether the property is verified or not for the system. The theorem prover must receive as input an appropriate formulation of the problem as axioms, hypotheses and a conjecture. The formulation is provided using a formal logic. During the proof, some theorem provers may require human assistance in order to provide some guidance to the prover. Those systems are therefore referred to as *proof assistants* instead of theorem provers. Theorem provers can output a yes/no answer or they can also output counterexamples or detailed proofs, for more elaborated ones. Many theorem provers have been developed; well-known and successfully systems include Otter [McC03], Coq [The13] and HOL [GM93, For12].

2.3.3 Limits to Adoption

When formal verification techniques started to be developed, people were rather sceptical about what they could bring to computer science and software engineering.

In 1979, De Millo et al. argued that software verification was doomed as summarised in the abstract of their paper [MLP79]:

> "It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design."

De Millo et al. were mainly targeting theorem proving at that time. However, as time passed and as verification techniques became more advanced, people changed their mind. Although many scholars agree that formal methods are a good way to bring safety and reliability in system design, the attitude of practitioners is not as enthusiastic [Hol97]. Except people from the hardware design community, practitioners from other fields are more reticent, not about the general idea of using formal methods, but about the lack of adequate tools, the necessity to understand too much of mathematics or high computational costs. People willing to use verification techniques are often required to be specialised in the field since formal verification is not always as easy as pressing on a button and getting a result. Verification indeed suffers from high computational complexity. It can only be applied with abstraction and scaling up to real-size is challenging. Those arguments have to be taken into account when developing new analyses approaches based on formal methods: new methodologies proposed must be supported by usable tools targeted to their final users who are not necessarily people with high technical backgrounds.

Moreover, as pointed out by Henzinger [Hen96], formal methods do not have to be taken as a holy grail. Formal methods cannot prove that a given system is error-free and that any interaction with it will never cause trouble:

> "The only sensible goal of formal methods is to detect the presence of errors and to do so early in the design process. Indeed, *'falsification'* would be a more appropriate name for the endeavor called *'verification.'*"

Formal methods should be used as a complementary tool inside the design process loop, where they can bring useful feedback about the design of systems to the system designers. Those considerations have been taken into account in the work of this thesis. The modelling approach has been chosen in order to be as close as possible as the way designers think about systems. Even if the work presented in this thesis is focused on the formal method developments and does not go until the production of a usable tool, the case study presented in Chapter 7 shows that the proposed techniques could be integrated with ADEPT, an existing toolset specialised for human-machine interaction analysis. Also, the integration of the proposed techniques in the system design loop has also been thought about, and are presented in Chapter 6.

2.4 Analysing HMI with Formal Methods

The work presented in this thesis aims at analysing human-machine interactions and more precisely verifying that interactions between operators and the machines are safe. Formal methods have proven useful for predicting system failures in computer hardware or software, but have not been used extensively to detect potential human errors yet.

Since the mid-1980s, researchers started to investigate the use of formal methods techniques for HMI. Initial work focused on small case studies but the field has been growing now for many years towards more general and generic techniques. According to Dix [Dix13], the first uses of formal methods in the HCI field date back to the work by Reisner who used BNF to specify user interfaces [Rei81] and the work by Dix et al. who considered the use of abstract models for interactive systems [DR85]. Today, many other researchers have investigated the use of formal methods to analyse HMI, such as described in [HT90, Pal97].

Formal methods can be used for several purposes in the analysis of human-machine interactions. The different approaches can be classified according to the three following categories:

- Systems can be *modelled* formally so that rigorous and systematic analysis can be performed on them. Given a formal model, systems can be characterised and compared. For example, measures for the level of usability or complexity can be computed.

- Properties can be expressed formally and *verified* against systems. Such verifications can be mainly done with either model checkers or theorem provers. For example, a system can be proved to have no deadlocks or it can be checked that some actions are always undoable by the operator.

- Finally, by using formal methods to model HMI, it is also possible to use algorithms that are able to *generate* artefacts, such as user manuals, other models, alternative system models and user procedures, satisfying some precise and well-defined properties. For example, a user manual covering all the behaviour of a system can be automatically generated.

This section presents a state of the art of approaches that have been developed to apply formal method techniques to analyse some aspects of human-machine interaction. For each piece of work, a representative example is concisely presented. Those case studies are a good illustration of real concrete interactive systems that are subjects of formal analysis of human-machine interactions. Some reviews about applying formal method techniques to the analysis of human-machine interactions have been previously realised by several authors [CH97, BBS13, Dix13].

2.4.1 Analysing Mode Confusion with Model Checking

Rushby was among the first researchers to use formal methods in order to provide a more systematic way to analyse human-machine interaction [Rus00, Rus02]. His work first focused on the specific case of *mode confusion* issues, that is, situations where an operator of a system is confused because he thinks the system is in one operating mode while it is actually in another one. Such confusion could possibly lead to accidents. Rushby started with the analysis of a well-known case study, and then extended his work to open perspectives to the more general case of automation surprises analysis.

Systematic Analysis of a Well-know Case Study

In [RCP99], Rushby et al. started from the description of an existing and well understood incident that occurred due to a bad operation between a pilot and an airplane autopilot. From that description, the

authors identified the different operating modes that played a role in
the incident. They also proposed two *state-machine models*: one for
the actual system and one mental model for the pilot. They clearly
identified a discrepancy that could occur between the two models, in one
execution scenario which was actually the one that occurred during the
analysed incident. The formalisation of the situation allowed the authors
to perform a systematic analysis, to reason about the situation and to
identify potential bad interactions.

The case study that was analysed is a *kill-the-capture* bust, a well-
known kind of automation surprise described by Palmer [Pal95]. That
analysis was done in a NASA study in which several crews flew realistic
missions in DC-9 and MD-88 airplanes simulators. In summary, the
analysed incident was due to an altitude deviation of the airplane.

The autopilot of an airplane can be instructed to maintain a given
altitude (hold mode, HLD) or to climb to reach a given altitude, by setting
its *pitch mode.* There are two ways of climbing: at a given rate of climb
(vertical speed mode, VSP) or at whatever rate consistent with a given air
speed (indicated airspeed mode, IAS). In addition to the pitch mode, the
pilot can set a *capture mode* to define a "goal to reach". The pilot can
ask the autopilot to drive the plane to a given altitude (altitude mode,
ALT). An additional capture mode that is not activated by the pilot,
but autonomously by the autopilot, levels off the airplane to the desired
altitude with a smooth levelling off (altitude capture mode, ALT CAP).
There are two more things to know to understand the case study: firstly,
if no capture mode is set and pitch mode is set to VSP or IAS, the airplane
climbs indefinitely, and secondly, if the ALT mode has been set, then
the ALT CAP mode is entered automatically when the airplane arrives
near the target altitude. In the latter case, the pitch mode automatically
transitions to HLD mode when the target altitude is reached. Figure 2.3
shows the controls of the autopilot that are of interest for this case.

The mode confusion the pilot has been confronted to, initially de-
scribed by Palmer in [Pal95], first formally analysed by Leveson et
al. [LP97] and rephrased from [RCP99], took place as follows:

> "The airplane was at 2,100 feet when the pilot received the clearance
> to climb and maintain 5,000 feet. The pilot changed the target
> altitude to 5,000 feet, which automatically armed the ALT capture
> mode. The pilot also set the pitch mode to VSP with a vertical
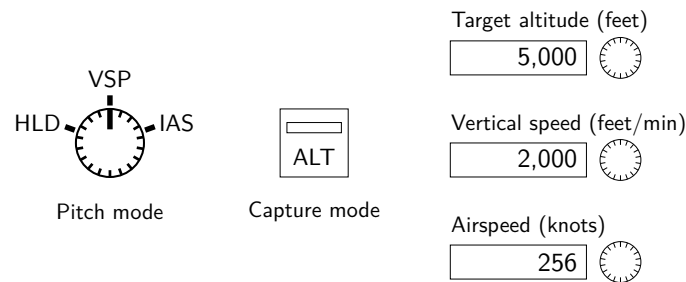
Figure 2.3. Control of pitch and capture modes. The pitch mode is chosen with a knob and the altitude capture can be set or not with a button (on the left). Target altitude, vertical speed rate and airspeed can be chosen with knobs (on the right).

> speed of 2,000 feet/min. When the altitude of the airplane reached 4,000 feet, the pilot changed the pitch mode to IAS, while the airspeed was previously set to 256 knots. The ALT capture mode was still armed. When the airplane approached the target altitude, the autopilot automatically switched to ALT CAP capture mode. A few seconds later, the captain changed the vertical speed to 4,000 feet/min. The autopilot automatically changed the pitch mode to VSP and the airplane continued to climb, not levelling off at the specified target altitude..."

Figure 2.4(a) shows the state-machine that represents the logic of the behaviour of the autopilot for the considered pitch and capture modes, as proposed by Rushby. This state-machine summarises the description presented here above. What caused the automation surprise is that the pilot does not have the same representation of the autopilot in mind. A good mental model may suppress behaviour details and this is necessary since a human operator cannot memorise every detail accurately in his mind. In that specific situation, the pilot was not aware of a difference in the behaviour of the autopilot between ALT and ALT CAP: VSP/IAS resets ALT modes (bold transition on Figure 2.4(a)). For the pilot, whenever the capture mode is active, it remains active even when changing the pitch mode to VSP. Rushby proposed the mental model on Figure 2.4(b). This is precisely what caused the incident: after the pilot changed the vertical speed to 4,000 feet/min, in the last manoeuvre, the actual system transitioned to the initial state where capture is not armed, but in the mind of the pilot, capture was still armed.
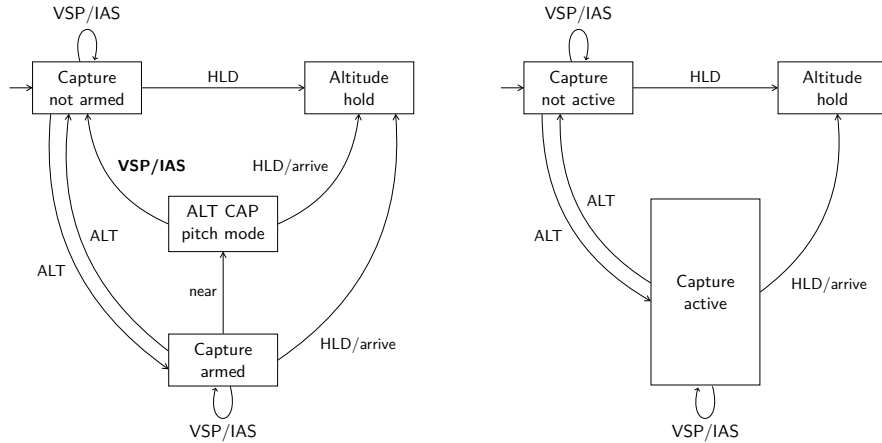
(a) State-machine for the actual system. (b) State-machine for the mental model.

Figure 2.4. State-machines representing the actual behaviour of the autopilot, and the one for the pilot. The transition to the ALT CAP mode is automatically triggered by the autopilot when the airplane is approaching the target altitude (near). The arrive transition is triggered when the airplane reaches the target altitude [RCP99].

The lesson learned from that incident is that the pilot has to know about the difference between ALT and ALT CAP capture modes. The reason is that the behaviour resulting from triggering the same action (entering VSP or IAS) is different depending on the actual capture mode. More precisely, the pilot has to be made aware of the silent event near. An explicit indicator on the user interface may be a solution; another possible solution is a redesign of the system.

Using a Model Checker to Automate Mode Confusion Analysis

The mode confusion analysis performed by manual inspection in [RCP99] has been extended in [Rus02], so as to automate the analysis using the *Murφ model checker* [Dil96]. The general idea is to write one Murφ model that embodied two set of variables, representing the state of the system and the one of the pilot. The model checker is then used to explore that model which represents the parallel execution of both the system and mental models. The property that is checked by the model checker guarantees coherence between the states of both models.

Figure 2.5 illustrates the performed analysis. Each node represents a state of the exploration, which is composed of the values of the variables in the system (s) and in the mental (m) models. If all the explored states are agreeing on the values of their variables (states of "type" j), the system is guaranteed to be safe according to mode confusion. Whenever one bad state (states of "type" i) is reached, the path from the initial state to this bad state is a scenario which may lead to a mode confusion.
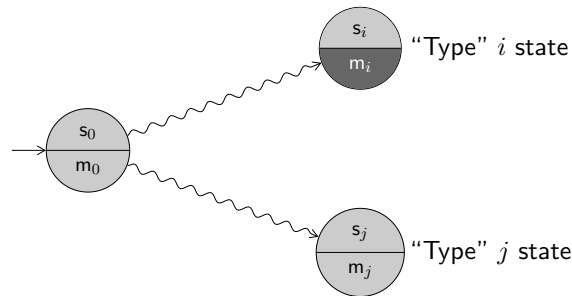
Figure 2.5. Automatic detection of mode confusion using a model checker. A possible mode confusion is detected whenever there is a discrepancy between variables in the system (s) and mental (m) models, during the exploration done by the model checker.

A Murϕ model consists in a set of state variables and a set of rules describing when and how the state variables are updated. The model checker can check invariants (conditions that must always be satisfied) and assertions (conditions that must only be satisfied in some situations). The kill-the-capture case study has been modelled with three state variables:

- two of them model the system: `pitch_mode` is an enumerated value for the pitch mode of the autopilot (VSP, IAS, ALT CAP or HLD) and `capture_armed` is a boolean value representing whether the capture is armed or not.

- and the third one models the mental model: `ideal_capture` represents whether the capture is armed or not in the operator's mental model.

Given the modelling, a potential mode confusion can be detected by stating an invariant exhibiting the fact that the mental model diverges

from the actual system model. Should the mental model be correct, ideal_capture should be true whenever the capture is armed on the actual system, that is, when the pitch_mode is ALT CAP or the capture mode is ALT. That condition is expressed with the following invariant:

> **Invariant** ideal_capture = (pitch_mode = alt_cap | capture_armed)

The Murϕ model checker exhaustively explores all the states of the system and finds a scenario where the invariant is false. That scenario precisely corresponds to the one that the pilot faced in the story presented at the beginning of this section.

The general goal of Rushby et al.'s work is to develop an automated methodology whose aim is to detect whether a given system design is *prone* to mode confusions. The proposed methodology consists in comparing the design of the system with a generic mental model. One issue with the proposed automated technique is that the two models have to be written together in a single Murϕ model, comprising variables for the system and for the mental models. Such a way to proceed is not very convenient. That criticism must be tempered in the sense that what is not convenient in this approach is that both models, with their sets of variables, have to be written in a single model. This has to be opposed to event-based modelling approach where the focus is not on the variables defining the states, but on the actions responsible of state changes. Such models can be defined separately and naturally composed together; and this is precisely the approach taken in this thesis.

Another shortcoming is that in order to detect the potential mode confusion, the potential divergence must be somewhat a priori identified to be stated as an invariant formula. Finally, a difficulty with that approach is to determine a good generic mental model, so as to be able to detect potential mode confusion. The quality of the mental model determines the quality of the potential mode confusion situations found by the approach.

Explicitly Separating the Actual System Model and the Mental Model

The work of Rushby et al. has been extended by Buth [But04] in order to explicitly compare the actual system model with the operator mental model. The analysis is done with the *FDR2 model checker* [For10] which permits to directly compare two models, as opposed to Murϕ which does

not allow that. The FDR2 model checker uses a machine-readable version of CSP [Hoa85] as a specification language. The comparison between models is done with the refinement properties of CSP models. The idea of this work is to define two CSP processes, one for the actual system and one for the operator. The FDR2 model checker is then used to check whether the two processes are equivalent. FDR2 can check for traces and failures equivalences, that are both considered by Buth in her work. *Trace equivalence* is first used to check whether both models are able to perform the same sequences of events. Even if both compared models can execute the same sequence of events, it may be the case that one of the models could refuse an event which cannot be refused by the other model. *Failure equivalence* can precisely detect such situations and is thus also used in the approach proposed by Buth.

One of the contributions of Buth's work is the ability to describe separately the actual system model and the operator mental model. It is however still necessary to have identified a priori where potential mode confusion can occur in order to focus on the right part of both models. Another difficulty resides in the analysis of error traces produced by the FDR2 model checker. It is not easy to express the error in terms that are meaningful at the human-machine interaction level. One of the reasons for that difficulty is that error traces provided by the model checker has to be executed on the models to understand if they are due to wrong modelling choices or to a real situation that may occur. Some noise may also have been introduced by spurious actions introduced only to make the analysis possible by the model checker.

Both works from Rushby et al. and Buth confirmed that model checking provides a convenient approach to investigate models for mode confusion situations. The proposed methodologies are not fully automated but provide a well-defined framework for analysing possible mode confusion in human-machine interactions. As summarised by Buth [But04], two main issues remain: how can the specifications for the system model and the mental model be derived in a systematic way and how errors found during the analyses can be related to situations in the system. The work presented in this thesis addresses the first question; the second one being discussed as perspective for future work.

2.4.2 User Interfaces Analyses Based on Metrics

Thimbleby is also investigating the use of formal methods to analyse
human-machine interactions [Thi10, TG07], but with a quite different
approach from Rushby's. The focus of the analysis performed by Thim-
bleby is put on the description of the interactive system itself, without
trying to model the user in any way, at least explicitly. The model of
the system must be as precise as possible, a rough abstraction is not
enough to have a good analysis, especially for safety-critical devices.
Furthermore, the focus is put on modelling and analysing the behaviour
of the user interfaces, more than the actual underlying system.

Graph-based Analysis

The choice in Thimbleby's work is to use *graph theory* [Wes00] as the
formal framework to analyse human-machine interaction. Graphs are
very suitable since they naturally provide an interpretation of systems at
the level of human interaction. For example, a path in a graph represents
a sequence of actions by the user, and the shortest path between any
two states represents the most efficient way for an operator to achieve
a given goal. That example also shows how properties on graphs could
be related to *usability properties* of interactive systems. In the work
presented in [TG07], Thimbleby et al. propose to use labelled directed
multigraphs to model systems. Based on that modelling, the authors
derive usability properties based on properties and metrics on graphs.
Some of them are summarised hereafter.

- The first concern is about *navigability*, that is, the ability for the
  user to navigate through the state space of the device. The strong
  connectivity ensures that every state of the graph is reachable from
  any other state if the graph has exactly one strongly connected
  component. The diameter of the graph measures the difficulty
  for the user to perform the worst task (the one with the greatest
  number of actions); where a task is defined as a path between two
  states of the graph.

- Another concern is evaluating the *cost of various error scenarios*.
  If the user engages in an action mistakenly, one interesting measure
  is the evaluation of the undo cost if the user wants to cancel the

last executed action, or the overrun cost if the user triggered an action once too often. Those costs can be measured by computing an average on some set of shortest paths.

- A third concern is about *knowledge about the system*. Those metrics aim at measuring whether an interactive system is easy to learn for a human. The minimum cut in a graph is the smallest set of transitions that, when removed, make the graph disconnected. The interpretation is that if the user does not know those transitions, there will be some functionalities that will never be used. Actions corresponding to such transitions must be focused in well-designed training materials.

- A last concern is about what the user can observe about the system. The *chromatic number* χ is the minimum number of colours to be used to colour the states of the graph so that no two states linked by a transition have the same colour. If the user must always know that the system has changed state, it must be provided with a number of indicators that is at least equal to the chromatic number (or to be more precise, with a set of indicators that can take χ different states altogether). Such a system is said to be trackable, meaning that the user can keep track of which state the system is in. Finally, a user that claims to know completely a system should know a Chinese postman tour of the graph, that is, a shortest tour that visits every transition of the graph.

Advantages of graphs is that they are simple to build and understand and they are very visual, in comparison to other formalisms such as BNF grammars, for example. Moreover, their properties can be easily related to the human-machine interaction language, to provide relevant analyses. One difference with the work of Rushby is that there is no need to think about what could be a good mental model since the performed analysis are targeting general interaction concerns. The drawback is that it is not possible to capture more dynamic aspects of the interaction, such as to analyse mode confusion potentials for example. Another weakness, stated in [TG07], is that graph models may be non-deterministic, but such situations have not been considered in this work since it may complicate many of the used metrics. The work presented in this thesis also relies on a simple formalism that is essentially a graph enriched with

elements on the states and transitions so as to be able to capture more dynamic aspects. Moreover, the user is explicitly taken into account and non-determinism in the system is directly taken into account.

2.4.3 Using Model Checking to Verify Usability Properties

Campos et al. are in some way in-between the two previous presented works. The authors use model checking techniques, as Rushby does, to check that a given system satisfies some properties such as usability, as used by Thimbleby. Campos et al. are using interactors and the MAL modelling notation to model systems [CH11, CHL04, CH08].

Model Checking Interactors

In [CH11], Campos and Harrison present *interactor*-based specifications [FP90, DH93] and the role they can play as a partial model of interactive systems in automation surprises analysis, especially to detect them early in the design process of interactive systems. Interactors are objects with a state and operations, but for which perceivable states and actions are explicitly identified. The authors emphasise that interactors bring nuances in the verification of interactive systems that differentiate from the more general problem of software verification.

An interactor, as illustrated on Figure 2.6, is an object which interacts with its environment through events. It is also capable of rendering (part of) its state. That is, an interactor is composed of three main components: a state, a behaviour and a rendering. There is no one prescribed notation to specify interactors and several have been used such as Z, *modal action logic* (MAL) and VDM.

In the work of Campos et al., the states of interactors consist of a set of typed attributes. The behaviour of the interactors is defined with axioms and a logic based on structured MAL [RFM91, DBMD95]. MAL contains the usual propositional operators and it adds a modal predicate $[a]Q$ meaning that Q is required to hold in any state that results after performing the action a and two deontic operators $\mathbf{per}(a)$ and $\mathbf{obl}(a)$ respectively meaning that the action a is permitted or obliged. Rendering aspects are defined by offering the possibility to annotate actions and attributes, to define that they are perceivable by the user. Interactors can be included within others in order to build more complex systems.
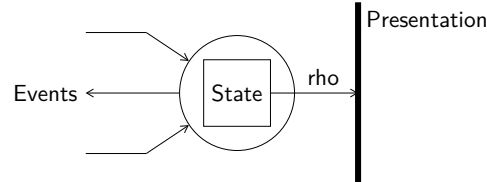
Figure 2.6. A York interactor is an object which interacts with its environment through events and which is capable of rendering (part of) its state in some presentation medium [DH93].

Figure 2.7 shows an interactor example for a very simple system which is a lamp that can be turned on and off by pressing on a button.

interactor lamp
attributes
    [vis] light: **boolean**
action
    [vis] press
axioms
    (1) [press] light' = !light

Figure 2.7. Example of a simple interactor representing the model of a lamp that can be turned on and off with a single button.

In addition to this modelling approach of interactive systems, Campos et al. developed algorithms to translate them into a SMV specification, the language used by the *SMV model checker* [McM00]. An *SMV specification* is a collection of SMV modules, each of them defining a finite state-machine. An SMV module is basically composed of state variables with a set of rules specifying how the module progresses from one state to the next one. The analogy between the state variables and the interactor attributes and between the set of rules and the interactor axioms is immediate, as shown by Figure 2.8 showing the SMV specification that is obtained after translation of the lamp interactor example.

Invariants can be stated in interactors and will be checked on the obtained SMV specification by a model checker. The methodology has been applied on the kill-the-capture case study of Palmer and it was also able to detect the potential mode confusion, just as it is the case with Rushby's approach. The advantage of the approach developed by

```
MODULE lamp
VAR
  light:  boolean;
  action:  {press};
INIT
  light = 0
TRANS
  next(action) = press -> next(light) = !light
```

Figure 2.8. SMV specification obtained after translation of the lamp interactor example of Figure 2.7.

Campos et al. is the modelling language that makes it possible to describe the model of a system in a modular way, with state variables and with the focus on actions and the dynamic behaviour it implies. This is exactly the same kind of expression that is captured by the modelling approach chosen in this thesis.

Checking Generic Usability Properties

In [CH08], Campos et al. use a model checking approach to automatically verify generic *usability properties* on a given system. The authors express usability properties as generic CTL formulæ that are instantiated for a particular model. The CTL properties are then checked against the system with the *NuSMV model checker* [CCJ$^+$10]. This is precisely where the approach of Thimbleby and the work by Campos et al. just presented here above are merged together. The HMI analyses that are performed only concern the system, without explicitly including the operator.

One example of a generic property is the following:

$$\mathbf{AG}(pred(s) \land c =_* x \implies \mathbf{AX}_a(c \neq_* x))$$

That property represents the *feedback usability property* which states that whenever the user performs an action on a system, there is always a perceivable change in some of the visible attributes of the system. The $pred(s)$ predicate can be used to add a condition to the property, c is a set of attributes, the $=_*$ operator is equality distributed over the attributes and $\mathbf{AX}_a\, p$ is a shortcut for $\mathbf{AX}(a \implies p)$, that is, in all next states arrived at by action a, p holds.

The case study that has been used by Campos et al. is the air conditioning panel of the Toyota Corolla (2001 European version). As illustrated by Figure 2.9, the user interface consists of ten buttons and a display showing three elements. Perceivable attributes of the system are the three values shown on the display (flow mode, fan speed and target temperature) and the four boolean attributes corresponding to some of the buttons at the bottom (air conditioning mode selection, windscreen front flow mode selection, air intake mode selection and automatic mode). In addition to those visible attributes of the system, there are also some hidden attributes such as some values that are memorised when the system is switched off.
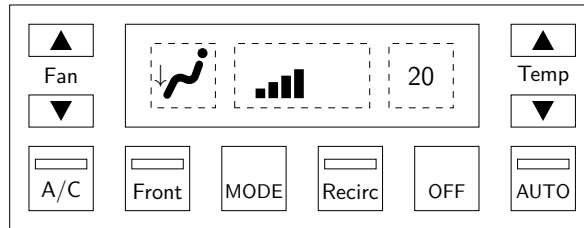
Figure 2.9. Control panel of the air-conditioning system of the Toyota Corolla (2001 European version). The four buttons A/C, Front, Recirc and Auto at the bottom are used to activate or deactivate some options. The MODE button is used to switch the operating mode. The OFF button is used to switch the air conditioner off. The button with arrows on the left (resp. on the right) is used to increase or decrease the speed of the fan (resp. the target temperature). The front display shows the flow mode, the fan speed and the target temperature [CH08].

The tricky part of this system is the flow mode which is not simply controlled by one unique button, as opposed to the fan speed and target temperature (which are in fact controlled by two well identified buttons). There is a total of five different flow modes which may change after a press on the A/C, front or MODE.

The generic feedback usability property can be instantiated to the visible attributes of the system. The generic property can for example be instantiated for the mode key, knowing that the feedback will be provided by the flow mode indicator on the display:

$$\mathbf{AG}(airflow = x \implies \mathbf{AX}_{modekey}(airflow \neq x))$$

The property will be instantiated for all the five possible values of the *airflow* and checked on the system. For that case study, it is indeed satisfied. However, if the *fanspeed* attribute is checked regarding the feedback usability property, the check fails. Nothing visible changes when the maximum speed is already reached and when the user is trying to augment the speed indeed. Such a situation may or may not induce a bad interaction; further analysis is left to the designer of the system, as stated in [CH11]:

> "A property of concern may not represent a failure for the inter-active system rather it may highlight scenarios where special care should be taken to understand how the user will interact with the system at this point."

Other generic usability properties are defined in [CH08] and the authors provide a collection of them, well documented, that can be used by the system designers without much knowledge about model checking and CTL temporal logic.

The work presented in this thesis is not concerned directly with usability properties. It defines a more general property that can be used to identify potential mode confusion, but more generally the property captures situations of potential automation surprises for the operator. Nevertheless, integrating usability properties as those defined by Campos et al. may bring additional constraints to analyse whether a given mental model is suitable to operate a system; and the idea of proposing a library of properties helps to make the formal methods usable by system designers, overthrowing the limits to adopt formal methods based techniques.

The IVY Tool

The *IVY workbench*[1] [CH08, CH09] is a model based tool that is used to perform formal analyses on interactive systems design. The purpose of that tool is to detect usability problems early in the design and development process. Models to be analysed are specified with the MAL interactor language and then compiled into the SMV language. The verifications that are performed by the tool are done with the NuSMV model checker.

---

[1]Available online at: http://ivy.di.uminho.pt/.

IVY is composed of several tools:

- The *model editor* is used to express the model of the system to be analysed, using the MAL notation. A model is composed of a set interactors, each composed of attributes defining the states, actions and axioms describing the state transitions. Two possible edition modes are available in the editor: a textual one and one based on a graphical notation inspired from UML class diagrams.

- The *property editor* is used to define the properties to be checked on the model. Those properties are expressed in CTL. The editor proposes a set of predefined generic usability properties which are specialised by the user in order to make them relevant for the analysed model.

- The *AniMAL plugin* is used to build user interface prototypes from MAL models. The prototypes take the form of a simple GUI with one panel for each interactor, showing the different attributes and actions. AniMAL can animate a sequence of events which helps the designer to get a more textured indication of what went wrong the case of failure, for example.

- Finally, the *verifier* is used to check the properties on the model. It uses i2smv to compile MAL interactor models into SMV models. Properties expressed thanks to the property editor are then checked by the NuSMV model checker. If the model does not satisfy the properties, the counterexample provided by NuSMV is analysed so as to provide a relevant representation for the designer.

2.4.4 Cognitive considerations for the human behaviour

The research presented in the previous sections either view the human as an automaton, state-based, or they do not consider explicitly the human operator. There is also lot of research more focused on the formalisation of more precise *user behaviour models*. Capturing the human behaviour with a formal model is not easy due to high variance in population and its dependence on context and situation [Suc87].

But a realistic simplification focusing on achievement of goals can be stated as follows [CseB07]:

> "It is a reasonable, and useful, approximation to say that humans behave rationally in the following sense. They enter an interaction with goals they try to achieve and have some knowledge that seems likely to help them achieve those goals."

One of the main reasons to be interested in cognitive considerations for the human behaviour is that it helps to better understand user errors and consequently to get a better analysis of human-machine interaction. According to Reason [Rea90], whole classes of systematic errors may occur due to cognitive causes, for certain types of interactive system designs.

Detecting Systematic Human Errors

Curzon et al. [CseB07] focus on the definition of a cognitively plausible human behaviour that is used, once combined with the design of a device, to detect potential erroneous actions and thus potential bad design choices. Unlike the other approaches that consider the human as perfectly following a given behaviour, Curzon et al. consider human errors that are essentially cognitive slips. The detection of systematic human errors due to cognitive causes is precisely the core of their approach. The user and the system are explicitly modelled separately with a higher-order logic formalisation. The verifications are performed with the *HOL interactive proof system* [GM93, For12].

Analysing Different User Behaviours

Basuki et al. [BCGS09] developed an approach that is used to analyse different user behaviours against a given system. They model human-machine interaction with interacting components (human and machine actors) that act like peers communicating together through a particular interface component. The interacting components are represented with *labelled transition systems* (LTS) which are essentially graphs whose transitions are labelled with actions. Those LTSs are modelled with the *Maude rewrite system* [CDE$^+$03]. The advantage of using a rewrite

system is that it makes it possible to define one user model that can be refined to get new user models just by adding new rewrite rules.

In Basuki et al.'s approach, users are modelled either as goal-based users or reactive users. The first category represents users that have a predefined goal they want to perform with the system and that will drive their interactions. The second category corresponds to users that will interact with the system in response to the signals it sends. Those different user behaviours are then combined with the system model and properties such as goal completion or flexibility are specified as LTL properties and verified by the model checking capabilities of Maude.

The assumption made by the author to model users is that they are acting like rational users, that is, users who want to achieve a goal and, for that, will perform actions. Random user behaviours are thus excluded from their models. Basuki et al. characterise the user behaviours according to several aspects. First of all, the user has a goal to achieve by interacting with the system, after which he will leave the interaction. To do so, the user has to perform operations to achieve the goal and must be able to react to stimuli produced by the system. Three other aspects are taken into consideration:

- The user may get used after having interacted several times with the same machine or similar ones. The user may just ignore stimuli from the machine and directly perform the sequence of actions to achieve his goal.

- The user has also a certain time tolerance, that is, he does not want to wait to the machine being processing information for a time longer than his tolerance time. After a too long wait, the user may suspect that something is going wrong and thus he may redo the last performed action.

- Finally, the user is acting carefully. It means that the user is not willing to loose any of his possessions if it is not required to loose any of them to achieve his goals. That third aspect influences the user when he is tempted to redo an action.

All those considerations are taken into account in the design of the rewriting rules used to specify the user behaviour model. In their paper, Basuki et al. illustrate their approach with a vending machine example. Figure 2.10 shows the interface of the chocolate vending machine. There

are two lamps that can light up to indicate the user that the machine
expects some action. There is also one button that allows the user to
select chocolate. Finally, there are three slots that are respectively used
for the user to insert one-euro coins, to get back the change and to take
the chocolate bar.
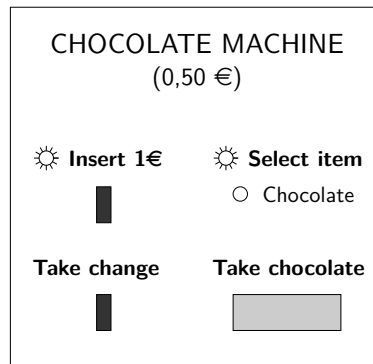
Figure 2.10. Interface of a chocolate vending machine [BCGS09]. One bar of chocolate
costs fifty eurocents and the user can only pay using one-euro coins. After having
inserted one euro, the user has to make a selection (only chocolate is possible) and
then can take back his chocolate bar and the change. The user is guided thanks to
two lamps that indicate him what the machine is waiting for.

Three models are necessary to model the interaction with the vending
machine: the *machine model*, the *user behaviour model* and the *interface
model* which is the communication medium between the user and the
machine. One basic property that can be checked is that the user is
always able to reach his goal, which can be expressed with the following
LTL property:

$$\mathbf{G}(\textit{Approaching} \implies \mathbf{F}\,\textit{GoalAchieved})$$

The property states that whenever the user approaches the machine
with the intention to interact, he eventually manages to achieve his goal.
That property is very general since it does not state whether the user
has to achieve his goal directly or whether he can leave and come back
later. Another stronger version of the property can be defined:

$$\mathbf{G}((\textit{Approaching} \wedge \textit{Enabled}) \implies (\neg\,\textit{UserLeft}\ \mathbf{U}\,\textit{GoalAchieved}))$$

The property states that whenever the user is approaching the machine and that the machine is working properly (that is, still has chocolate bars to sell and fifty eurocents coins left) the user does not leave the machine unless his goal is achieved.

These properties, and all the others described in [BCGS09] are generic properties that can be used to check whether a user is able to achieve his goal during an interaction with a machine.

2.4.5 Integrating User's Tasks into the Interaction Analysis

Up to now, the presented works do not take *user's tasks* into account in the interaction analyses performed. Such a consideration is useful for the human-machine interaction analysis since human factors engineers use task-analytic methods to describe the normative human behaviours that are required to control the system to be operated [DS03]. Those *task models* can be viewed as a kind of mental model, containing the mental and physical activities that are carried out by the operators. Formal task-analytic models have been developed and include *ConcurTaskTrees* (CTT) [PMM97], *operator function models* (OFM) [MM86] and *user action notation* (UAN) [HSH90]. Those modelling approaches represent the task models as graphs representing hierarchies of activities that are decomposed into sub-activities and so on until those sub-activities are reduced to atomic actions. The execution of the different sub-activities are controlled by conditions.

Bringing Together Tasks and System Models

Palanque et al. [PB97, NPB01, BNP03, NPLB09] defined and developed *interactive cooperating objects* (ICO), an object-oriented formal dedicated to the modelling of interactive systems. The ICO formalism is built upon the object-oriented Petri nets paradigm. An object consists in four parts: a cooperative object, a presentation part, an activating function and a rendering function. ICO is a genuine combination of concepts from the object-oriented approach for the description of the structure of the interactive systems and high-level Petri nets to describe the dynamic behaviour. *Petri Nets* (PN) [Pet62, Mur89] is a formalism that has both an algebraic and a graphical representation, which makes it possible to

perform mathematical analyses on them and which is intuitive and easy to model thanks to the graphical representation.

The ICO formalism aims at providing a rapid prototyping way of designing user interface. All the parts of an ICO model are thus related to some part of the application. *Cooperative objects* are expressed with *object petri nets* (OPN), a variant of Petri net where tokens are complex objects defined in some object-oriented language. They encode how the objects react to external stimuli, according to their current state. Based on that, the *presentation part* defines the external appearance of the object, which can be a window or a single widget. Then the *activation function* part associates inputs from the user on the interface to the corresponding cooperative object. Finally, the *rendering function* associates a change of state of the cooperative object to an output to the user. This decomposition approach is very similar to the one captured by York interactors, presented above.

One advantage of the ICO formalism is that, in addition to the ability to describe the dynamic behaviour, it also makes it possible to define the structure of the system with the object-oriented concepts such as dynamic instantiation, encapsulation and inheritance. Another advantage is that user's tasks can be directly described with ICO. The analyses that are available come directly from the mathematical tools provided by the Petri net theory [PB95], and supported by a tool.

The *CASE tool Petshop*[2] [NPLB09] is a Java-based tool which is used to support the modelling of systems with the ICO formalism. Petshop offers tools to perform classical manipulations on Petri nets, in order to build the model. A great advantage of Petshop is that it is possible to execute simultaneously the interactive application and its ICO underlying model. Furthermore, the ICO model can be modified while the application is running and the changes are directly passed on the interactive application.

Formal analyses of interactive systems can be done with the analyses module. It works by first translating the ICO models into their underlying Petri nets. Standard analyses such as deadlock or invariant analyses can then be performed on the models. It is also possible to perform richer analyses by checking the system against properties expressed in ACTL. For those analyses, Petshop relies on the CPN Tools.

---

[2]Available online at: http://www.irit.fr/recherches/ICS/softwares/petshop/.

Modelling Human Errors with Tasks Models

Bolton et al. [BBS08, BSB11] are going one step further in the analysis of human-machine interaction. In their work, they are also modelling explicitly users' tasks with *enhanced operator function models* (EOFM) which is a generic task-analytic modelling language. Based on those task models, and on models of erroneous human behaviour, they developed a methodology to identify potential interaction issues.

The general framework was initially described in [BBS08] and illustrated with a model of the *Therac-25*, a medical device that is used to treat tumours by patient and that has been involved in accidents that led to death, due to human errors [LT93]. The authors extended the classical model checking process (Figure 2.2 on page 21) by adding a model of human error that is combined with the model of the system to get a single model that can be fed into a model checker for analysis.

Figure 2.11 shows a statechart representing the formal system model of the Therac-25 that has been used by Bolton et al. The system is composed of four concurrent *finite state-machines* (FSM). The left FSM represents the interface which allows the operator to select the kind of beam that will be administered: electron beam is used for shallow tissue treatment and X-ray is for deeper treatments. For the X-ray mode of operation, a spreader has to be put between the beam and the patient. The bottom middle FSM models the spreader which is mandatory for the X-ray mode, or the patient may get a lethal dose of radiation. The top right FSM represents the beam that is effectively selected. One peculiarity is the two 8s transitions which do not correspond to an action of the user, but that occur after a time lapse of eight seconds. Finally, the bottom right FSM represents the firing of the beam. Again, there is an automatic reset transition that takes place whenever the beam has been fired.

Many problems have been found with the Therac-25. The most notorious occurred when a benign human error caused the machine to administer X-rays to patients, without the spreader in place, giving rise to important consequences. The interaction took place as follows [LT93, BBS08]:

> "The operator wanted to apply an electron beam treatment to a
> patient. He first erroneously pressed the selectX key. The operator
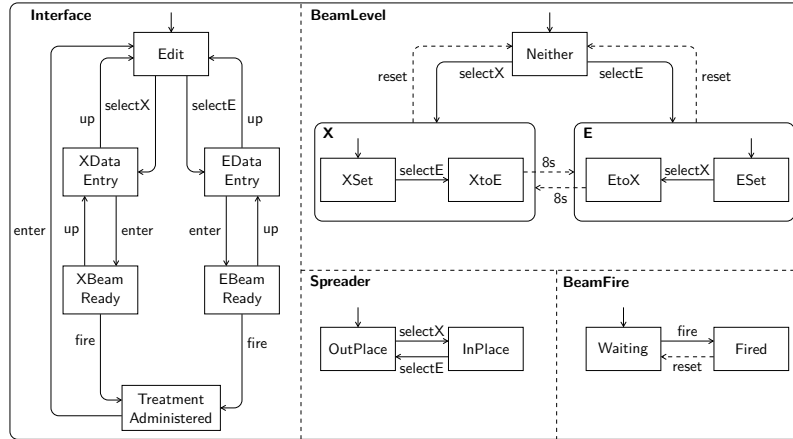
Figure 2.11. A statechart representing the formal system model of the Therac-25 machine that has been used by Bolton et al. [BBS08]. The model is made of four concurrent finite state-machines. The 8s transitions of the BeamLevel FSM is in dashed-style since they represent automatic transitions that take place after eight seconds. Similarly, the reset transition of the BeamFire FSM takes place after the beam has been fired.

> directly reacted by pressing the ↑ key, so as to take the interface back into the initial state. He then successively pressed on selectE, ↵ and then fire. The Therac-25 applied an X-ray treatment, but without the spreader in place which caused a lethal dose of radiation for the patient..."

The issue that caused the accident is related to the beam level. The mode was initially set to X-ray erroneously, which makes the BeamLevel FSM to be in the XSet state. If the operator performs the procedure to recover from his error to set the machine in the electron beam mode and fires, in less than 8 seconds, then the actual mode of the machine will not have time to change from X-ray to electron beam and the patient will get X-rays without the spreader in place.

Bolton et al. explain how that potential bad interaction may have been predicted with the approach they propose in [BBS08]. The first step of the analysis if to provide a normative *human behaviour model*, describing the task to be executed. For the Therac-25 case study, this task corresponds to the administration of an electron beam treatment. The human behaviour model is described with the OFM formalism and

basically corresponds to the sequential execution of the following four actions: selectE, ↵, fire, ↵. The second step of the analysis is to define the specification that has to be satisfied. The following CTL formula states that there will never be a state where X-rays are administered without the spreader in place:

$$\mathbf{AG}\neg(BeamLevel = X \wedge Spreader = OutPlace \wedge BeamFire = Fired)$$

The third step is to derive an erroneous human behaviour model, from the normative one and the formal specification of the system. The common type of error that is of interest here is the execution of a familiar action (selectX) at the inappropriate time, known as Reason's slip [Rea90] or Hollnagel's erroneous act [Hol93]. Finally, the last step is the combination of the erroneous human behaviour model with the system model so as to get a single model that is analysed by the *SAL model checker* (Symbolic Analysis Laboratory) [BGL$^+$00]. The specification failed to be satisfied and the provided counterexample corresponded exactly to the described accident.

Bolton et al. continued to work on using task analytic models to analyse human-machine interaction [BSB11]. The authors proposed a systematic approach for the analyses, based on EOFM, an extension of the OFM notation which can be automatically translated to be analysed by the SAL model checker.

2.4.6 Human Factors Considerations

In addition to human-computer interaction and formal methods communities, there is another field that is heavily involved in the study and analysis of HMI, namely human factors. *Human factors* is an area of psychology interested in different topics including ergonomics, human error and human capability.

The *International Ergonomics Association* (IEA) defines human factors as follows [3]:

> "Ergonomics (or human factors) is the scientific discipline concerned
> with the understanding of the interactions among humans and other

---

[3]http://www.iea.cc/01_what/What%20is%20Ergonomics.html.

> elements of a system, and the profession that applies theoretical
> principles, data and methods to design in order to optimise human
> well being and overall system performance."

Researchers from the human factors community also started to investigate the use of formal methods to analyse HMI. That opened new directions for collaborative work for the prediction and analysis of the behaviour of interactive systems, for example in the aeronautical or medical domains [CJR00]. Human factors researchers started to develop models of automation behaviour, for example the OFM formalism, *operational procedures table* (OPT), control block diagrams with mode transition matrices or diagrams of mode transition conditions. Those descriptions, relying on the idea that system behaviour can be modelled as finite state-machines, were a springboard toward the formal methods community. In order for formal methods techniques to bring relevant additional support for the design of interactive systems, the human factors needs must be understood and integrated.

Two examples illustrating how and where formal methods should intervene in the system design process are described in [CJR00]. The first example concerns the already introduced *detection and avoidance of automation surprises*. Human factors considerations suggest that those automation surprises can be related to incomplete or approximate mental models of the system behaviour [LPS+97]. Model checking can help finding bad potential scenarios as studied by Rushby [Rus02] for example. An additional point to be raised is that experts from human factors are necessary when developing the system and mental models to be analysed, and especially to get the right abstraction containing the behaviour which is relevant for the analysis.

The second example is about situations where an *incomplete mental model* may have an impact on safety issues. The presented study is about determining a so-called *minimal mental model* which is required to operate the A340-200/300 autopilot safely and proficiently. For that case study, formal methods are used to validate whether a mental model, obtained by questioning instructors and pilots, is both minimal and safe relative to an autopilot design.

The work by Javaux [Jav02] put the stress on some specific aspects of the formal modelling of systems, that are illustrated on an autopilot. The first key aspect is the notion of *transition scenarios* which capture

the different possible independent circumstances under which a transition can occur between two states of the model. Those transition scenarios are more meaningful from a psychological and operational point of view. Figure 2.12 shows an example of a transition scenario for the transition between the autopilot disengaged state to the engaged one.

AP engagement in flight
AP1 or AP2 is engaged in flight by pushing the respective pushbutton on the FCU
(A/C airborne for 5s) **and** (IAS within VLS and VMAX) **and** (pitch within $-10°$ and $22°$) **and** (bank less than $40°$) **and** (AP pushbutton pushed)

Figure 2.12. One of the transition scenarios for the autopilot of the A340-200/300, for the transition between the autopilot disengaged state to the engaged one. The transition scenarios consists in a name, a textual description and the conditions for the transition [Jav02].

The second key aspect is the difference between *commanded* and *uncommanded* state transitions. This notion is related to the fact that the transition is triggered by the operator or occurs autonomously without any intention or action from the user. This is a key point to distinguish both kinds of state transitions since the attentional resource the operator has to allocate is not the same for a commanded or uncommanded transition. Moreover, automation surprises are more likely to happen when uncommanded transitions are taking place, since the user may not be paying attention to them or the user interface may not be complete enough to inform the user.

The main lesson learned from the human factors community is that formal methods can bring automated and replicable methods and tools to analyse human-machine interaction and search for bad interactions potentials, early in the design process. Formal methods bring a complementary view to human factor issues to the view of the HCI community. In the reverse direction, human factors considerations help the formal methods community to develop psychologically interesting models and drive the analysing methods to use and develop.

2.4.7 Automatic Generation of User Interfaces

As highlighted by Degani et al., accidents that happened when machines
are used by an operator can be attributed to a bad interaction that can
be caused by a lack of mode awareness, mode confusion or automation
surprise. Two factors are repeatedly cited in literature: either the issue
is due to the interface which shows inadequate information about the
machine status, or the operator has an inadequate mental model of the
machine being used. All the research presented so far is focusing on the
first factor, while the approach by Degani et al. rather focus on what
information is shown to the operator, and not on how the information is
presented.

Degani et al. pioneered the analysis of human-machine interaction
focusing on the second accident factor, that is, the properties of the model
the operator has about the machine being used. Their work consists in
automatically generating an adequate mental model for a given system
so that an operator using this mental model is guaranteed to always
interact adequately with the machine. Their proposed formal procedure
is used to assess the reliability of the interaction between the operator
and the machine so as for the operator to be able to achieve specified
operational goals.

The approach focuses on the mode confusion issue. They observed
that in most accident cases, the operator had difficulties to anticipate
the next configuration of the machine, or its *mode*. A mode is a set of
states of the machine which exhibits a specific behaviour. Four elements
are taken into consideration in their work:

- the *machine model* represents all the behaviour of the machine,
  that is, all the possible executions of the machine;

- the *task specification*, or operational goals, represents what task
  the operator will perform on the system;

- the *user interface* provides information to the operator about the
  state of the machine and its responses;

- and the *user model* represents the view that the operator has about
  the machine.

The behaviour of the machine is supposed to be deterministic by
hypothesis, that is, its response to every action by the user or to external

signals is unique and unambiguous. The task specification can be for example the execution of a sequence of actions (procedure), the monitoring of the mode changes of the machine or the prevention from reaching illegal states. The user interface is an abstraction of the system and it only shows monitored events to the operator. The user model is also an abstraction of the machine but its events are those coming from the user interface. There are three kinds of events that can occur on the machine:

- the *observed events* can be seen by the operator and are either under the control of the machine (caused by internal dynamics or external environment) or by the operator;

- the *masked events* are events that the operator cannot distinguish and sees as a single other event;

- and the *unobserved events* are events that the operator cannot observe and is not aware of.

Figure 2.13 illustrates the three models involved in Degani et al.'s approach. The different kinds of events are also shown. Events α and β are undistinguishable for the operator and are rendered as the unique μ event by the user interface (masked events). Event ν is observed by the operator (observed event) and event τ is unobserved and completely internal to the machine (unobserved event).
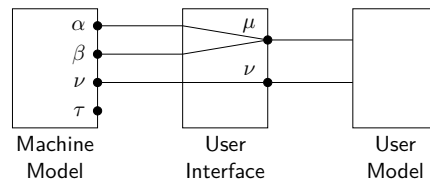
Figure 2.13. The three models involved in Degani et al.'s approach.

Degani et al. also use *finite state-machines* to model the machine and the user model. They propose an approach for analysing the interaction by considering that the task specification is that the operator is always able to determine unambiguously the current and the next mode of the machine. The states of the machine are partitioned into specification classes which represent the different operational modes of the machine.

The interaction between the machine and an operator is represented by the parallel execution of the machine model and the user model. That parallel execution can be represented as a finite state-machine whose states are composite states representing the state the machine is into and the current state of the user model. From that interaction model, two kinds of inadequate interaction can be identified:

- An *error state* corresponds to a situation where the machine can move into an illegal state while the user thinks that the transition is legal. The reverse can also occur, that is, the operator thinks a move is illegal while it is in fact legal on the machine. Both situations leads to a situation where the state from the machine and the one in the user model do not belong to the same specification class (or mode).

- A *blocking state* corresponds to a situation where the operator is unaware of certain events that can in fact take place on the system. If those events are automatic events triggered by the machine, it can surprise the operator when occurring.

The methodology developed by Degani et al. is used to perform a systematic abstraction of the machine model. The abstraction is then used to build an interface for the machine, by capturing exactly what are the operational modes the operator must be aware of. The built interface must be correct and succinct. The *correctness* criterion ensures that the operator will be able to perform the specified task correctly. The *succinctness* criterion ensures that the abstraction is small enough so that it can fit into the operator's memory. The *machine model reduction* problem consists in finding all the best possible user models, best in the sense that those models cannot be further reduced, an guarantees no mode confusion potentials.

The work presented in this thesis directly follows the one of Degani et al. [DH00, HD02, DH02, HD07] The goal of the work is to start from a given system model and to automatically generate a user model from it. The focus in this work has been moved from the precise mode confusion problem to the more general problem of automation surprises. For that purpose, this work defines the full-control property, that must hold for all the possible interactions between a user model and a model of the

system being operated. In opposition to Degani et al. who are moving towards the use of higher level formalism, namely statecharts, this work focuses on a lower level mathematical formalism, but towards which models described in other formalisms may be translated. Adachi et al. [AUU06] formalise the approach described by Degani et al. focusing on the generation of an adequate user interface from a given system model. The work of this thesis focuses on the conceptual model and extends the adequacy notion used by Degani et al.

2.5 Context and Discussion

Different approaches where formal method techniques are used to analyse human-machine interactions are described in Section 2.4. Table 2.1 on page 54 summarises the main formal methods based techniques to analyse human-machine interaction presented in the previous section. In this section, reference to the different research will be done only by mentioning the first author. The section lays up the context and the direction followed in the work presented in this thesis and also discusses the choices that has been done and the frame in which this work is in, in relation to the state of the art.

2.5.1 Human-Machine System

In the work presented in this thesis, a human-machine system is only composed of *one machine* being used by *a single operator*. The only direct interactions that are considered are between the user and the machine. The environment is not explicitly considered but implicit interactions with the system may be taken into account, for example if a temperature sensor would provide events in the system. Unlike Palanque, user's tasks are not taken into account directly. Chapter 6 presents an experiment taking tasks into account and gives future direction about how it could be included as a separate component in the analyses proposed in this thesis.

The behaviour of the system is described as a discrete reactive system. Even if objects in the world are by nature continuous, automation is getting digital and thus discrete by nature. Moreover, only finite-state

Author(s)	References	Inputs	Analysis
Rushby et al.	[Rus02]	One Murφ model (for the system and the user) and assertions or invariants	Mode confusion
Buth	[But04]	Two FDR2 models (one for the system and one for the user)	Automation surprises
Thimbleby et al.	[TG07]	A graph model for the system	Usability properties
Campos et al.	[CH11]	A interactors-based model of the system and invariants	Mode confusion
	[CHL04, CH08]	and CTL properties	Usability properties
Curzon et al.	[CseB07]	HOL models	
Basuki et al.	[BCGS09]	Maude model	User behaviours comparison
Palanque et al.	[PB97, NPB01, BNP03, NPLB09]	ICO model (Petri-net based model)	Task analysis
Bolton et al.	[BBS08, BSB11]	EOFM	Task analysis
Degani et al.	[DH00, HD02, DH02, HD07]	Statecharts	User interface generation
Combéfis	This thesis and [CP09, Com09, CGPF11a, CGPF11b, CGPM11]	HMI-LTS or HVM	Full-Controllability (and automation surprises)

Table 2.1. Comparison of formal methods based techniques to deal with and to analyse human-machine interactions issues.

systems are considered in the work presented in this thesis. Time is not taken into account, in the sense that events are instantaneous and time between events is unspecified.

The modelling approach that has been chosen is similar to the one of Rushby or Javaux, that is, a kind of finite state-machines, but that are described explicitly and completely in the way Thimbleby does it with graphs. The difference is that the models do include the distinction between commanded and uncommanded transitions, proposed for example by Javaux. Finally, as it is the case for Campos, information about attributes of the system is present in the states of the system. This modelling approach is the subject of Chapter 3.

2.5.2 Interaction Analysis

As shown in the review of Section 2.4, different possible analyses can be done on human-machine interactions. The focus of this work is on a safe interaction for the operator with the machine being used, as highlighted by Javaux in [Jav02]. In particular, this thesis deals with the detection and avoidance of automation surprises during the design of the system. Similarly to Rushby, both the system model and a mental model are considered, but modelled separately just as in the extended work of Buth. Usability properties as developed by Thimbleby or Campos are not studied in the frame of this work, but they can be integrated easily since a similar modelling approach as that of Thimbleby is used for the modelling part and since a model checking approach is used as a verification method. The definition of safe interaction, and the algorithm used to check it is the subject of Chapter 4. Finally, user's tasks profusely used by Palanque and Bolton are not taken into account yet, even if Chapter 6 proposes a first direction of integration.

2.5.3 Safe Minimal Mental Model

The human factors community is interested in the notion of a minimal safe mental model, for a given system [CJR00]. This is precisely the focus of Chapter 5. As presented in the next section, the work of Rushby has been extended by Degani et al. in order not only to be able to check a mental model against a system model, but also to automatically generate a mental model with some prescribed properties.

Chapter 3

Modelling

Human-Machine Interactions

This chapter describes the modelling choice that has been made in this work to represent the system and the user. The choice has been made so as to be simple and rich enough to at least represent the aspects necessary to formally check whether all the possible interactions between a user and the machine being operated can take place without potential errors. The modelling choice used in this work has been mainly influenced by the work of Degani et al. [DH02, HD07]. But whereas they are using statecharts [Har87] as the mathematical formalism, the work presented in this thesis is based on enriched labelled transitions systems (LTSs). Section 3.1 provides background information and notations that are used throughout this thesis. Section 3.2 presents the HMI-LTS formalism that is used to model the system and the human, and presents how the interaction between them is computed. Then, Section 3.3 presents the enriched HMI-LTSs which are mainly HMI-LTS with the addition of observable information on the states. It also shows how enriched HMI-LTS can be reduced to HMI-LTS. Finally, Section 3.4 presents work related to the modelling of human-machine interactions, and compares them to the proposed modelling approach.

3.1 Background and Basic Notation

Models that are used to model human-machine interaction in this thesis are based on *labelled transition systems* (LTS) [Mil80, Kat05]. This section gathers definitions, notations and main results related to LTSs, that are used throughout this thesis.

An LTS is essentially a directed graph [Wes00] whose vertices are called *states*, one of those being the *initial state*, and edges are called *transitions*. In this thesis, only finite sets of states are considered. Transitions of an LTS are labelled with a visible *action* or with a τ that represents the internal (invisible) action. The set of visible actions (that is, excluding τ) is called the *alphabet* of the LTS.

Definition 3.1 (Labelled Transition System)**.** *A labelled transition system (LTS) is a tuple $\langle S, \mathcal{L}, s_0, \rightarrow \rangle$ where S is a finite set of states, $\mathcal{L}$ is a finite set of labels representing visible actions, $s_0 \in S$ is the initial state and $\rightarrow \subseteq S \times (\mathcal{L} \cup \{\tau\}) \times S$ is the transition relation, where $\tau \notin \mathcal{L}$ is the label for the internal action.*

Figure 3.1 shows a graphical representation of an LTS example. The LTS example has five states (depicted with boxes and named **A**, **B**, **C**, **D** and **E**), three labels (a, b and c) and eight transitions (depicted with the arrows linking two states). Transitions labelled with the internal action τ are depicted with dotted lines. The initial state is the state **A** and is identified with an arrow pointing on it.
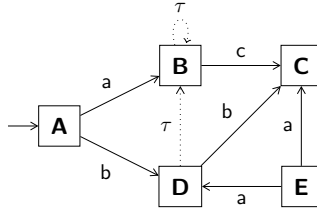
Figure 3.1. Graphical representation of an LTS example with five states depicted as boxes and eight transitions which are the arrows linking states. The initial state is the state A identified with an ingoing arrow pointing it. The alphabet of the LTS is $\mathcal{L} = \{a, b, c\}$.

The LTS example of Figure 3.1 is formally defined as the tuple $\langle S, \mathcal{L}, s_0, \rightarrow \rangle$ where:

- $S = \{A, B, C, D, E\}$;
- $\mathcal{L} = \{a, b, c\}$;
- $s_0 = A$;
- and $\rightarrow = \{(A, a, B), (A, b, D), (B, \tau, B), (B, c, C), (D, \tau, B),$
$(D, b, C), (E, a, C), (E, a, D)\}$.

LTS are similar to *finite state machines* (FSM), *deterministic finite automata* (DFA) or *nondeterministic finite automata* (NFA) [RS59a]. LTSs allow a state to have multiple outgoing transitions with the same label, which is forbidden by FSMs and DFAs. Moreover, the transition relation of DFAs are complete functions, meaning that there is one transition labelled with every value of the alphabet for all the states of the DFA. Also, the notion of accepting states is not present in LTSs that are mainly used to represent the behaviour of reactive systems, whereas the other formalisms are used to represent languages, that is, a set of words on a given alphabet. Finally, *nondeterministic finite automata with ε-moves* (NFA-ε), which are NFAs with the addition of the empty string ε, are even closer to LTSs.

3.1.1 Transitions, Executions, Traces and Reachable States

Given an LTS $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, the notation $s \xrightarrow{\alpha} s'$ is used as a shortcut for the (strong) *transition* $(s, \alpha, s') \in \rightarrow$. The state s is called the *source* (state) of the transition and the state s' the *destination* (state). A transition whose source and destination are the same is called a *loop*.

An *execution* is a sequence of transitions $s_0 \xrightarrow{\alpha_1} s_1 \cdots s_{n-1} \xrightarrow{\alpha_n} s_n$ with $\alpha_i \in (\mathcal{L} \cup \{\tau\})$. The *length* of the execution is n which corresponds to the number of transitions in the execution. An execution corresponds to a path in the graph of the LTS, and the two words *execution* and *path* are used interchangeably in this work. The execution can be written as $s_0 \xrightarrow{\sigma} s_n$ with $\sigma = \alpha_1 \cdots \alpha_n$. The notation $s \xRightarrow{\alpha} t$ is used as a shortcut for the *weak transition* $s \xrightarrow{\tau^* \alpha \tau^*} s'$ where τ^* is a possibly empty sequence of τ. The notation $s \xrightarrow{\alpha}$ (resp. $s \xRightarrow{\alpha}$) is a shortcut for the existence of a state $s' \in S$ such that $s \xrightarrow{\alpha} s'$ (resp. $s \xRightarrow{\alpha} s'$).

A *trace* of $\mathcal{M}$ is a sequence $\sigma = \langle \alpha_1 \ldots \alpha_n \rangle$ with $\alpha_i \in \mathcal{L}$, such that there exists an execution $s_0 \xRightarrow{\alpha_1} s_1 \cdots s_{n-1} \xRightarrow{\alpha_n} s_n$ in $\mathcal{M}$. The set of traces of $\mathcal{M}$ represents all the sequences of labels for which there exists an execution in the LTS. The *empty trace* is denoted ε.

Definition 3.2 (Trace). *Given an LTS $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, a sequence $\sigma = \langle \alpha_1 \cdots \alpha_n \rangle \in \mathcal{L}^*$ is a* trace *of $\mathcal{M}$ if and only if there exists an execution $s_0 \xRightarrow{\alpha_1} s_1 \cdots s_{n-1} \xRightarrow{\alpha_n} s_n$. The* set of traces *of $\mathcal{M}$ is defined as $\mathbf{Tr}(\mathcal{M}) = \{\sigma \in \mathcal{L}^* \mid s_0 \xRightarrow{\sigma}\}$. The empty trace is denoted ε.*

A state s' is *reachable* from another state s if there exists an execution between the two states. The *reachable states* of $\mathcal{M}$ is the set of states that are reachable from the initial state. It is denoted $reach(\mathcal{M})$ and defined as the union of the sets of states that are reachable from the initial state, for any traces of $\mathcal{M}$.

Definition 3.3 (Reachable states). *Given an LTS $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, the set of states that are* reachable *from a given state $s \in S$ with a trace $\sigma \in \mathcal{L}^*$ is denoted and defined as $s\,\mathbf{after}\,\sigma = \{s' \in S \mid s \overset{\sigma}{\Longrightarrow} s'\}$. The reachable states of $\mathcal{M}$ is denoted and defined as:*

$$reach(\mathcal{M}) = \bigcup_{\sigma \in \mathbf{Tr}(\mathcal{M})} s_0\,\mathbf{after}\,\sigma$$

An execution of the LTS example of Figure 3.1 is $\mathbf{A} \overset{b}{\longrightarrow} \mathbf{D} \overset{\tau}{\longrightarrow} \mathbf{B}$. The trace corresponding to this execution is $\sigma = \langle b \rangle$. The set of states that can be reached after executing this trace from the initial state is $\mathbf{A}\,\mathbf{after}\,b = \{\mathbf{B}, \mathbf{D}\}$. Those two states correspond to the weak transitions $\mathbf{A} \overset{b}{\Longrightarrow} \mathbf{D}$ and $\mathbf{A} \overset{b}{\Longrightarrow} \mathbf{B}$. The set of traces of $\mathcal{M}$ is $\mathbf{Tr}(\mathcal{M}) = \{\varepsilon, a, b, ac, bb, bc\}$. The reachable states are $reach(\mathcal{M}) = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$ and there is one unreachable state which is $\mathbf{E}$.

The set of traces may not be finite, which is for example the case when there are loops with visible actions, or cycles in the LTS. Figure 3.2 shows an example of an LTS with one visible loop and one cycle between three states. There is an infinity of traces among which all the sequences of a least one "a" label: $a, aa, aaa, aaaa, \ldots$ but also any sequence made of repetitions of the "abc" sequence: $abc, abcabc, abcabcabc, \ldots$. The set of traces of the example is formed of the trace satisfying the $(\mathsf{aa^*bc})^*$ regular expression.

3.1.2 Enabled and Possible Actions

Since LTSs are used to model reactive systems, labels on the transitions are usually referred to as actions. A useful information for an operator using a reactive system is to know what actions he can execute given the current state of the system.

Given an LTS $\mathcal{M}$, the set $\Gamma(s)$ of *enabled actions* of a state s is the set of actions that are directly executable from s, that is, actions corresponding to (strong) visible transitions outgoing from s.
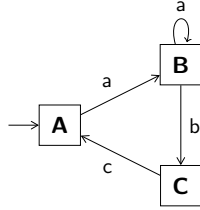
Figure 3.2. Example of an LTS with a visible loop on state **B** and with a cycle between states **A**, **B** and **C**. The set of traces of this LTS is infinite.

Definition 3.4 (Enabled action)**.** *Given an LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, *the set of* enabled actions *of a state* $s \in S$ *is denoted and defined as* $\Gamma(s) = \{\alpha \in \mathcal{L} \mid s \xrightarrow{\alpha}\}$.

Due to τ-transitions, there may be situations where some actions are not directly executable, but will become available only after some internal transitions. The execution of those internal actions may take some time that the operator has to wait for. Such an execution has an associated empty trace.

The set $A(s)$ of *possible actions* of a state s of an LTS contains the actions that are directly executable from any state that is reachable from s with the empty trace, including s itself. A direct consequence of this definition is that the set of enabled actions is a subset of the set of possible actions $\Gamma(s) \subseteq A(s)$.

Definition 3.5 (Possible action)**.** *Given an LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, *the set of* possible actions *of a state* $s \in S$ *is denoted and defined as* $A(s) = \{\alpha \in \mathcal{L} \mid s \xRightarrow{\alpha}\}$.

Looking back to the LTS example of Figure 3.1 on page 58, the initial state A has two enabled actions $\Gamma(\mathbf{A}) = \{a, b\}$, but three possible actions $A(\mathbf{A}) = \{a, b, c\}$.

3.1.3 Exploration

LTSs are typically used to represent all the possible behaviours of a given system. In order to enumerate those behaviours, it is useful to explore LTSs. Algorithm 1 presents a generic exploration algorithm for LTSs. There are many ways to explore an LTS, that is, to visit

each of its reachable states exactly once. The *removeElem* and *addElem* functions that are used in the generic algorithm, as well as the kind of structure that is used for L, define the exploration strategy. The two most common exploration algorithms are *breadth-first search* (BFS) and *depth-first search* (DFS) [CLRS09]. The BFS exploration is obtained by using a (FIFO) queue for L and the DFS exploration uses a (LIFO) stack.

Algorithm 1: Generic LTS exploration algorithm.

> **Input**: $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, an LTS
> **Output**: –
> **Side Effect**: The function *doSomething* has been applied to all the reachable
> $\qquad\qquad$ states of $\mathcal{M}$, exactly once.
>
> **foreach** $s \in S$ **do**
> $\quad \llcorner$ **mark** s **as not** *visited*
>
> $L \leftarrow [s_0]$
> **while not** *isEmpty* (L) **do**
> $\quad \mid \quad s \leftarrow removeElem\ (L)$
> $\quad \mid \quad doSomething\ (s)$
> $\quad \mid \quad$ **mark** s **as** *visited*
> $\quad \mid$
> $\quad \mid \quad$ **foreach** $(s, \alpha, s') \in \rightarrow$ **do**
> $\quad \mid \quad \mid \quad$ **if not** s' **is** *visited* **then**
> $\quad \mid \quad \mid \quad \llcorner\ addElem\ (L, s')$

In addition to visiting exactly once each reachable state of the LTS, a function *doSomething* is applied to each visited state. That function can for example be used to check a property on all the states of a LTS. The algorithm can also be modified to return a value, for example a boolean indicating whether the property is satisfied for all the states of the LTS or not. If the property is not satisfied for a given state s, an execution starting at the initial state and reaching the state s can be provided. That execution is called an *error trace* or a *counterexample*. Such a verification of a property, with a counterexample if the property is not satisfied, is precisely the essence of model-checking [CGP99] as already presented in more detail in Section 2.3.1.

Both BFS and DFS exploration algorithms have the same time complexity which is $\mathcal{O}(n + m)$ where n is the number of reachable states and m the number of transitions between those states. The space

complexity is also the same and is $\mathcal{O}(n + m)$. One advantage of BFS is that it gives a minimal-length counterexample when used for model-checking and one advantage of DFS is that it requires less additional memory in practice than BFS does, since only the states of the current explored path are memorised in L during the exploration.

3.1.4 Internal Actions

The *internal actions* correspond to actions that are invisible and uncontrollable from an operator's point of view. Those actions occur inside the system without any trigger from the operator who cannot see them anyway. When LTSs are used to model systems, internal actions indeed correspond to concrete actions inside the system, but since they are not distinguishable by the operator, they are all represented with the same label τ.

Internal actions in an LTS can produce some particular behaviours of the system, when seen by the operator. Since internal actions correspond to concrete actions taking place in the system, the system may appear as unresponsive to the user for some time. Also, the behaviour of the system may completely change without any visible feedback to the user.

Such issues caused by the presence of internal actions is a major concern for the analysis of human-machine interactions and occupies a non-negligible part of this thesis. Those issues are introduced further in this chapter and are the focus of the next chapter.

3.1.5 Determinism

For a given trace of an LTS, it may be the case that several different executions exist in the LTS for the trace. Such a situation can occur if multiple transitions with the same visible action are going from the same state to different states. The situation can also occur when there are internal actions in the LTS. Figure 3.3 shows two LTS examples where the trace $\langle a \rangle$ corresponds to multiple executions.

In general, an LTS is said to be *deterministic* if and only if for any state of the LTS, and for every action of the alphabet, there is at most one state that can be weakly reached. In other words, executing any trace of the LTS from the initial state always lead to the same state, no
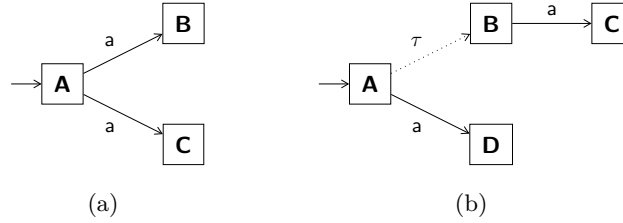
Figure 3.3. Examples of LTS for which there exists multiple execution for the trace $\sigma = a$: (a) the two executions $\mathbf{A} \xrightarrow{a} \mathbf{B}$ and $\mathbf{A} \xrightarrow{a} \mathbf{C}$ and (b) the two executions $\mathbf{A} \xrightarrow{\tau} \mathbf{B} \xrightarrow{a} \mathbf{C}$ and $\mathbf{A} \xrightarrow{a} \mathbf{B}$.

matter what is the underlying execution. In particular, this definition implies that a deterministic LTS does not contain any τ-transition, except possibly τ-loops. Deterministic LTSs are also characterised by the fact that their states have the same set of enabled and possible actions, that is, $\Gamma(s) = A(s)$.

Definition 3.6 (Determinism). *An LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$ *is deterministic if and only if* $\forall s \in S : \forall \alpha \in \mathcal{L} : |s \, \mathbf{after} \, \alpha| \leq 1$.

The LTS example from Figure 3.1 on page 58 is definitely not deterministic as it contains a τ-transition which is not a loop. More precisely, the set of reachable states from the initial state A does contain more than one element for the action b, since $\mathbf{A} \, \mathbf{after} \, b = \{\mathbf{B}, \mathbf{D}\}$.

Divergence

Divergence is another side effect that is introduced with τ-transitions, when infinite executions are considered. A *divergent execution* is an execution that consists of τ-transitions only. An LTS is *divergent* if there exists one execution that contains a divergent execution.

Divergence can be an issue for reactive systems. A divergent execution can represent the fact that the system is busy executing internal invisible actions which may make it non-responsive to external solicitation. A divergent LTS represents a reactive system which may appear as deadlocked from the external point of view of the operator. Divergences are not treated in a special way in this work.

Definition 3.7 (Divergence). *Given an LTS $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, a state $s \in S$ is said to be* divergent *if and only if $s \xrightarrow{\tau^\omega}$. The LTS $\mathcal{M}$ is said to be* divergent *is there exists a divergent state $s \in reach(\mathcal{M})$.*

The LTS example of Figure 3.1 on page 58 is definitely divergent since there are some divergent executions in it, for example **D** $\xrightarrow{\tau}$ **B** $\xrightarrow{\tau}$ **B** $\xrightarrow{\tau}$ **B** $\xrightarrow{\tau}$ . . . .

Determinisation

An non-deterministic LTS can always be determinised, so as to preserve the traces, with the *subset construction* that goes back to Rabin and Scott [RS59b]. That construction builds a new LTS that keeps the same traces, but with the non-determinism removed. The idea of the algorithm is to solve the non-determinism by grouping states that can be reached with the same trace, those groups forming the state of the determinised LTS. There is a transition $S \xrightarrow{\alpha} S'$ between two states of the determinised LTS if and only if there exists a transition with the α action between a state belonging to the set S to a state from the set S'. Formally, it means that $S' = \{s' \mid \exists s \in S : s \xrightarrow{\alpha} s'\}$.

Algorithm 2 shows the Rabin-Scott subset construction algorithm adapted to LTS [Sch04]. In the worst case, the algorithm will run in exponential time, that is, with a $\mathcal{O}(2^n)$ time complexity where n is the number of states of the LTS. In the worst case, the set of states of the determinised LTS is indeed the power set of the set of states of the LTS. The determinised LTS is denoted $det(\mathcal{M})$ and is also an LTS. The proposed algorithm also gets rid of any τ-transitions, even τ-loops which may have been kept.

Figure 3.4 shows the determinisation by the subset construction algorithm of the LTS example of Figure 3.1 from page 58. The non-determinism that was originally present due to the τ-transition between states **D** and **B** is solved with the transition $\{\textbf{A}\} \xrightarrow{b} \{\textbf{B}, \textbf{D}\}$ in the determinised LTS.

As previously stated, one interesting property is that determinisation preserves the traces, that is, $\textbf{Tr}(\mathcal{M}) = \textbf{Tr}(det(\mathcal{M}))$. Moreover Algorithm 2 builds deterministic LTSs free of any τ-transitions (and thus also not divergent).

Algorithm 2: LTS determinisation.

Input: $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, an LTS
Output: $det(\mathcal{M}) = \langle S_D, \mathcal{L}, s_{0_D}, \rightarrow_D \rangle$, the determinisation of $\mathcal{M}$

$s_{0_D} \leftarrow s_0 \textbf{ after } \varepsilon$
$L \leftarrow \{s_{0_D}\}$
while not $isEmpty\ (L)$ **do**
$\quad$ $s_D \leftarrow removeElem\ (L)$
$\quad$ $S_D \leftarrow S_D \cup \{s_D\}$
$\quad$ **foreach** $\alpha \in A(s_D)$ **do**
$\quad\quad$ $s'_D \leftarrow \bigcup_{s \in s_D} s \textbf{ after } \alpha$
$\quad\quad$ $\rightarrow_D \ \leftarrow \{(s_D, \alpha, s'_D)\} \cup \rightarrow_D$
$\quad\quad$ **if not** $s'_D \in S_D$ **then**
$\quad\quad\quad$ $addElem\ (L, s'_D)$

return $\langle S_D, \mathcal{L}, s_{0_D}, \rightarrow_D \rangle$
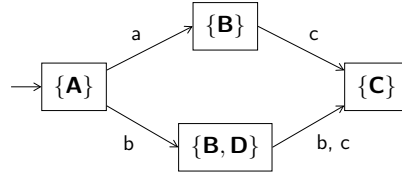
Figure 3.4. Determinisation of the LTS example of Figure 3.1 that has been computed by the subset construction algorithm.

Operational Determinism

The notion of determinism defined just above is also referred to as *structural determinism*, that is, any given sequence of visible actions belonging to the set of traces leads to a unique state, starting from the initial state. Such a definition is quite strong and could be too restrictive. The notion of *operational determinism* [HV06] is more flexible. It allows some non-determinism provided that different executions with the same trace lead to states with equivalent behaviours. Intuitively, if several states can be reached with the same trace, they have to have the same sets of possible actions.

Definition 3.8 (Operational Determinism). *An LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$ *is* operationally deterministic *if and only if* $\forall s_1, s_2 \in S : s_1 \overset{\sigma}{\Longrightarrow} s'_1$ *and* $s_2 \overset{\sigma}{\Longrightarrow} s'_2 : A(s'_1) = A(s'_2)$ *and* $s'_1 \overset{\tau^\omega}{\longrightarrow}$ *implies* $s'_2 \overset{\tau^\omega}{\longrightarrow}$.

Figure 3.5 illustrates the notion of operational determinism. None of the three LTSs are (structurally) deterministic. Only the first one of Figure 3.5(a) is operationally deterministic. The LTS of Figure 3.5(b) is not operationally deterministic since the states **B** and **C** reached with the same trace have different sets of possible actions. Finally, the third example of Figure 3.5(c) illustrates a case where two states reached with the same traces do not have the same behaviour according to divergence.

(a)   (b) $A(B) \neq A(C)$.   (c) There are divergent executions starting from B and not from C.
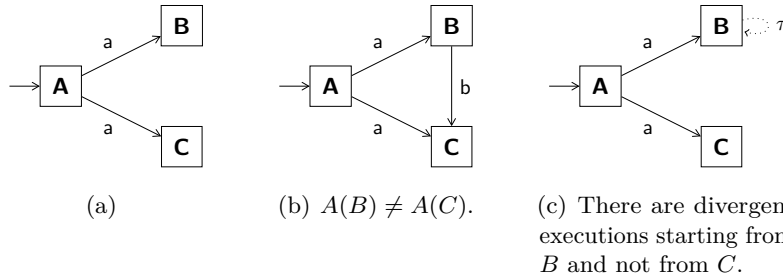
Figure 3.5. Examples of LTS to illustrate operational determinism. The three examples are characterised by the fact that $s_0 \, \mathbf{after} \, a = \{\mathbf{B}, \mathbf{C}\}$. The first one is operationally deterministic and the last two ones are not.

As previously stated, determinism issues play a crucial role for the analysis of human-machine interactions. All those issues are discussed thoroughly in the two next chapters (which present the main contributions of this thesis).

3.1.6 Synchronous Parallel Composition

Two LTSs can be composed together with synchronisation on the actions that are common between their respective alphabets. The *synchronous parallel composition* between two LTSs $\mathcal{M}$ and $\mathcal{M}'$ is an LTS whose states are pairs of states from both LTSs. The composition represents the parallel execution of both LTSs. For an action α that is common to both alphabets, the composition can proceed if and only if there is a transition with α in both LTSs. For the other actions and for τ-transitions, each LTS can move independently. Synchronous parallel composition represents parallelism with interleaving.

Definition 3.9 (LTS synchronous parallel composition)**.** *Given two LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$ *and* $\mathcal{M}' = \langle S', \mathcal{L}', s_0', \rightarrow' \rangle$, *the* synchronous parallel composition *between both LTSs, denoted* $\mathcal{M} \parallel \mathcal{M}'$, *is an LTS* $\mathcal{C} = \langle S_C, \mathcal{L}_C, s_{0_C}, \rightarrow_C \rangle$ *where* $S_C \subseteq (S \times S')$, $\mathcal{L}_C = \mathcal{L} \cup \mathcal{L}'$, $s_{0_C} = (s_0, s_0')$ *and* $\rightarrow_C \subseteq S_C \times (\mathcal{L}_C \cup \{\tau\}) \times S_C$ *is defined as the smallest set such that:*

- *if* $\alpha \in \mathcal{L} \cap \mathcal{L}' : (s, t) \xrightarrow{\alpha} (s', t')$ *if* $s \xrightarrow{\alpha} s'$ *and* $t \xrightarrow{\alpha} t'$;

- *otherwise,* $\alpha \in (\mathcal{L}_C \cup \{\tau\}) \setminus (\mathcal{L} \cap \mathcal{L}')$ *and:*

    - $(s, t) \xrightarrow{\alpha} (s', t)$ *if* $s \xrightarrow{\alpha} s'$;
    - *and* $(s, t) \xrightarrow{\alpha} (s, t')$ *if* $t \xrightarrow{\alpha} t'$.

Figure 3.6 shows an example of the parallel synchronous composition between two LTSs. The common alphabet between the two LTSs is $\{b, c\}$ which means that those actions must be enabled in the two LTSs for the composition to proceed.

3.2 Human-Machine Interaction LTS

This work focuses on the dynamic aspects of interaction, at the low level of actions executed on a system by an operator. Those actions correspond to events that can be classified according to different criteria. Actions can be *visible*, meaning that the operator is able to see them, or *invisible*. Visible actions can further be classified as *controllable* if they are initiated by the operator, or *uncontrollable* if they are triggered by the system. This section defines an enriched LTS to take those considerations about actions into account.

Figure 3.7 shows the three possibilities for the actions, according to the visibility and controllability criteria. The set of actions is thus partitioned into three sets:

- The *commands* correspond to visible and controllable actions that the operator performs on the system;

- The *observations* represent visible but uncontrollable actions that cannot be controlled without any initiative from the operator;

- and finally the *internal actions* occur inside the system and are completely invisible to the operator.
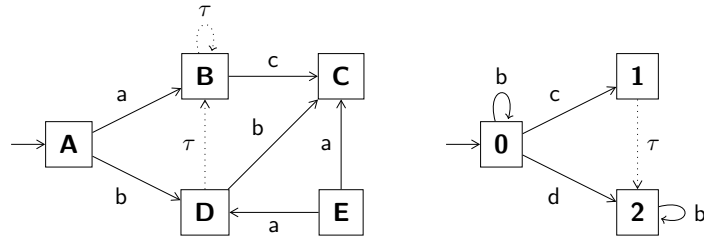
(a) An LTS $\mathcal{M}$, with an alphabet $\mathcal{L} = \{a, b, c\}$.

(b) An LTS $\mathcal{M}'$, with an alphabet $\mathcal{L} = \{b, c, d\}$.
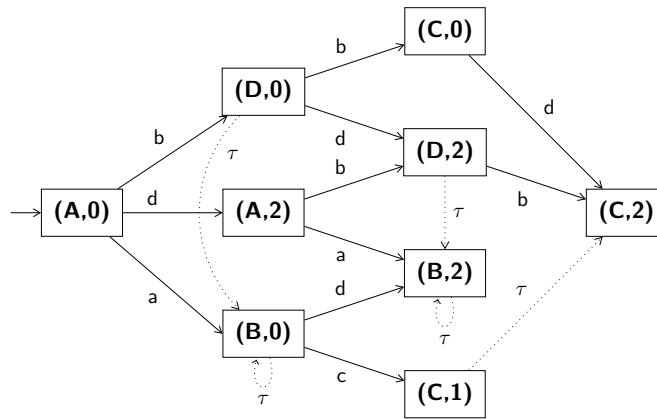
(c) The synchronous parallel composition $\mathcal{M} \parallel \mathcal{M}'$.

Figure 3.6. Example of the synchronous parallel composition between two LTSs.

The distinction between commands and observations, which are also respectively referred to as inputs and outputs with respect to the system, plays a crucial role when studying human-machine interaction [Jav02, HD07] as argued in Section 2.4.6. In order to take that distinction into account, *human-machine interaction labelled transition systems* (HMI-LTS) are an enriched variant of usual LTSs where the set of labels is partitioned into two sets $\mathcal{L}^c$ and $\mathcal{L}^o$ respectively representing commands and observations. Internal actions are represented with τ. Visible actions are the labels of the underlying LTS.
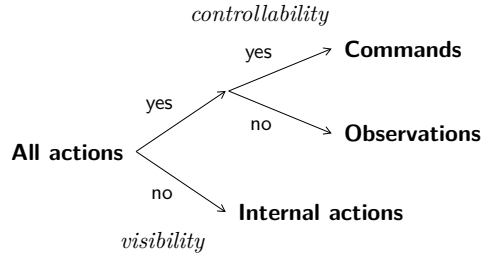
Figure 3.7. Classification of actions for human-machine interaction LTS into three sets according to the visibility and the controllability criteria.

Definition 3.10 (Human-Machine Interaction Labelled Transition System)**.** *A human-machine interaction labelled transition system (HMI-LTS) is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$ where $\langle S, \mathcal{L}^c \cup \mathcal{L}^o, s_0, \rightarrow \rangle$ is a labelled transition system, $\mathcal{L}^c$ is a finite set of command labels and $\mathcal{L}^o$ is a finite set of observation labels. The two sets $\mathcal{L}^c$ and $\mathcal{L}^o$ are disjoint and the set of visible actions is $\mathcal{L}^c \cup \mathcal{L}^o = \mathcal{L}$.*

Since an HMI-LTS is just an extension of an LTS, all the definitions from the previous section still apply for HMI-LTS. Figure 3.8 shows the graphical representation of an HMI-LTS example. Transitions labelled with commands are represented with plain lines and transitions labelled with observations with dashed lines. As for LTSs, internal transitions are represented with dotted lines.
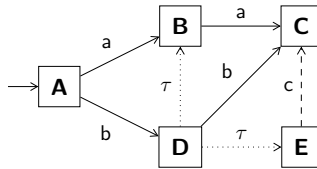
Figure 3.8. Graphical representation of an HMI-LTS example with five states and seven transitions. The initial state is the state A. The alphabet is $\mathcal{L}^{co} = \{a, b, c\}$ and is partitioned into commands a and b (plain lines) and observation c (dashed line).

The HMI-LTS example of Figure 3.8 is formally defined as the tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$ where:

- $S = \{A, B, C, D, E\}$;

- $\mathcal{L}^c = \{a, b\}$;
- $\mathcal{L}^o = \{c\}$;
- $s_0 = A$;
- and $\rightarrow = \{(A, a, B), (A, b, D), (B, a, C), (D, \tau, B),$
$$(D, b, C), (D, \tau, E), (E, c, C)\}.$$

In addition to the sets of enabled and possible actions defined in the previous section (Definitions 3.4 and 3.5 on page 61), it is sometimes interesting to consider the sets of enabled or possible commands or observations. The definitions of those sets are similar to those for LTS. They are denoted Γ^c and A^c for commands and Γ^o and A^o for observations. On the HMI-LTS example of Figure 3.8, $\Gamma^c(\mathbf{D}) = \{b\}$, $\Gamma^o(\mathbf{D}) = \emptyset$, $A^c(\mathbf{D}) = \{a, b\}$ and $A^o(\mathbf{D}) = \{c\}$.

Definition 3.11 (Enabled and possible sets of commands and observations). *Given an HMI-LTS $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$, the set of enabled commands (resp. observations) of a state $s \in S$ is $\Gamma^c(s) = \Gamma(s) \cap \mathcal{L}^c$ (resp. $\Gamma^o(s) = \Gamma(s) \cap \mathcal{L}^o$) and the set of possible commands (resp. observations) of a state $s \in S$ is $A^c(s) = A(s) \cap \mathcal{L}^c$ (resp. $A^o(s) = A(s) \cap \mathcal{L}^o$).*

Figure 3.9 shows an HMI-LTS example used to model a simple vending machine that repeatedly serves customers. To use the machine whose model is on the left, the customer has first to introduce a coin into the machine. Then, the machine will check whether there is coffee remaining, in which case it serves a cup to the customer. After a cleanup, the machine is ready for the next customer. It the machine has no more coffee, it does not accept coins any more and is closed. The HMI-LTS on the right represents the mental model of one customer who believes that the machine is only able to serve one coffee.

3.2.1 Interaction Model

The focus of this work is the analysis of the interactions between an operator and a machine. In order to perform such an analysis, all the possible interactions have to be considered. The synchronous parallel composition presented in Definition 3.9 on page 67 exactly contains all the possible interactions between two LTSs.

(a) System model.   (b) Mental model.
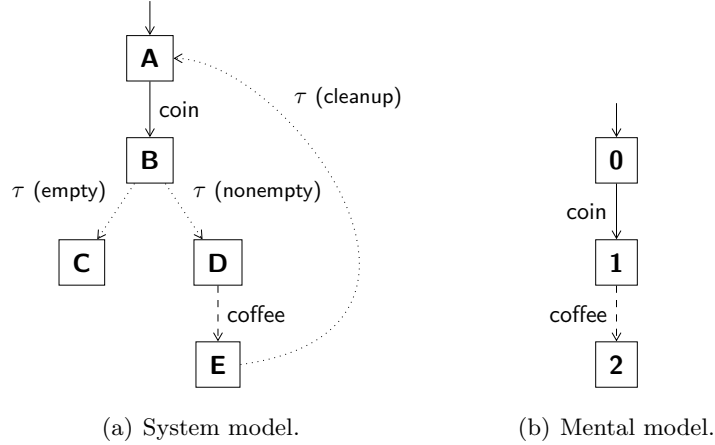
Figure 3.9. Example of a vending machine system modelled with an HMI-LTS. The system is on the left and a mental model representing a user that thinks that the machine only delivers one coffee is on the right. The **coin** command corresponds to the customer inserting money in the machine and the **coffee** observation corresponds to the machine delivering coffee. The τ's internal actions **empty** and **nonempty** correspond to the check of the stock level of coffee and the τ internal action **cleanup** corresponds to the machine reseting to the initial state after a clean-up.

When considering HMI analyses, both the system and the operator are modelled with HMI-LTSs. The letters $\mathcal{S}$ and $\mathcal{H}$ are used from now on to represent respectively the system and the human. In the frame of the work presented in this thesis, some hypotheses have been made about the system and mental models. First of all, the system and the mental models are sharing exactly the same alphabet. Secondly, whereas the system model may contain internal actions, the mental model does not contain any internal actions and is also supposed to be deterministic.

Generally speaking, a mental model may be non-deterministic. It would mean that whenever the operator is performing an action, several possible resulting states may be reached as a response to the action. In this work, it is a choice to only consider deterministic mental models. That is of course a modelling choice of how humans are thinking about systems whenever they use them, meaning that they always know for sure the effect of any performed action.

At any time during the interaction, the system is in a state s_S of the system model and the operator is in a state s_H of his mental model.

From a *composite state* (s_S, s_H), either a command can be performed by the user if it is possible on the system or an observation can be made by the user if he expects it according to his mental model. Internal actions may also occur in the system at any time. Figure 3.10 shows the interaction model for the vending machine example. The synchronous parallel composition $\mathcal{S} \parallel \mathcal{H}$ is referred to as the *interaction model*.
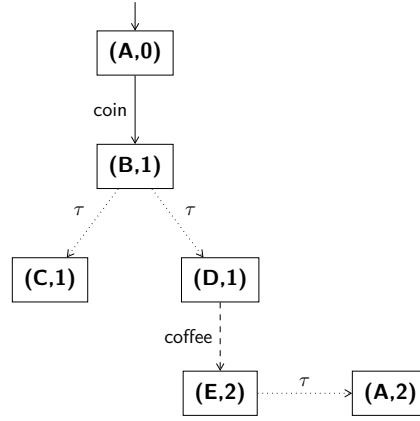
Figure 3.10. Interaction model for the vending machine example, between the system from Figure 3.9(a) and the mental model of Figure 3.9(b).

The set of traces of the synchronous parallel composition between two LTSs contains all the traces that are common to both composed LTSs. In terms of HMI, it means that the synchronous parallel composition operator computes the behaviour that is common to two LTSs.

Property 3.12. *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and a deterministic non-divergent HMI-LTS $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, $\mathbf{Tr}(\mathcal{S} \parallel \mathcal{H}) = \mathbf{Tr}(\mathcal{H}) \cap \mathbf{Tr}(\mathcal{S})$.*

Proof. Let $\sigma = \langle \alpha_1 \cdots \alpha_n \rangle \in \mathbf{Tr}(\mathcal{S} \parallel \mathcal{H})$, a trace of the synchronous parallel composition. By definition of trace, it means the existence of an execution $(s_{0_S}, s_{0_H}) \stackrel{\alpha_1}{\Longrightarrow} (s_{1_S}, s_{1_H}) \cdots (s_{n-1_S}, s_{n-1_H}) \stackrel{\alpha_n}{\Longrightarrow} (s_{n_S}, s_{n_H})$ in $\mathcal{S} \parallel \mathcal{H}$.

Let us consider the weak transition $(s_S, s_H) \stackrel{\alpha}{\Longrightarrow} (s'_S, s'_H)$ that exists if and only if there exists a sequence of transitions $(s_S, s_H) \xrightarrow{\tau^*} (t_S, s_H) \xrightarrow{\alpha} (t'_S, s'_H) \xrightarrow{\tau^*} (s'_S, s'_H)$, by definition of weak transition.

Moreover, by definition of the synchronous parallel composition, such a sequence exists if and only if the sequence $s_S \xrightarrow{\tau^*} t_S \xrightarrow{\alpha} t'_S \xrightarrow{\tau^*} s'_S$ exists in the system model, that is, $s_S \xRightarrow{\alpha} s'_S$, and the sequence $s_H \xrightarrow{\alpha} s'_H$ exists in the mental model.

To conclude, the trace $\sigma = \langle \alpha_1 \cdots \alpha_n \rangle$ is in $\mathbf{Tr}(\mathcal{S} \parallel_I \mathcal{H})$ if and only if $(s_{i_S}, s_{i_H}) \xRightarrow{\alpha_{i+1}} (s_{i+1_S}, s_{i+1_H})$ for all $i \in 0, \cdots, n-1$, which is satisfied if and only if $s_{i_S} \xRightarrow{\alpha_{i+1}} s_{i+1_S}$ and $s_{i_H} \xrightarrow{\alpha_{i+1}} s_{i+1_H}$ for all $i \in 0, \ldots, n-1$, that is, $\sigma \in \mathbf{Tr}(\mathcal{S})$ and $\sigma \in \mathbf{Tr}(\mathcal{H})$, by definition of traces of an HMI-LTS. $\square$

3.3 Enriched Models

Using an HMI-LTS to represent a system model may sometimes be not adequate with respect to how the designers think about a system. Just having blackbox states with all the information encoded as actions on transitions may not be convenient enough for the designers. The modelling approach used must be rich enough to be able to perform the desired analysis but it should also be kept as simple as possible to ease the analysis methods. HMI-LTS may be too simple and this section introduces an enriched version of HMI-LTS, adding information on the states in the style of *Kripke structures* [CGP99]. As it was observed in the related work presented in the previous chapter, most of the time, system designers think about systems as composed of a set of variables, each state of the system corresponding to an assignment of values to all those variables. Accordingly, the knowledge of the human must also be modelled so as to take into account the information that is added on states.

To illustrate the modelling approach used in this work, Figure 3.11 shows an example of a timer such as used when cooking. For that example, the counter of the timer takes its value over a range from 0 to N (here fixed to $N = 2$) and is initially set to 0. The user can press on a button (inc) to increase the value while setting up the machine. The counter cycles between 0 and 2. Whenever the counter is different from zero, the user can set it back to its initial value (reset) or can start the countdown (start). When the countdown is running, the user can observe the value of the counter which is decreasing through the screen.

When the value of the counter reaches zero, the machine goes back to the initial state while emitting a sound (ring). The user can cancel (cancel) the countdown while it is running and has not reached zero.
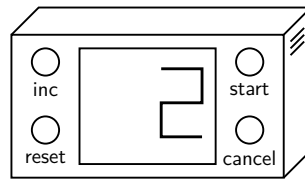
Figure 3.11. An example of a countdown machine that is entered an integer value between 0 and 2 (inclusive) and that will countdown until it reaches zero.

3.3.1 Enriched System Model

For the countdown example, it is clear that there is one variable which is a counter c that can take three different values : 0, 1 and 2. Figure 3.12 shows a graphical representation of a system model for the countdown machine. It is essentially an HMI-LTS whose states have been labelled as c_0, c_1 or c_2 depending on the value of the counter. The model has six states: the states A, B and C correspond to the configuration of the countdown machine and the states D, E and F correspond to the machine which is running.
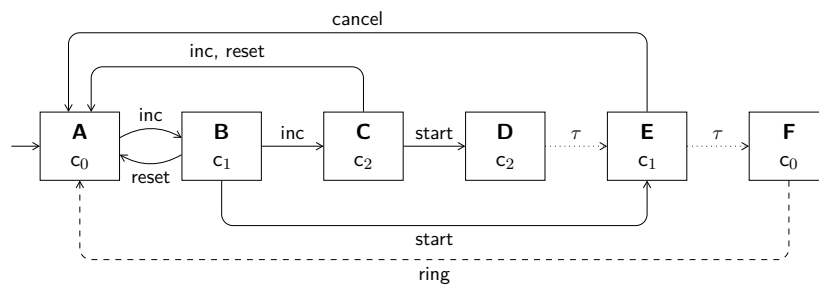
Figure 3.12. An example of a system model representing a countdown machine.

In order to represent observable information on states, *HMI state-valued system model* (HVS) are enriched HMI-LTSs with the addition of

a set of *state-values* and with a mapping function associating each state to one state-value, just like the valuation used for Kripke structures.

Definition 3.13 (HMI state-Valued System model). *A human-machine interaction state-valued system model (HVS) is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O} \rangle$ where $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$ is an HMI-LTS, $\mathcal{L}^v$ is a finite set of state-values and $\mathcal{O} : S \mapsto \mathcal{L}^v$ is a state-value mapping function. The three sets $\mathcal{L}^c$, $\mathcal{L}^o$ and $\mathcal{L}^v$ are disjoint.*

The information that is observable on the states of the system is modelled with a set of state-values $\mathcal{L}^v$ and a function $\mathcal{O}$ which gives for each state the observation that the operator can make when the system is in that state. This approach is more general than just setting information on states as a set of variables with their values, as detailed at the end of this section. When interacting with the system, the operator can, at any time, look at the state-value corresponding to the current state of the system.

The countdown example HVS shown on Figure 3.12 is formally defined as the tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O} \rangle$ with:

- $S = \{A, B, C, D, E, F\}$
- $\mathcal{L}^c = \{cancel, inc, reset, start\}$
- $\mathcal{L}^o = \{ring\}$
- $s_0 = A$
- $\rightarrow = \{(A, inc, B), (B, inc, C), (B, reset, A), (B, start, E),$
  $(C, inc, A), (C, reset, A), (C, start, D), (D, \tau, E),$
  $(E, cancel, A), (E, \tau, F), (F, ring, A)\}$
- $\mathcal{L}^v = \{c_0, c_1, c_2\}$
- $\mathcal{O}(s) = \begin{cases} c_0 & , \text{if } s \in \{A, F\} \\ c_1 & , \text{if } s \in \{B, E\} \\ c_2 & , \text{if } s \in \{C, D\} \end{cases}$

HMI-LTS is in fact a particular case of HVS where the set of state-values has only one element that labels all the states of the model. The consequence is that all the definitions that applied to LTS and HMI-LTS can also be applied to the underlying LTS and HMI-LTS of the HVS.

State-values

The enriched model has now two kinds of observation that can be done by the operator. The operator can both observe *state-values* on the states and make observations on visible transitions. A legitimate question that can be asked is whether the two kinds of observations are necessary and why they can be in fact complementary.

The difference between the two kinds of observation is in the interpretation according to a human-machine interaction point of view. The observations that can be made on the state may be ignored by the operator while interacting with the system. In contrary, observations occurring on a transition are output by the system and seen by the operator. If the operators do not take into account those observations in their models, the interaction will not proceed according to the chosen definition. Of course, such a modelling choice does not consider the case where the operator may get distracted and miss the observation. This latter point is discussed in Chapter 6. Moreover, observations on a transition also bring another message to the operator, they indicate clearly a change of state in the system. This choice of having two kinds of observation has consequences for the definition of good interaction and for the analyses that are performed, as explained in the next chapter.

State-variables

As introduced above, a typical way to represent information about the state of a system is by means of *state-variables*. A system is characterized by a set of variables and, in any state of the system, it is possible to observe the values of some subset of the state-variables, exactly those which are visible. For example, ADEPT models [Fea10] and models based on interactors [CH11] are using variables to model the state-values.

This can be modelled within the HVS structure by considering that the set of state-values is composed of all the valuations for the *visible state-variables*. If the states of the system are characterised by n state-variables $x_1, \cdots, x_n$ whose values are respectively ranging over the domain $D_1, \cdots, D_n$, if the visible state-variables are $x_1, \cdots, x_k$ ($k \leq n$), and if $S = D_1 \times \cdots \times D_n$, then the set of state-values is defined as $\mathcal{L}^v = D_1 \times \cdots \times D_k$ and the observation function is defined as $\mathcal{O}(\langle v_1, \cdots, v_n \rangle) = \langle v_1, \cdots, v_k \rangle$.

To illustrate that, Figure 3.13 shows an alternative model for the countdown example where state-values are modelled with state-variables. The system is characterised by three state-variables: the integer state-variable c represents the value of the countdown machine ($c \in \{0, 1, 2\}$), the boolean state-variable a is used to represent whether the alarm is ringing or not ($a \in \{T, F\}$), and the boolean state-variable r is used to represent whether the countdown machine is running or not ($r \in \{T, F\}$). Only the two first state-variables are visible and thus, the set of state-values is $\mathcal{L}^v = \{\langle 0, F \rangle, \langle 1, F \rangle, \langle 2, F \rangle, \langle 0, T \rangle, \langle 1, T \rangle, \langle 2, T \rangle\}$.
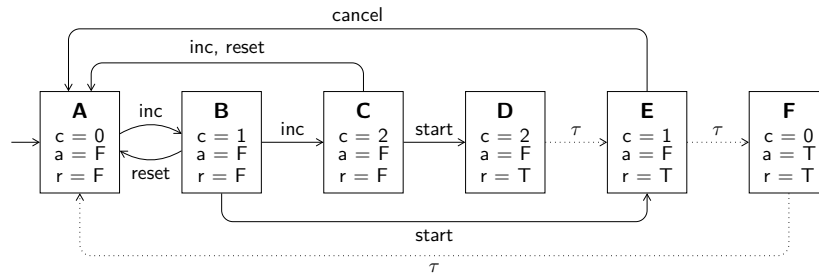
Figure 3.13. An alternative (enriched) system model for the countdown machine example where the ring observation has been replaced by the r state-variable.

The modelling choice has consequences on the behaviour that is captured, according to a human-machine interaction point of view. In the original model of Figure 3.12, if the operator misses the ring observation for any reason, it is not possible for him to know whether the system is still in the **F** state or if it has already transitioned into the **A** state. With the alternative model, the operator can use the state-value to distinguish among the two states **F** and **A**. State-values is in a way a more stable information than observation on transitions, since they are always available as long as the system is not changing state. The user cannot miss the state-values, except if he has to track their change in which case τ-transitions may make him loose some state-values. Given a weak transition $s \overset{a}{\Longrightarrow} s'$, all the states just before and after the strong transition labelled with a are considered as *unstable* with respect to state-values since inattention from the operator can make the tracking of the state-values changes impossible. Figure 3.14 illustrates those unstable states. Greyed states and all the states following them before white states are unstable states, assuming that the s' state has no outgoing τ-transition.
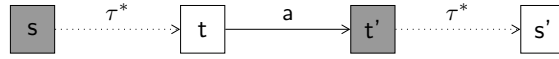
Figure 3.14. Unstable states with respect to state-values are introduced with τ-transitions which makes it impossible for the operator to track state-values changes.

3.3.2 Enriched Mental Model

The mental model describing the behaviour of the system from the point of view of the operator may also take into account state information. The operator can interact with the system in two ways: he can perform commands or perceive information, that is, observe the system. Observations that can be done are either event-based (observations on transitions) or state-based (state-values on states).

State-values are taken into account in the human model by *action guards*, that is, conditions on the state-value that must be verified in the current state of the system. In order to represent mental models, *HMI state-valued mental model* (HVM) are enriched HMI-LTSs with the addition of state-values on the transitions. Moreover, as already stated, mental models are considered deterministic and free of τ-transitions in the frame of this work.

Definition 3.14 (HMI state-Valued Mental model)**.** *A human-machine interaction state-valued mental model (HVM) is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v \rangle$ where $\mathcal{L}^v$ is a finite set of state-values, $\rightarrow \subseteq S \times \mathcal{L}^v \times \mathcal{L} \times S$ and $\langle S, \mathcal{L}^v \times \mathcal{L}^c, \mathcal{L}^v \times \mathcal{L}^o, s_0, \rightarrow \rangle$ is a deterministic HMI-LTS without τ-transition. The three sets $\mathcal{L}^c$, $\mathcal{L}^o$ and $\mathcal{L}^v$ are disjoint.*

Figure 3.15 shows a graphical representation of an HVM for the countdown machine. The model has two states respectively corresponding to the configuration of the countdown machine and to the machine which is running. State-values are indicated between brackets, that is, the notation $s \xrightarrow{[v]\alpha} t$ denotes $(s, v, \alpha, t) \in \rightarrow$. That HVM is formally defined as the tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v \rangle$ with:

- $S = \{S_0, S_1\}$
- $\mathcal{L}^c = \{cancel, inc, reset, start\}$

- $\mathcal{L}^o = \{ring\}$

- $s_0 = S_0$

- $\rightarrow = \{(S_0, c_0, inc, S_0), (S_0, c_1, inc, S_0), (S_0, c_2, inc, S_0),$
  $(S_0, c_1, start, S_1), (S_0, c_2, start, S_1), (S_0, c_1, reset, S_0),$
  $(S_0, c_2, reset, S_0), (S_1, c_1, cancel, S_0), (S_1, c_0, ring, S_0),$
  $(S_1, c_1, ring, S_0), (S_1, c_2, ring, S_0)\}$

- $\mathcal{L}^v = \{c_0, c_1, c_2\}$
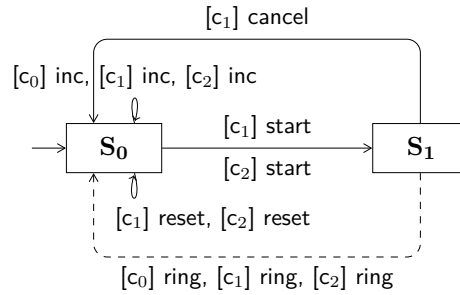
Figure 3.15. An example of an (enriched) mental model for the countdown machine.

As it is the case for HVS, HMI-LTS is also a particular case of HVM, where there is only one possible state-value. In such a situation, the (unique) state-value is satisfied in all system states. Again, all the properties that apply to LTS and HMI-LTS are also applicable to the underlying LTS and HMI-LTS of the HVM.

Action Guards

For HVM, only single state-values are used as guards on transitions. It is possible to define conditions on transitions with the more general notion of *action guards*. During an interaction, the visible action on the transition will only be taken into account by the operator if the state-value of the current state of the system satisfies the action guard. A transition $s \xrightarrow{[g]\alpha} s'$ is a shortcut for the set of transitions $\{s \xrightarrow{[v]\alpha} s' \mid v \models g\}$.

If the operator does not need to care about the state-value of the current state, any transition can be chosen so that the action guard is

simply defined as *true*. Figure 3.16 shows the correspondence for the transition shortcut, in that case.
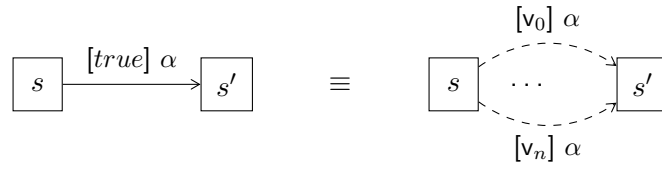
Figure 3.16. Correspondence for the transition shortcut used in HVM when the user action is not conditionned by an action guard.

Transitions of enriched mental models are considered to take place as an single atomic step. The operator checks that the action guard is satisfied and then immediately performs the action at the same time. Action guards are acting like preconditions on the execution of transitions.

For HVM, enabled and possible commands and observations sets are the same since HVM are free of τ-transitions. It is also possible to focus on the set of actions that are enabled and possible given a state-value, or by extension, an action guard. That information makes sense to the operator, as it is, for example, possible to read in a user manual an instruction like: *"If the LED is green, you can activate the pouring function."*.

Definition 3.15 (Enabled sets of commands and observations conditioned by a state-value)**.** *Given an HVM $\mathcal{H} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v \rangle$, the set of* enabled commands *(resp. observations) conditioned by the state-value $v \in \mathcal{L}^v$ of a state $s \in S$ is $\Gamma_v^c(s) = \{\alpha \in \mathcal{L}^c \mid s \xrightarrow{[v]\alpha} s'\}$ (resp. $\Gamma_v^o(s) = \{\alpha \in \mathcal{L}^o \mid s \xrightarrow{[v]\alpha} s'\}$).*

The definition can be easily extended to action guards by defining $\Gamma_g^c(s) = \bigcup_{v \models g} \Gamma_v^c(s)$ (and similarly for the observations).

3.3.3 Modelling the Interaction

The interaction between an operator and a system modelled with the enriched HMI-LTSs can be defined in the same way as it is defined

with HMI-LTSs. From a *composite state* denoted (s_S, s_H), there are different possible interactions. Either the operator can perform a possible command or the operator can see a possible observation. Of course, those two visible actions can only happen if the state-value of the current state of the system agrees with the action guard that is present on the mental model. As it was the case for HMI-LTS, internal actions may also occur inside the system. The mental model is still considered deterministic and free of τ-transitions, that is, non-divergent, and the two models share the same alphabet of actions and of states-observations.

Definition 3.16 (Interaction between a system and a mental model)**.** *Given an HVS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O} \rangle$ *and an HVM* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^v \rangle$*, the interaction between* $\mathcal{S}$ *and* $\mathcal{H}$*, denoted* $\mathcal{S} \parallel_I \mathcal{H}$ *, is an LTS* $\mathcal{I} = \langle S_I, \mathcal{L}^c \cup \mathcal{L}^o, s_{0_I}, \rightarrow_I \rangle$ *where* $S_I \subseteq (S_S \times S_H)$*,* $s_{0_I} = (s_{0_S}, s_{0_H})$ *and* $\rightarrow_I \subseteq S_I \times (\mathcal{L}^c \cup \mathcal{L}^o \cup \{\tau\}) \times S_I$ *is defined so that:*

- $(s_{S_1}, s_{H_1}) \xrightarrow{\alpha} (s_{S_2}, s_{H_2})$ *if and only if* $s_{S_1} \xrightarrow{\alpha} s_{S_2}$ *and* $s_{H_1} \xrightarrow{[v]\,\alpha}$ s_{H_2} *with* $v = \mathcal{O}(s_{S_1})$

- *and* $(s_{S_1}, s_{H_1}) \xrightarrow{\tau} (s_{S_2}, s_{H_1})$ *if and only if* $s_{S_1} \xrightarrow{\tau} s_{S_2}$*.*

Figure 3.17 shows the *interaction model* for the countdown machine example. The states from that model are composite states indicating the state of the system and the state for the user. In this case, the interaction model has exactly the same number of states and the same behaviour as the system model. That comes from the fact that the mental model has a nice property which allows an operator following that mental model to be able to control properly the system. That is developed in the next chapter. Also note that, even if it is not directly included in the interaction model, information about the state-values is indeed available thanks to the $\mathcal{O}$ function that can be applied to the state of the system model which is part of the composite states of the interaction model. That information can be useful, for example to know what state-values were conditioning the transitions.

3.4 Alternate Models for HMI

Choosing a formalism to model HMI can be done according to several criteria. One concern is how the notation allows the system designers and
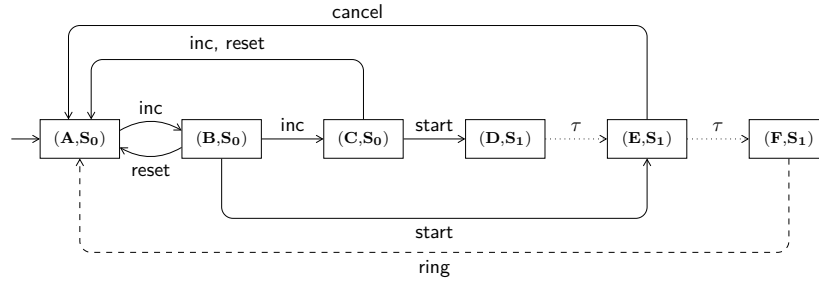
Figure 3.17. The interaction model for the countdown machine example.

analysts to express their design model. Another element that can vary among the formalisms is how they can be used to perform formal analyses on the models, in order to get interesting analyses within reasonable computation time.

Moreover, it is important to distinguish the formal model from the design language. The *formal model* is the mathematical formalism used in order to reason about the models and to propose algorithms capable to analyse them. Contrarily, the *design language* is the one that will be used by the system designers. Once designed, systems described within a design language are translated into the formal models in order to get analysed. The results of the analyses are then transmitted back to the system designer, after another translation in the other way.

This section is not concerned with design languages but focuses on formal models. It does not provide a detailed review of all the existing formalisms, but rather presents different kinds of formalisms and compares them to the LTS-based approach chosen in this work. A survey of various existing formal notations to model human-machine interactions is available in [Jac83]. This section focuses on the following formalisms: labelled transition systems with input and output and input-output transition systems [Tre08] which are the closest formalisms to the one chosen in this thesis, interface automata [dAH11], statecharts [HP85, Har87] or tabular notations [HKB08] which provide a high-level visual notation, modal specifications [Lar90] and finally mode automata [HJS01] that highlight the notion of mode.

Original notations have been preserved most of the time, except when similar notations are defined in this thesis, in which case the notations

of this thesis are used in order to ease comprehension. The focus in this section is only on the modelling aspects, while the analysis capabilities are discussed in the next chapter.

3.4.1 LTS with Inputs and Outputs

The importance of inputs and outputs has been highlighted in [Tre08]. In the setting of the work by Tretmans on model-based testing, systems are interacting with the environment. The communication between systems and their environment is made through inputs and outputs. Outputs are actions initiated by the system while inputs are triggered by the environment. That distinction corresponds exactly to the command/observation distinction that is made in the work of this thesis.

Tretmans defines two kinds of models: *labelled transition system with input and output* (LTS/IO) and *input-output transition system* (IOTS). While LTS/IO are exactly the same as HMI-LTS, IOTS do differ from HMI-LTS. The difference is in the fact that for an IOTS, the environment can never refuse an output produced by the system and the system can never refuse and input that is sent to it by the environment. The direct consequence is that all inputs are possible in all states.

Definition 3.17 (Input-Output Transition System). *An* input-output transition system *(IOTS) is a tuple* $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$*, which is an HMI-LTS such that* $\forall s \in reach(\mathcal{M}) : A^c(s) = \mathcal{L}^c$*.*

Making a system *input-enabled*, meaning that inputs are accepted in any state, can be done in various ways. The definition of IOTS do not require anything special about the ways system is made input-enabled. Among all the possibilities, two are more common. One possible way is to complete the system by adding input self-loops for any input that is not possible. That way to proceed is referred to as *angelic completion*. Whenever an unforeseen input occurs, the system just ignores it, that is, stays in the same state. The other common completion possibility is the *demonic completion* where transitions are added for all the states, for all the inputs that are not possible, to a special *error state*, which is a special state from where all actions are possible, as self-loops.

Definition 3.18 (Error state). *Given an LTS $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, the error state for that LTS which is denoted Π is a state having all the transitions $\{\Pi \xrightarrow{\alpha} \Pi \mid \alpha \in \mathcal{L}\}$.*

LTS/IO and IOTS do not bring anything more than what is possible with HMI-LTS, except the notions of input-enablement that can also be defined for HMI-LTS. Figure 3.18 shows an example of an HMI-LTS with its angelic and demonic completion. In any state of the system, all the commands of the alphabet are possible.
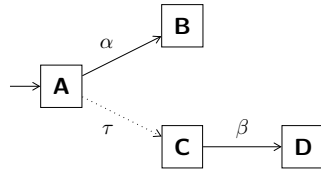
(a) An HMI-LTS with $\mathcal{L}^c = \{\alpha, \beta\}$.
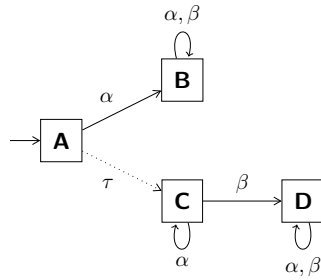
(b) Angelic completion of an HMI-LTS.
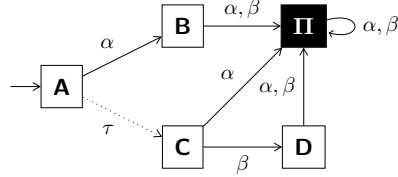
(c) Demonic completion of an HMI-LTS.

Figure 3.18. HMI-LTS can be completed in several ways to get an input-enabled system, that is, $A^c(s) = \mathcal{L}^c$ for any state s of the HMI-LTS. (b) In angelic completion, input self-loops are added to all the states and (c) in demonic completion, all the states are completed with transitions going to a special error state Π.

3.4.2 I/O and Interface Automata

I/O automata [LT87] and *interface automata* [dAH11] have been introduced to add inputs and outputs to automata. They are typically used in component-based design, a software engineering approach which focuses

on the *"separation of concerns"* principle, and they serve as an interface description language. The difference between I/O automata and interface automata is that the latter are not required to be input enabled. Except that, the two formalisms are syntactically similar, thus only interface automata are considered in this section. An *interface automaton* consists of a set of states and a set of actions that are partitioned into input, output and internal actions.

Definition 3.19 (Interface Automaton)**.** *An* interface automaton *is a tuple* $\mathcal{P} = \langle V, \mathcal{V}^{init}, \mathcal{A}^I, \mathcal{A}^O, \mathcal{A}^H, \mathcal{T} \rangle$, *where* V *is a set of states,* $V^{init} \subseteq V$ *is the set of initial states which is required to be non-empty,* $\mathcal{A}^I, \mathcal{A}^O$ *and* $\mathcal{A}^H$ *are mutually disjoint sets of input, output and internal actions and finally* $\mathcal{T} \subseteq V \times \mathcal{A} \times V$ *is the set of steps, where* $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O \cup \mathcal{A}^H$.

Those two formalisms are similar to LTS/IO and IOTS in what they model. The only difference is that with automata, internal actions are explicit and not gathered into one single τ action. From a modelling point of view, they do not bring anything more than HMI-LTS.

A specialised version of interface automata defined in [dAH11], namely *single-threaded interface automata*, is used to model systems that are made of a single thread of execution. The idea is to divide states into two categories: the states where only internal and output actions are enabled, and those where only input actions are enabled. The states from the first category are called *running states* since the system is in execution mode. The states from the second category are called *waiting states* since the system is waiting for inputs from the user.

Definition 3.20 (Single-Threaded Interface Automaton)**.** *A* single-threaded interface automaton *is an interface automaton* $\mathcal{P} = \langle V, \mathcal{V}^{init}, \mathcal{A}^I, \mathcal{A}^O, \mathcal{A}^H, \mathcal{T} \rangle$ *satisfying the two following conditions:*

1. *The set of states is partitioned into two sets* $V = V^O \cup V^I$ *of running and waiting states. For all states from* V^O, *only internal and output actions are enabled. For all states from* V^I, *only input actions are enabled.*

2. *All transitions with output actions must lead to a waiting state and all transitions going to a waiting state have an output action.*

Single-threaded interface automata model interactions where there is an alternation between locally controlled actions (internal and output

actions) and input actions. That partition adds a constraint that is not
present in HMI-LTSs, but single-threaded interface automata are indeed
somewhat similar to HVS. It is indeed the case if it is considered that
the operator alternates between checking a state-value and performing
a command or making an observation. The difference resides in the
kind of partitioning that is done between the two types of states. Those
considerations are developed in the next chapter, where the translation
between HVS and HVM towards HMI-LTS is described. However, single-
threaded interface automata models are close to ADEPT models. In
particular, as described in Chapter 7, HVS obtained from the translation
of ADEPT models with the proposed semantics are indeed single-threaded
interface automata models.

3.4.3 Statecharts

LTSs and automata with inputs and outputs are low-level models where
there is no organisation among the states, except a partition between
running and waiting states for single-threaded interface automata.

Statecharts [Har87] are an extension of state machines developed by
Harel in order to get a suitable formalism to specify and design discrete-
event systems. The additions to state machines brought possibilities
to model hierarchy, concurrency and communications. Statecharts are
also used by Degani et al. [DH02] as the modelling formalism used in
their work. Another advantage of statecharts is that they are visually
intuitive and provide a compact way to present a formal definition of the
behaviour of a system.

Moreover, they have been defined with as main purpose to model
reactive systems, which makes them quite suitable to describe models for
human-machine interaction analyses. The Therac-25 example presented
in the previous chapter (Figure 2.11 on page 46) is an example of a
statechart. Another example presented on Figure 3.19 shows the state-
chart of the mode part of a simple microwave. But statecharts remain
a visual formalism and should be rather classified as a design language.
Nevertheless, formal semantics have been defined for statecharts, and
especially in terms of LTSs [US94, LvdBC99].
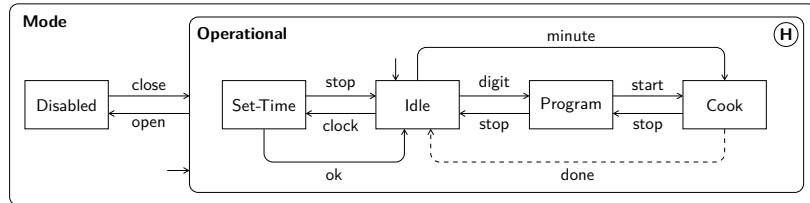
Figure 3.19. Example of a statechart representing the mode part of the behaviour of a microwave oven (taken from [Luc93]).

3.4.4 Modal Specifications

Modal specifications defined by Larsen [Lar90] allows for loose specifications of systems. The idea behind modal specifications is that it is possible to define what behaviour is necessary and what behaviour is admissible for a valid implementation of the specification. A *modal transition system* (MTS) is essentially an LTS whose transitions are partitioned into necessary and admissible transitions.

Definition 3.21 (Modal Transition System)**.** *An modal transition system (MTS) is a tuple* $\mathcal{P} = \langle S, A, \rightarrow_\square, \rightarrow_\lozenge \rangle$*, where S is a set of specifications, A is a set of actions and $\rightarrow_\square, \rightarrow_\lozenge \subseteq S \times A \times S$, satisfying $\rightarrow_\square \subseteq \rightarrow_\lozenge$, are respectively the sets of necessary and admissible transitions.*

Contrarily to other formalisms presented so far, an MTS does not only model one system, but rather a set of systems satisfying some constraints. MTS adds the possibility to define a set of admissible transitions. This kind of model is much used in the model-based testing field, where the main goal is to test whether an implementation respects a given specification. Therefore, necessary transitions must be present in valid implementations whereas admissible transitions may be part of it. The formalisms presented previously also have a sort of constraints on the transitions. They implicitly include the notion of a "forbidden transition", a transition that is not present in the model.

From a human-machine perspective, such distinction among the transitions can be used to analyse the interaction. Given a system model, it could be interesting to distinguish which are the execution scenarios that must be known and those which may be known in order to operate

correctly the system. It is not possible with HMI-LTS to directly model admissible transitions but, as developed in Chapter 5, the executions of system models can be indeed classified into three sets: necessary, admissible and forbidden executions.

3.4.5 Mode Automata

As stated above in this chapter, there are systems which clearly exhibit independent running modes. For analysis purposes, it is worth to have a modelling formalism that is able to encode explicitly those modes. Maraninchi et al. [MR98] define *mode automata* which are used to express a mode structure in a reactive system. They are focused on integrating modes into dataflow languages used to model reactive systems, whose family is known as *synchronous languages*. Mode automata have been designed with two objectives. The first one is that the mode structure, and in particular how the modes are organised into the global behaviour of the system, must be described. The second goal is to offer the ability to get a projection of the system on one mode so as to obtain the behaviour of the system restricted to a single mode.

A mode automaton is composed of an automaton part enriched with dataflow equations coming from the Lustre dataflow language [CPHP87]. The states of mode automata correspond to the running modes of the system and the dataflow equations represent the control laws of the system. Figure 3.20 shows an example of a mode automaton that has two modes (incrementation and a decrementation) represented by the two states **A** and **B**.
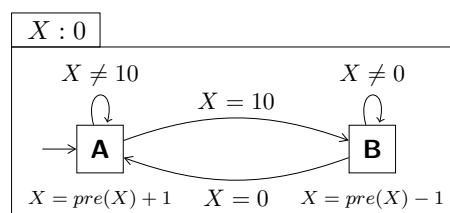
Figure 3.20. Graphical representation of a mode automaton example [MR98]. The mode automaton has two modes corresponding to the incrementation or decrementation of the X integer variable.

Definition 3.22 (Mode Automaton)**.** *An* mode automaton *is a tuple* $\mathcal{P} = \langle Q, q_0, \mathcal{V}_i, \mathcal{V}_o, \mathcal{I}, f, T \rangle$, *where* Q *is a set of states,* $q_0 \in Q$ *is the initial state,* $\mathcal{V}_i$ *and* $\mathcal{V}_o$ *are mutually disjoint sets of names for input and output integer variables,* $T \subseteq Q \times \mathcal{C}(\mathcal{V}_i \cup \mathcal{V}_o) \times Q$ *is the set of transitions, labelled with conditions on the variables,* $\mathcal{I} : \mathcal{V}_o \mapsto \mathbb{Z}$ *is a function defining the initial values of the output variables and finally* $f : Q \mapsto \mathcal{V}_o \mapsto \mathsf{EqR}$ *defines the labelling of states with total functions from output variables to right expressions.*

Mode automata are required to be *deterministic* and *reactive.* The first condition just means that the underlying automaton must be deterministic. The second requirement means that for any state of the mode automaton, there must always be a transition whose condition is satisfied.

In the vein of what operations are possible with statecharts, Maraninchi et al. defined a parallel composition operator that is useful whenever the modes of a system can be split into several orthogonal sets. Also, they provide ideas for hierarchic modes where the states belonging to a given mode can be split into sub-modes. Maraninchi et al. provide a semantic for mode automata, based on Mini-Lustre, a subset of Lustre defined by the same authors.

Chapter 4

Full-Control Property

This chapter introduces automation surprises issues and defines the full-control property which is a property between a system and a mental model. The full-control property captures the notion of good interaction, that is, interaction free of automation surprises. Section 4.1 gives the definition and intuition of the full-control property and provides a characterisation of the property. Then, Section 4.2 presents in detail the full-control property for HMI-LTSs. In particular, the full-control determinism property which guarantees the existence of a full-control mental model is defined in the section. Finally, section 4.3 extends the results from the previous section to enriched models. Section 4.4 relates the full-control property to other common properties used in similar situations.

4.1 Characterisation of Good Interaction

As introduced in Chapter 2, the work of this thesis is focused on the characterisation of good interaction between an operator and a system. This section describes what is behind *"good interaction"* and how it is related to the work by Degani et al., which is the starting point of this work. As a reminder, the goal of Degani et al. is to generate a model for the user so that using the model to operate the system makes him avoid potential automation surprises, that is, situations where the operator is faced with an unexpected behaviour of the system, according to his own mental model of it. In their work, Degani et al. focus on mode confusion issues, and they defined a correct user interface being one where the three following kinds of error are absent from any possible interaction:

- an *error state* corresponds to a situation where the machine is in a certain mode, while the user thinks it is in another one;

- a *restricting state* represents a situation where the user could trigger a mode change, but that change is not present in his mental model;

- and an *augmenting state* is a state where the user thinks that some actions are possible according to his mental model, but those actions are effectively not available on the machine.

Those three situations are to avoid since they may cause surprise. They can be identified based on the interaction model. Intuitively, a check on the composite states of the interaction model can be done, to test whether there are situations where the machine and the user are not agreeing anymore on what further behaviour is possible. Such situations where the behaviours diverge are in fact spots of potential automation surprises.

In this work, an interaction between an operator and a system is qualified as *"good"* if no potential automation surprise ever occurs during the interaction. For that to be possible, the operator must always be able to predict the behaviour of the system, at any time during the interaction. This point of view is more general than the one of Degani et al. who only consider that the user must be able to predict the next mode of the system. For the user to be able to predict the behaviour of the system, two conditions have to be satisfied between the predicted and the actual behaviours:

1. The operator must always know *exactly the commands* that are available on the system. Whenever a command is performed on the system, it is guaranteed that it will be accepted by it.

2. Whenever the system provides any feedback to the operator, it must be foreseen by the operator. That condition does not prevent the operator to expect or be prepared for *more observations* than exactly those which can effectively occur in the current state of the system.

This notion of good interaction is captured by the *full-control* property which is the subject of this chapter. The property must hold between the operator and the system being used, in order to guarantee the absence of potential automation surprises. Of course not all automation surprises are taken into account by the full-control property. The property rather captures a notion of controllability without surprises.

The full-control property can be defined based on the interaction model presented in the previous chapter. The idea is that at any reachable state of the interaction model, the set of expected commands must coincide exactly with the set of actual commands whereas the set of expected observations can be a superset of the set of actual observations. A mental model such that the two conditions actually hold for a given system model is said *to allow full-control* of the system and it is referred to as *a full-control mental model* for the system. A system model for which there exists a full-control mental model is said to be *full-controllable.*

4.1.1 Potential Automation Surprises

The full-control property ensures that situations that may surprise the operator of a system do not occur during the interaction. The bad situations described by Degani et al. [HD07] and summarised in Section 2.4.7 are taken into account by the full-control property as detailed below. Figure 4.1 shows a composite state of the interaction model, with the four potentially confusing situations that may occur.
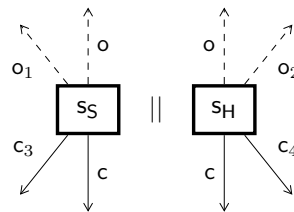
Figure 4.1. The four potentially surprising situations that may occur during the interaction between a human $\mathcal{H}$ and a system $\mathcal{S}$ being used, in every state of the interaction model.

The four situations that may occur are:

1. There is an observation o_1 that may be produced by the system, but the operator is not aware of it according to the mental model. In that situation, the operator may get surprised when the observation occurs, and will not know how to react. That could be dangerous if the observation is a hazard alert signal, for example.

2. According to his mental model, the operator expects an observation o_2 to occur but it will indeed never occur in the system. Such a situation is not so severe since the observations are under control of the system. It can only disturb the operator if he is actively waiting for the observation and does not get any information from the system that it will in fact never occur.

3. There is a command c_3 on the system that is available in the current state but that the operator is not aware of according to his mental model. That is not a surprising situation at all since the commands are under control of the operator. The only issue that such a situation raises is that it does not allow the user to use all the functionalities of the system.

4. There is a command c_4 that the operator can perform according to his mental model but that is in fact not available on the system. It can confuse the user since he will perform the command and expect some behaviour from the system that will never occur.

Situations 1, 3 and 4 are all avoided by the full-control property, that is, there cannot be an observation in the system model which is not in the mental model and the commands must be the same in both the system and mental model, at any time during the interaction. Situations 1 and 4 correspond to an automation surprise and may lead to incidents or accidents, since in both situations the user can be surprised either by an event that was not expected or by an action that does not perform what is expected. The situation 1 corresponds to Degani et al.'s blocking state and the situation 4 corresponds to an augmenting state. Situation 3 does not induce an automation surprise but is avoided by the full-control property. A variant of the full-control property is more flexible and allows the system to have available commands that the user is not aware of according to his mental model. That variant is discussed in Chapter 6.

In summary, the full-control property guarantees that there will never be any potential automation surprise situations during the interaction, but also that the operator will be able to use all the functionalities of the system.

The remainder of this chapter focuses first on the full-control property for HMI-LTSs. Then, the presented results are extended for the general

situation where HVS and HVM are used. The extension is based on a transformation of HVS and HVM models into equivalent HMI-LTSs.

4.2 Full-control Property for HMI-LTSs

This section develops on the full-control property for HMI-LTSs. After stating formally the definition of the full-control property, issues that can be introduced by τ-transitions are discussed. Then, the end of the section presents the condition that has to be satisfied on system models to guarantee the existence of a full-control mental model for those systems.

4.2.1 Full-control Property

The *full-control property*, as stated above, can be defined based on the interaction model. For any reachable state of the interaction model, two conditions must hold, relating the sets of commands (resp. observations) expected by the operator with the sets of commands (resp. observations) actually available on the system.

Definition 4.1 (Full-Control Property for HMI-LTSs)**.** *Given two HMI-LTSs* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ *and* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, $\mathcal{H}$ *is said to* allow full-control *of* $\mathcal{S}$, *which is denoted* $\mathcal{H} \mathbf{fc} \mathcal{S}$, *if and only if for all reachable composite states* $(s_S, s_H) \in \mathcal{S} \parallel \mathcal{H}$, *the two following conditions hold:*

1. $A^c(s_S) = A^c(s_H)$
2. $A^o(s_S) \subseteq A^o(s_H)$

Figure 4.2 shows an extended version of the vending machine example used in Chapter 3. A new observation is added to inform the user if the machine is empty and cannot therefore serve coffee. The mental model has also been extended. Two additional behaviours have been added:

- The user is ready to accept directly a free coffee from the vending machine;
- The user thinks that it is possible to order a coffee with a single coin, but also with two coins.

The proposed mental model for this vending machine does not allow full-control of it. It can be easily noticed on the interaction model since there are five states where the full-control conditions are not satisfied (the greyed states).

The four different potentially bad situations captured by the full-control property are illustrated with that example:

- In composite state $(\mathbf{C}, 1)$ the system may produce an alert observation that is not foreseen by the operator, which may surprise him if it occurs. On that state, $A^o(\mathbf{C}) \nsubseteq A^o(\mathbf{1})$.

- In composite state $(\mathbf{A}, \mathbf{0})$ the user expects an coffee observation that will not occur on the system. That is not an issue according to the full-control property, since the user may expect more observations than those that can actually occur.

- In composite state $(\mathbf{A}, \mathbf{2})$ the operator is not aware that the command coin is available on the system. This may not lead to an automation surprise. That is rather a lack of knowledge of the user regarding an existing feature of the system. On that state, $A^c(\mathbf{A}) \nsupseteq A^c(\mathbf{2})$.

- In composite state $(\mathbf{D}, \mathbf{1})$ the operator thinks that the command coin is available on the system, but it is actually not the case. If the operator does execute the command, he may get confused since the system is not responding as he expects. On that state, $A^c(\mathbf{D}) \nsubseteq A^c(\mathbf{A})$.

As a direct consequence of Property 3.12 (defined on page 73) which states that the set of traces of the interaction model is the intersection of the sets of traces of the system and mental models, the full-control property can also be expressed in term of traces that are common to the two models.

Property 4.2 (Full-Control Property for HMI-LTS). *Given two HMI-LTSs $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, $\mathcal{H}$ is said to* allow full-control *of $\mathcal{S}$, which is denoted $\mathcal{H} \mathbf{fc} \mathcal{S}$, if and only if for all $\sigma \in \mathcal{L}^*$ such that $s_S \in (s_{0_S} \mathbf{after} \sigma)$ and $s_H \in (s_{0_H} \mathbf{after} \sigma)$:*

$$A^c(s_S) = A^c(s_H) \text{ and } A^o(s_S) \subseteq A^o(s_H)$$
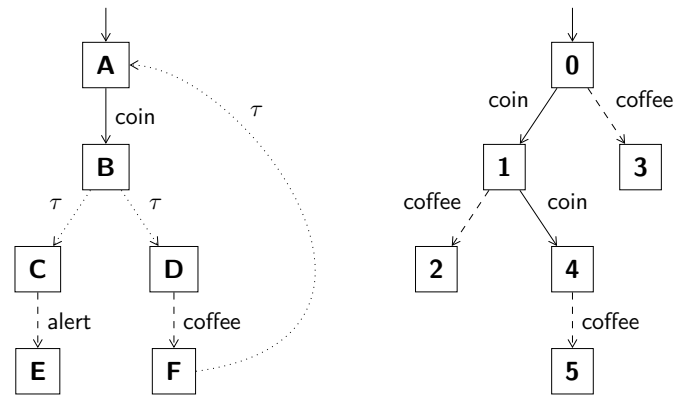
(a) System model $\mathcal{S}$.

(b) Mental model $\mathcal{H}$.
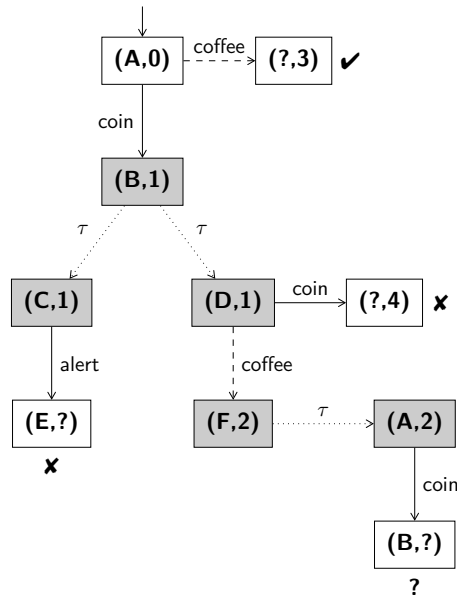
(c) Interaction model $\mathcal{S} \parallel \mathcal{H}$.

Figure 4.2. Example of a vending machine system modelled with an HMI-LTS $\mathcal{S}$. The proposed mental model $\mathcal{H}$ does not allow full-control of the system since there are states of the interaction model $\mathcal{S} \parallel \mathcal{H}$ where the full-control criterion is not satisfied (highlighted with a grey background colour).

Checking Full-control

Checking whether a mental model allows full-control of a system model
is a task that can be done easily with an exploration of the interaction
model. Once the interaction model is built, it suffices to explore it, with
a DFS for example, and to check the full-control conditions for each
composite state. Actually, it is in practice not necessary to first build
the interaction model since the conditions can be checked on-the-fly.
Algorithm 5 in Appendix 5 performs the full-control check. It builds the
interaction model on-the-fly in a BFS fashion so as to get the shortest
counterexample if the full-control property is not satisfied.

The full-control check algorithm can be seen as an execution of
a model checking on the interaction model. Consequently, the time
complexity of the algorithm is in $\mathcal{O}(n + m)$ where n is the number of
states and m the number of transitions of the interaction model. In the
worst case, the interaction model has $n_S n_H$ states

4.2.2 Enabled or Possible Sets of Commands and Observations

The sets of actions that are used in the definition of full-control are the
possible sets (A) and not the enabled sets (Γ), meaning that actions that
can occur after τ-transitions are also taken into account. That choice
has an important consequence on the property that is captured and
on the assumed behaviour of the operator. The fact that possible sets
are used implies that the equality of the sets of commands at any time
during interaction must be verified in the weak sense. A command that
is possible for the operator means that the operator may perform it on
the system, but maybe not directly since he may have to wait for some
time, during which some internal transitions will occur in the system.

Figure 4.3 illustrates the difference between enabled and possible sets.
The difference between the two systems resides in the fact that in the
left system, the operator is sure that he can *directly* execute commands
c_1 and c_2, that is, $\Gamma^c(\mathbf{A}) = \{c_1, c_2\}$. For the system on the right, the
operator is only sure that he may execute the commands, after having
potentially waited for a certain amount of time, that is, $A^c(\mathbf{A}) = \{c_1, c_2\}$.
Only command c_1 is immediately available since $\Gamma^c(\mathbf{A}) = \{c_1\}$.

According to the full-control definition, the model of Figure 4.3(a) is
a full-control mental model for both system models of Figure 4.3. For the

system of Figure 4.3(a), it is obvious that it allows full-control of it, but less so for the system of Figure 4.3(b). Figure 4.4 shows the interaction model and it is easily verified that for each composite state of that system, the full-control conditions on commands and observations sets are satisfied. That interaction model clearly shows that the c_2 command is possible when the system is either in the **A** state or in the **C** state, except that he will not be able to perform it directly if the system is still in the **A** state.

4.2.3 Full-control Compatibility and Determinism

Given a model of a system, some of its states may exhibit similar behaviour according to the point of view of the operator, in a full-controllability perspective. Such states are said to be *full-control compatible*. It is important to be able to identify those states since, in order to explain the behaviour of a system to an operator, they do not need to be explained separately. Two states are full-control compatible if they agree on possible commands for any common execution trace starting from them.

Definition 4.3 (Full-Control Compatibility). *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow \rangle$, the two states $s_1, s_2 \in S$ are* full-control compatible *(fc-compatible in short), which is denoted $s_1 \approx_{\mathbf{fc}} s_2$, if and only if for all $\sigma \in \mathcal{L}^*$ such that $s_1' \in (s_1 \, \mathbf{after} \, \sigma)$ and $s_2' \in (s_2 \, \mathbf{after} \, \sigma)$:*

$$A^c(s_1') = A^c(s_2')$$

The full-control compatibility corresponds exactly to the definition of operational determinism (Definition 3.8 on page 66), but restricted to commands. The interpretation of the fc-compatible property between two states s_1 and s_2 is that, no matter in what state the operator is, he can execute any command from the set $A^c(s_1) = A^c(s_2)$, and be sure to be able to predict what will be the possible commands in the reached states.

For the system example of Figure 4.3(b), the states **A** and **C** are fc-compatible since they agree on commands for all common traces (ε and a). In both states **A** and **C**, if the operator does nothing the possible commands are $\{c_1, c_2\}$ and after the operator performs an c_1 or c_2 command, there is no possible further command.

(a) A system model with $\Gamma^c(\mathbf{A}) = A^c(\mathbf{A}) = \{c_1, c_2\}$. That system model is also a mental model for itself.

(b) A system model with $\Gamma^c(\mathbf{A}) = \{c_1\} \neq A^c(\mathbf{A}) = \{c_1, c_2\}$.
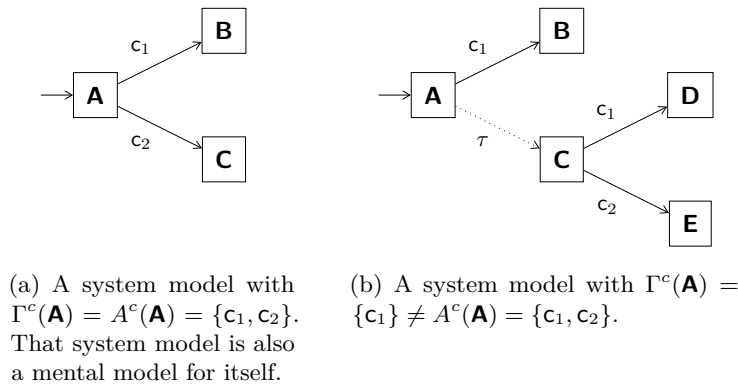
Figure 4.3. Two system models whose initial states do not have the same set of enabled commands, but do have the same set of possible commands. For the system model on the left, the operator knows that he can always perform directly commands c_1 and c_2, that is, $\Gamma^c(\mathbf{A}) = \{c_1, c_2\}$. For the system model on the right, the operator cannot execute directly commands c_1 and c_2, but can possibly execute them, that is, the operator knows that he should maybe wait some time before being able to execute them. In that situation, $\Gamma^c(\mathbf{A}) = \{c_1\} \neq A^c(\mathbf{A}) = \{c_1, c_2\}$.
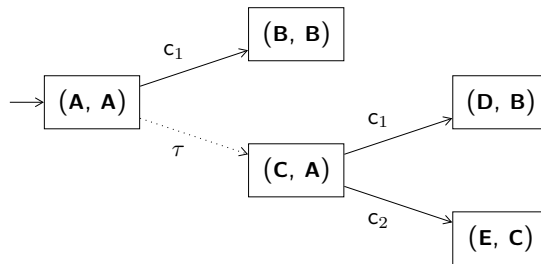
Figure 4.4. The interaction model between the system and mental models of Figures 4.3(b) and 4.3(a). The full-control conditions are satisfied for every composite state which means that the mental model allows full-control of the system model.

The fc-compatibility property only requires the set of possible commands to be equal after all the common traces. Figure 4.5 shows an example where the two states **A** and **D** are fc-compatible even if after executing the c command, the two states that are reached (respectively **B** and **E**) do not have the same set of actions. Since the differences are only caused by observations, it is not an issue. Since **A** and **D** are fc-compatible, it means that they share the same behaviour from the user's point of view. The common behaviour is that the user is, in both cases, able to execute a c command, and after having executed the command, he can expect one observation from the set $\{o_1, o_2\}$. That ability to put together several observations that may not always occur, in the same set is a direct consequence of the non-symmetric characteristic of the full-control property, which is more flexible for observations. That is the key point which makes it possible to identify more compatible behaviours in a given system, that consequently only have to be presented once to the user in his mental model.
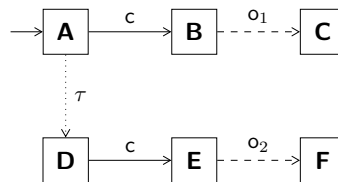
Figure 4.5. States **A** and **D** are fc-compatible even if it is possible to reach two different states with different sets of possible actions.

Characteristics of fc-compatibility

The fc-compatibility relation is *symmetric*, by construction. However, it is not reflexive nor transitive. The fc-compatibility relation is *not reflexive* as illustrated by the example of Figure 4.6. In this example, the initial state **A** is not fc-compatible with itself since there exists one trace violating the fc-compatibility condition, namely the empty trace. The system can indeed reach both states **A** and **C** with the empty trace, but those states do not have the same set of possible commands: $A^c(\mathbf{A}) = \{c_1, c_2\} \neq A^c(\mathbf{C}) = \{c_2\}$.

Intuitively, it means that without doing anything, the operator may be refused some commands just because the system has changed its state
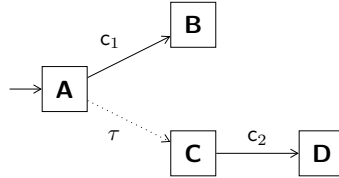
Figure 4.6. HMI-LTS example of a system illustrating that the $\approx_{\mathbf{fc}}$ relation is not reflexive. The state A is not fc-compatible with itself.

in-between and that the command that was available for the user is not available anymore. Initially, the user was able to perform the commands c_1 and c_2, but after the τ-transition occurred in the system, only the c_2 command remains possible. Such situations are bad in terms of full-controllability since, without any interaction from the operator, the set of commands that are possible change. That situation is in contradiction with the full-control property and makes the system intrinsically not full-controllable. A direct consequence is that it is impossible to build any full-control mental model for such a system.

The issue of non-full-controllability is introduced by the τ-transition that adds non-determinism in the system. Not all non-determinism is bad, as illustrated by the three examples of Figure 4.7, which are non-deterministic but for which there exist full-control mental models. What makes those systems full-controllable is that the non-determinism they contain is not harmful since it leads to states that are fc-compatible. The third example $\mathcal{S}_3$ is operationally deterministic, in opposition to the two first examples that are not, neither are they (structurally) deterministic.

The fc-compatibility relation is also *not transitive* as illustrated by the example of Figure 4.8. The states **B** and **C** are fc-compatible since they do not share any common traces except the empty trace, and following it from any of both states leads to states with the same set of commands (which is the empty set). The same reasoning can be done for states **D** and **C**. However, states **B** and **D** are not fc-compatible. Indeed, following the o_1 common trace leads to states **E** and **F** which do have different sets of possible commands: $A^c(\mathbf{E}) = \{c_4\} \neq A^c(\mathbf{F}) = \{c_5\}$. The intuition is that the user can safely consider as similar either states **B** and **C** or states **C** and **D**, but not both.

(a) A system model $\mathcal{S}_1$.   (b) A system model $\mathcal{S}_2$.   (c) A system model $\mathcal{S}_3$.

(d) A full-control mental model for $\mathcal{S}_1$. (e) A full-control mental model for $\mathcal{S}_2$. (f) A full-control mental model for $\mathcal{S}_3$.
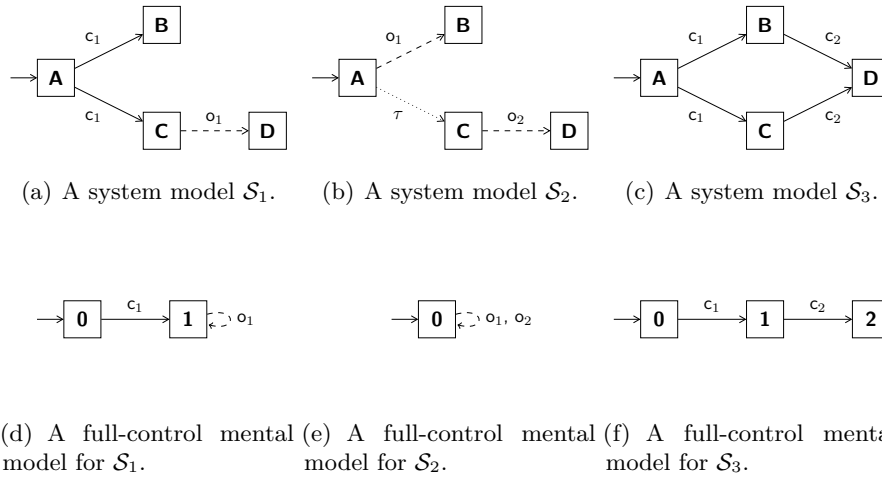
Figure 4.7. Three HMI-LTS examples of systems that are not deterministic but for which it exists a full-control mental model.
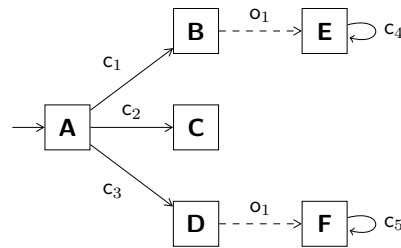
Figure 4.8. HMI-LTS example of a system illustrating that the $\approx_{\mathbf{fc}}$ relation is not transitive. States B and C and states D and C are pairwise fc-compatible, but states B and D are not fc-compatible.

Full-control determinism

Full-control determinism, or fc-determinism in short, characterises system models for which there exists a full-control mental model. The only condition that has to be satisfied is that given a trace, the set of states that are reachable from the initial state with that trace all have the same set of possible commands. The fc-determinism property is in fact the particular case of fc-compatibility of the initial state with itself.

Definition 4.4 (Full-Control Determinism for HMI-LTS). *An HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow \rangle$ *is* full-control deterministic *if and only if for each* $\sigma \in \mathcal{L}^*$:

$$\forall s, s' \in (s_{0_S} \textbf{ after } \sigma) : A^c(s) = A^c(s')$$

that is $s_{0_S} \approx_{\textbf{fc}} s_{0_S}$.

Checking Full-control Determinism

Full-control determinism of a given system can be checked by computing the synchronous parallel composition of the system with itself. A system model $\mathcal{S}$ is fc-deterministic if and only if the fc-determinism condition is satisfied for every composite state of the composition.

Property 4.5 (Full-control determinism check). *Given an HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow \rangle$ *and the interaction model* $\mathcal{S} \parallel \mathcal{S}$:

> $\mathcal{S}$*is fc-deterministic*
> $\iff \forall \sigma \in \mathcal{L}^* : \forall (s, s') \in ((s_{0_S}, s_{0_S}) \textbf{ after } \sigma) \text{ in } S \parallel S : A^c(s) = A^c(s')$

Proof. If the system $\mathcal{S}$ is fc-deterministic, it implies by definition that $\forall \sigma \in \mathcal{L}^* : \forall s, s' \in (s_{0_S} \textbf{ after } \sigma) : A^c(s) = A^c(s')$. The set $(s_{0_S} \textbf{ after } \sigma)$ is not empty only if $\sigma \in \textbf{Tr}(\mathcal{S})$. Consequently, it means that the trace σ also belong to $\textbf{Tr}(\mathcal{S} \parallel \mathcal{S})$ by definition of the synchronous parallel composition. Moreover, by definition of **after**, $s_{0_S} \overset{\sigma}{\Longrightarrow} s$ and $s_{0_S} \overset{\sigma}{\Longrightarrow} s'$, that is, (s, s') is a composite state of $(\mathcal{S} \parallel \mathcal{S})$ and $A^c(s) = A^c(s')$ by hypothesis.

The state (s, s') belongs to the synchronous parallel composition if it is reachable from the state (s_{0_S}, s_{0_S}). It means by definition that $s_{0_S} \overset{\sigma}{\Longrightarrow} s$ and $s_{0_S} \overset{\sigma}{\Longrightarrow} s'$ for some $\sigma \in \mathcal{L}^*$, that is, $s, s' \in (s_{0_S} \textbf{ after } \sigma)$. Since $A^c(s) = A^c(s')$, it means that $\mathcal{S}$ is fc-deterministic. $\square$

Checking fc-determinism is thus an execution of model checking on the $(\mathcal{S} \parallel \mathcal{S})$ model. Consequently, the time complexity of the algorithm is $\mathcal{O}(n + m)$ where n is the number of states of the composed model and m its number of transitions.

4.2.4 Existence of Full-control Mental Model

As stated in the previous section, a full-control mental model only exists if
the system model is fc-deterministic. This section proves it and presents
a procedure that can be used to build a full-control mental model for any
fc-deterministic system model, based on the determinisation algorithm
presented in Section 3.1.5.

Theorem 4.6. *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$:*

$$\mathcal{S} \text{ is fc-deterministic} \implies det(\mathcal{S}) \textbf{ fc } \mathcal{S}$$

Proof. Let $\mathcal{H} = det(\mathcal{S}) = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$. By definition of the
determinisation, $\textbf{Tr}(\mathcal{S}) = \textbf{Tr}(\mathcal{H})$. For a given sequence $\sigma \in \textbf{Tr}(\mathcal{S})$, there
is only one execution $s_{0_H} \xrightarrow{\sigma} s_H$ in the determinised model and by
construction $(s_{0_S} \textbf{ after } \sigma) = \{s_H\}$.

Let us first consider the commands: $A^c(s_H) = \bigcup_{s_S \in (s_{0_S} \textbf{ after } \sigma)} A^c(s_S)$
by construction. Since $\mathcal{S}$ is fc-deterministic, $A^c(s_S)$ are exactly the same
for all $s_S \in (s_{0_S} \textbf{ after } \sigma)$. Consequently, $s_{0_H} \xrightarrow{\sigma} s_H$ and $s_{0_S} \xRightarrow{\sigma} s_S$
with $A^c(s_H) = A^c(s_S)$.

Let now consider the observations: $A^o(s_H) = \bigcup_{s_S \in (s_{0_S} \textbf{ after } \sigma)} A^o(s_S)$
by construction. There are no constraints on observations and thus,
$s_{0_H} \xrightarrow{\sigma} s_H$ and $s_{0_S} \xRightarrow{\sigma} s_S$ with $A^o s_H \supseteq A^o(s_S)$.

To conclude, for any $\sigma \in \mathcal{L}^*$ such that $s_{0_H} \xrightarrow{\sigma} s_H$ and $s_{0_S} \xRightarrow{\sigma} s_S$,
it holds that $A^c(s_H) = A^c(s_S)$ and $A^o(s_H) \supseteq A^o(s_S)$, that is, $\mathcal{H}$ allows
full-control of $\mathcal{S}$. $\square$

Minimal Full-control Mental Model

Generally, there is no single unique mental model for a given fc-determi-
nistic system model. The previous section shows one possible full-control
mental model which is the determinised version of the system model.
The purpose of the mental model is to help a human operator to use a
system by generating training material or to design training sessions, for
example. Mental models can also be used for analysis purposes. For all
those uses, it is preferable to have the smallest possible mental model,
in order to reduce the size of training materials, to be easier to explain
to a human, to be possible for the human to memorise it and finally to
decrease the time and space costs for the automated analyses.

Figure 4.9 shows an fc-deterministic system model (Figure 4.9(a)) and two full-control mental models for it. The first one (Figure 4.9(b)) is obtained by computing the determinisation of the system model. The second one (Figure 4.9(c)) is a minimal full-control mental model, that is, the one with the smallest number of states.

(a) A system model $\mathcal{S}$.   (b) The determinised system   (c) The minimal
                                    model $det(\mathcal{S})$.   full-control mental
                                                                            model $\mathcal{H}$.
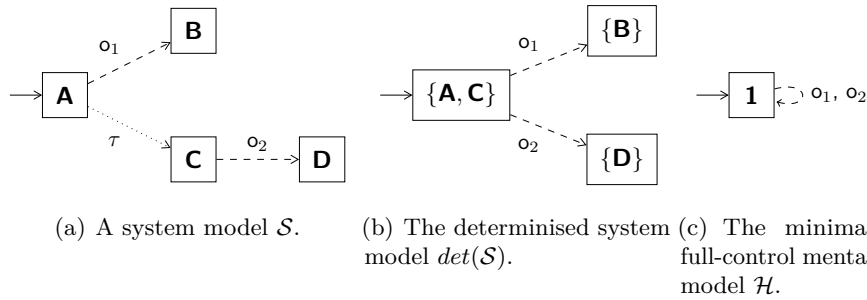
Figure 4.9. HMI-LTS example of system model $\mathcal{S}$ with the corresponding determinised system model $det(\mathcal{S})$ which is not a full-control mental model for $\mathcal{S}$.

Determinisation does not achieve minimality and it only ensures that the traces are preserved. Moreover, actions are all processed on an equal footing, the determinisation procedure does not distinguish between commands and observations. The set of traces of the system model example of Figure 4.9(a) is $\mathbf{Tr}(\mathcal{S}) = \{\varepsilon, c_1, c_2\}$. The determinised system has the same set of traces, by construction. The particularity about full-control is that it allows the mental model to have more observations than those which can effectively occur on the system. Using that flexibility, it is possible to build much more compact mental models. This is precisely what drives the next chapter. The minimal mental model for the system of Figure 4.9(a) has only one state and its set of traces is $\mathbf{Tr}(\mathcal{H}) = (c_1 + c_2)^*$, that is, any sequences composed with any number of c_1 and c_2, including the empty trace.

4.3 Full-control Property for Enriched Models

Enriched models presented in Section 3.3 add information on the states of the models. This additional information, state-values, may help the operator to know whether an internal action has occurred in the system.

The system model of Figure 4.10(a) is the same as the one of Figure 4.6, but on which state-values have been added. The issue with the original system is that it was not possible for the operator to distinguish between states **A** and **C**. In the enriched model, those two states do have a different state-value, respectively v_1 and v_2. In order to know whether the command c_1 is possible or not, that is, whether the internal action has taken place, the operator has to check whether the observed state-value is v_1 or v_2. For that reasoning to be valid, an assumption that is made is that the lookup of the state-value and the performing of the command occur at the same time as a single atomic event. Figure 4.10(b) shows a full-control mental model for the system of Figure 4.10(a). The command c_1 will only be performed if the state-value corresponding to the current system state is v_1 and similarly for the c_2 command with the v_2 state-value.

(a) System model.   (b) Mental model.
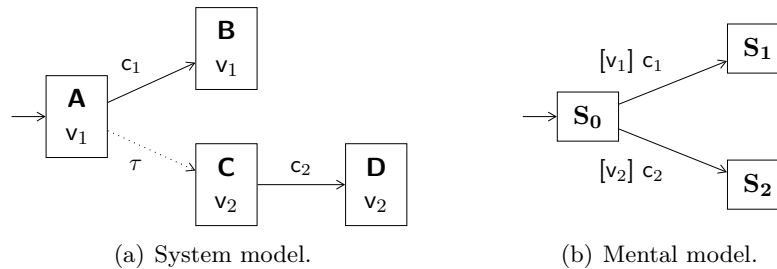
Figure 4.10. The system model of Figure 4.6, which is not fc-deterministic, has been enriched with state-values so that to make it fc-deterministic (on the left). The mental model on the right is a corresponding full-control mental model.

The full-control property can also be defined for the enriched models, based on the interaction model between the enriched system and mental models. The difference with the definition based on HMI-LTS is that the sets of possible commands and observations are now sets of pairs consisting of a state-value associated with an action. It is indeed assumed, according to the definition of the interaction model (Definition 3.16 on page 82), that an operator will never perform a command or see an observation whose action guard is not compatible with the state-value corresponding to the current state of the system.

Definition 4.7 (Full-Control Property)**.** *Given an HVS* $\mathcal{S} = \langle S_S,$ $\mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O}\rangle$ *and an HVM* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^v\rangle$, $\mathcal{H}$ *is said to* allow full-control *of* $\mathcal{S}$, *which is denoted* $\mathcal{H}\,\mathbf{fc}\,\mathcal{S}$, *if and only if for all reachable* $(s_S, s_H) \in \mathcal{S}\parallel_I \mathcal{H}$:

- $A_e^c(s_S) = A_e^c(s_H)$;
- *and* $A_e^o(s_S) \subseteq A_e^o(s_H)$

where $A_e^c(s_S) = \{(v, c) \mid \exists s_S \xrightarrow{\tau^*} s_S' \xrightarrow{c} s_S'' \wedge v = \mathcal{O}(s_S') \wedge c \in \mathcal{L}^c\}$ *and* $A_e^c(s_H) = \{(v, c) \mid \exists s \xrightarrow{[g]c} s' \wedge v \models g \wedge c \in \mathcal{L}^c\}$ *for HVMs. The* $A_e^o(s_S)$ *and* $A_e^o(s_H)$ *sets are defined similarly.*

4.3.1 Enriched Traces

Traces on HMI-LTS only capture the sequence of actions performed by the operator (commands and observations). For enriched models, the operator has the possibility to check the state-value before any action. *Enriched traces* capture the fact that the operator is always checking the state-value before performing an action, in a single atomic step. Enriched traces are sequences of pairs composed of a state-value and a label. They can be defined for enriched system models by taking into account the state-value of the state from which the executed action is enabled.

Definition 4.8 (Enriched trace for HVSs)**.** *Given an HVS* $\mathcal{S} = \langle S, \mathcal{L}^c,$ $\mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O}\rangle$, *a sequence* $\sigma = (v_1, \alpha_1) \cdots (v_n, \alpha_n) \in (\mathcal{L}^v \times \mathcal{L})^*$ *is an enriched trace of* $\mathcal{S}$ *if and only if there exists an execution* $s_0 \xrightarrow{\tau^*}$ $s_0' \xrightarrow{\alpha_1} s_1 \cdots s_{n-1} \xrightarrow{\tau^*} s_{n-1}' \xrightarrow{\alpha_n} s_n$ *such that* $\mathcal{O}(s_i') = v_{i+1}$. *The set of enriched traces of the HVS is denoted with* $\mathbf{ETr}(\mathcal{S})$.

For example, the set of enriched traces of the example of Figure 4.10(a) is $\{\varepsilon, (\mathsf{v}_1, \mathsf{c}_1), (\mathsf{v}_2, \mathsf{c}_1)\}$. The sequence $\langle(\mathsf{v}_1, \mathsf{c}_2)\rangle$ does not belong to the enriched traces of the system. Looking more closely at the execution $\mathsf{A} \xrightarrow{\tau} \mathsf{C} \xrightarrow{\mathsf{c}_2} \mathsf{D}$ corresponding to the trace $\langle\mathsf{c}_2\rangle$ indicates that, if the user is tracking the changes of state-values, he will see that the system transitions from v_1 to v_2. However, even if the user can see the state-value v_1 and after that execute the command c_2, the sequence $\langle(\mathsf{v}_1, \mathsf{c}_2)\rangle$ is not an enriched trace of the system since it is assumed in this work that the user observes the state-value and directly executes an action, in a single

atomic step, without leaving the time to the system to make an internal transition. Again, τ-transitions can introduce some instability in the system from a controllability perspective as it is detailed further in this section.

Enriched traces can also be defined for enriched mental models, just as they are defined for enriched system models. The difference is that the state-values come from the action guard.

Definition 4.9 (Enriched trace for HVMs)**.** *Given an HVM* $\mathcal{H} = \langle S, \mathcal{L}^c,$ $\mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v \rangle$, *a sequence* $\sigma = (v_1, \alpha_1) \cdots (v_n, \alpha_n) \in (\mathcal{L}^v \times \mathcal{L})^*$ *is an enriched trace of* $\mathcal{H}$ *if and only if there exists an execution* $s_0 \xrightarrow{[v_1]\alpha_1}$ $s_1 \cdots s_{n-1} \xrightarrow{[v_n]\alpha_n} s_{n+1}$. *The set of enriched traces of the HVM is denoted with* $\mathbf{ETr}(\mathcal{H})$.

4.3.2 Full-control Compatibility for Enriched Models

Full-control compatibility can be extended for enriched models. As already illustrated by the example of Figure 4.10, the addition of state-values may make a system model full-controllable, although it is not full-controllable without the state-values. Full-control compatibility for HMI-LTS ensures that the set of possible commands after a given trace is always the same, no matter the underlying execution. For enriched models, it may be possible that the set of possible commands are not the same, after the execution of a given trace, provided that the state-values of those states are different. In other words, if the operator can unambiguously distinguish states corresponding to different sets of possible commands, it is not an issue.

Definition 4.10 (Full-Control Compatibility)**.** *Given an HVS* $\mathcal{S} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O} \rangle$, *the two states* $s_1, s_2 \in S$ *are full-control compatible (fc-compatible in short), which is denoted* $s_1 \approx_{\mathbf{fc}} s_2$, *if and only if for all* $\sigma \in \mathcal{L}^*$ *such that* $s_1 \xRightarrow{\sigma} s_1'$ *and* $s_2 \xRightarrow{\sigma} s_2'$:

$$\mathcal{O}(s_1') = \mathcal{O}(s_2') \implies A^c(s_1') = A^c(s_2')$$

Full-control determinism is defined in the same way as for HMI-LTS. An HVS is *full-control deterministic* if and only if its initial state is fc-compatible with itself.

Figure 4.11 shows a system model that is not fc-deterministic. The property is not verified since for the empty trace, three states can be reached $(s_{0_S}\ \mathbf{after}\ \varepsilon = \{\mathbf{A}, \mathbf{C}, \mathbf{E}\})$ and among those states, two have identical state-values $(\mathcal{O}(\mathbf{A}) = \mathcal{O}(\mathbf{E}) = \mathsf{v}_1)$, but different sets of possible commands $(A^c(\mathbf{A}) = \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3\} \neq A^c(\mathbf{E}) = \{\mathsf{c}_3\})$. Suppose that the operator starts the system and then after some time, decides to perform a command on it. With the state-value v_1, the operator cannot know for sure whether the system is in the state $\mathbf{A}$ or $\mathbf{E}$.
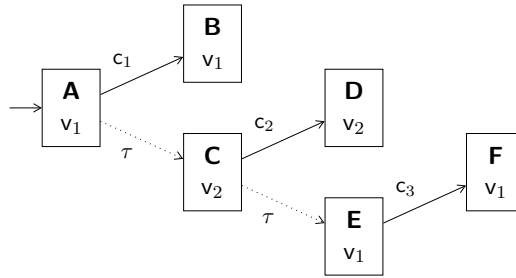
Figure 4.11. The chain of internal transitions causes a non-fc-determinism situation since the operator, who does not track the changes of state-values, cannot decide whether he can perform c_1 or c_3 when he observes the v_1 state-value.

It is important to remember that this is a choice of modelling that has been done. If the operator was able to track the state-value changes, it would indeed be possible for him to distinguish between states $\mathbf{A}$ and $\mathbf{E}$, just by noticing that the state-value has changed from v_1 to v_2 and then again to v_1.

To sum up, two hypotheses are made about enriched models. First of all, the operator does not remember state-values corresponding to spontaneous changes of state in the system. In the example, he cannot track the changes $\mathbf{A} \xrightarrow{\tau} \mathbf{C} \xrightarrow{\tau} \mathbf{E}$. Moreover, the operator only observes the state-value at the moment of performing the action. In the example, he can observe v_2 and perform at the same time the c_2 command when the system is in state $\mathbf{C}$, without falling into the state $\mathbf{E}$ in-between.

4.3.3 Expansion of Enriched Models

Enriched models can be expanded into HMI-LTSs so that the full-control property is preserved. The motivation for such an expansion is to make

it possible to perform the same reasoning and analyses that can be done on HMI-LTSs. The expansion operation transforms state-values into observation actions.

HVS Expansion

The expansion of an HVS is based on the mapping shown on Figure 4.12. Every non-τ transition $s \xrightarrow{\alpha} t$ is expanded so that the state-value v is checked before the occurrence of the action, that is, a new state s^v is added to the expanded model with the sequence of transitions $s \xrightarrow{v} s^v \xrightarrow{\alpha} t$. For τ-transitions, no transformation occurs, they are preserved in the expanded HMI-LTS.

Figure 4.12. Transition mapping for HVS to HMI-LTS translation for non-τ transitions. The transition from the original system model (on the left) induces two transitions in the expanded system model (on the right), where $v = \mathcal{O}(s)$.

Definition 4.11 (HVS expansion). *Given an HVS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O} \rangle$, its expansion, denoted $exp(\mathcal{S})$, is an HMI-LTS $\mathcal{E} = \langle S_E, \mathcal{L}^c, \mathcal{L}_E^o, s_{0_S}, \rightarrow_E \rangle$ where $\mathcal{L}_E^o = \mathcal{L}^o \cup \mathcal{L}^v$ and:*

- $S_E = S_S \cup \{s^v \mid s \in S_S, v = \mathcal{O}(s) \text{ and } \Gamma(s) \neq \emptyset\}$;
- $\rightarrow_E = \{(s, \tau, t) \mid (s, \tau, t) \in \rightarrow_S\}$
  $\cup \{(s, v, s^v), (s^v, \alpha, t) \mid (s, \alpha, t) \in \rightarrow_S \text{ and } v = \mathcal{O}(s)\}$.

By construction, the states of the expanded HMI-LTS can be partitioned into two sets:

- *Observation states* are those from the original HVS and intuitively correspond to the fact that the operator must check the state-value before the occurrence of a visible action on the system. Those states can have outgoing τ-transitions and at most one outgoing transition labelled with a state-value. An observation state s is characterised by $|A^o(s)| = 1$, $A^o(s) \subseteq \mathcal{L}^v$ and $A^c(s) = \emptyset$.

- *Action states* are the states added to the HVS. Those states have outgoing transitions with commands and observations that correspond to those from the original HVS. They do not have any τ-transitions. An action state s is characterised by $A(s) \subseteq \mathcal{L}$.

Transitions outgoing from an observation state lead to an action state, except for τ-transitions that lead to another observation state. And conversely, transitions outgoing from an action state always lead to an observation state.

Figure 4.13 shows the expansion of an HVS example. The two τ-transitions are still there, and all the other transitions have been replaced by two transitions. The expanded model has five observation states and three action states (greyed on the figure).

(a) An HVS example $\mathcal{S}$.   (b) The expansion $exp(\mathcal{S})$.

Figure 4.13. Expansion of an HVS example into an HMI-LTS. Greyed states of the expanded mode are action states and white ones are observation states.

HVM Expansion

HVMs can also be expanded into HMI-LTSs, so as to make enriched traces explicit. The expansion is based on the intuition that the operator must always first check whether the action guard is satisfied before doing any visible action. Figure 4.14 shows the mapping between the transitions from the HVM and the corresponding expanded HMI-LTS. Every transition $s \xrightarrow{[v]\alpha} s'$ is expanded so that the state-value v is checked before performing the α action, that is, a new state s^v is added to the expanded model with the sequence of transitions $s \xrightarrow{v} s^v \xrightarrow{\alpha} s'$.

Figure 4.14. Transition mapping for HVS expansion. The transition from the HVM (on the left) induces two transitions in the expanded HMI-LTS (on the right).

Definition 4.12 (HVM expansion)**.** *Given an HVM $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o,$ $s_{0_H}, \rightarrow_H, \mathcal{L}^v \rangle$, its expansion, denoted $exp(\mathcal{H})$, is an HMI-LTS $\mathcal{E} = \langle S_E, \mathcal{L}^c, \mathcal{L}^o_E, s_{0_H}, \rightarrow_E \rangle$ where $\mathcal{L}^o_E = \mathcal{L}^o \cup \mathcal{L}^v$ and:*

- $S_E = S_H \cup \{s^v \mid s \in S_H, (s, v, \alpha, t) \in \rightarrow_H\}$;

- $\rightarrow_E = \{(s, v, s^v), (s^v, \alpha, t) \mid (s, v, \alpha, t) \in \rightarrow_H\}$.

As it is the case with HVS, two kinds of states are identifiable by construction: observation states and action states. The observation states intuitively correspond to the check of the action guard by the user. They are characterised by $\Gamma(s) \subseteq \mathcal{L}^v$ and their outgoing transitions always lead to an action state. The action states correspond to those from the HVM. They are characterised by $\Gamma(s) \subseteq \mathcal{L}$. Their outgoing transitions are labelled with commands and observations and always lead to an observation state.

Figure 4.15 shows the expansion of an HVM example. Every transition has been replaced by two transitions. The expanded model has four observation states and five action states (greyed on the figure).

(a) An HVM example $\mathcal{H}$. (b) The expansion $exp(\mathcal{H})$.
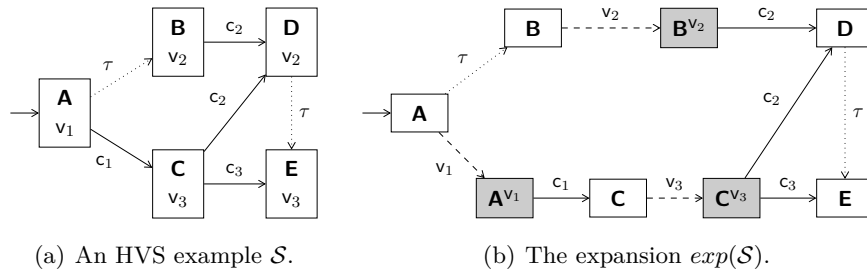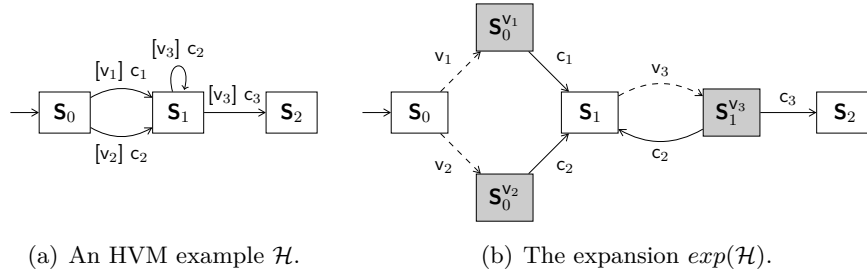
Figure 4.15. Expansion of an HVS example into an HMI-LTS. Greyed states of the expanded mode are action states and white ones are observation states.

Expanded Models and HMI-LTSs Equivalence

Using the expanded models, it is possible to check whether an HVS allows full-control of an HVM, by expanding both models into HMI-LTSs and by checking whether the expanded mental model allows full-control of the expanded system model.

Theorem 4.13. *Given an HVS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^v, \mathcal{O} \rangle$ *and an HVM* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^v \rangle$:

$$\mathcal{H} \, \mathbf{fc} \, \mathcal{S} \iff exp(\mathcal{H}) \, \mathbf{fc} \, exp(\mathcal{S})$$

Proof. Let us suppose that $\mathcal{H} \, \mathbf{fc} \, \mathcal{S}$. It means that for any trace $\sigma = \langle \alpha_1 \cdots \alpha_n \rangle$ of the interaction model $\mathcal{S} \parallel_I \mathcal{H}$, which is such that:

$$(s_{0_S}, s_{0_H}) \xrightarrow{\tau^*} (s'_{0_S}, s_{0_H}) \xrightarrow{\alpha_1} (s_{1_S}, s_{1_H}) \cdots$$
$$(s_{n-1_S}, s_{n-1_H}) \xrightarrow{\tau^*} (s'_{n-1_S}, s_{n-1_H}) \xrightarrow{\alpha_n} (s_{n_S}, s_{n_H}),$$

we have $A^c_v(s_{n_S}) = A^c_v(s_{n_H})$ and $A^o_v(s_{n_S}) \subseteq A^o_v(s_{n_H})$.
By definition of the interaction model, we have that, for $0 \le i < n$:

- $s_{i_S} \xrightarrow{\tau^*} s'_{i_S} \xrightarrow{\alpha_{i+1}} s_{i+1_S}$ belongs to $\mathcal{S}$;

- and $s_{i_H} \xrightarrow{[v]\alpha_{i+1}} s_{i+1_H}$ belong to $\mathcal{H}$ with $v = \mathcal{O}(s'_{i_S})$

which implies, by construction of the expanded models, that we have, for $0 \le i < n$:

- $t_{i_S} \xrightarrow{\tau^*} t'_{i_S} \xrightarrow{v} t'^v_{i_S} \xrightarrow{\alpha_{i+1}} t_{i+1_S}$ belongs to $exp(\mathcal{S})$;

- and $t_{i_H} \xrightarrow{v} t^v_{i_H} \xrightarrow{\alpha_{i+1}} t_{i+1_H}$ belong to $exp(\mathcal{H})$

with $v = \mathcal{O}(s'_{i_S})$ and such that there is a one-to-one relation between s_{i_S} and t_{i_S}, between s'_{i_S} and t'_{i_S} and between s_{i_H} and t_{i_H}.
Let $\mathcal{H}' = exp(\mathcal{H})$, $\mathcal{S}' = exp(\mathcal{S})$ and $\sigma' \in (\mathcal{L}^c \cup \mathcal{L}^o \cup \mathcal{L}^v)^*$, be a trace of the interaction model $\mathcal{H}' \parallel \mathcal{S}'$. Two cases have to be considered:

1. The trace is of the form $\sigma'_\alpha = \langle v_1 \alpha'_1 \cdots v_n \alpha_n \rangle$, which means that there exists $(t_{0_S}, t_{0_H}) \xRightarrow{\sigma'_\alpha} (t_{n_S}, t_{n_H})$ in $\mathcal{H}' \parallel \mathcal{S}'$. By construction, $A(t_{n_S})$ and $A(t_{n_H})$ are both subsets of $\mathcal{L}^v$. Consequently, $A^c(t_{n_S}) = A^c(t_{n_H}) = \{\}$. Moreover, $A^o(t_{n_S}) = \{v \mid (v, \alpha) \in A^o_v(s_{n_S})\}$ and $A^o(t_{n_H}) = \{v \mid (v, \alpha) \in A^o_v(s_{n_H})\}$, and since $A^o_v(s_{n_S}) \subseteq A^o_v(s_{n_H})$ by hypothesis, it implies that $A^o(t_{n_S}) \subseteq A^o(t_{n_H})$.

2. The trace is of the form $\sigma'_v = \langle v_1 \alpha'_1 \cdots v_n \rangle$, which means that there exists $(t_{0_S}, t_{0_H}) \overset{\sigma'_v}{\Longrightarrow} (t'^v_{n-1_S}, t^v_{n-1_H})$ in $\mathcal{H}' \parallel \mathcal{S}'$. By construction, $A^c(t'^v_{n-1_S}) = \Gamma^c_v(s'_{n-1_S})$ and $A^c(t^v_{n-1_H}) = \{\alpha \mid (v, \alpha) \in A^c_v(s_{n-1_H})\}$, which implies $A^c(t'^v_{n-1_S}) = A^c(t^v_{n-1_H})$. A similar reasoning with observations allows us to prove that $A^o(t'^v_{n-1_S}) \subseteq A^o(t^v_{n-1_H})$.

To conclude, it means that $\mathcal{H}' \textbf{ fc } \mathcal{S}'$. The proof of the other way is similar. $\qquad\square$

4.4 Comparisons with Other Relations

There are plenty of relations that have been established in order to compare models in several fields such as model-based engineering and model-based testing. This section reviews the main relations that exist, discusses their potential use for human-machine interaction analysis and compares them to the full-control property. For the remaining of this section, the comparison will be made between a system model $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and a mental model $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, both represented as HMI-LTSs. In order to simplify the presentation, $\mathcal{S}$ and $\mathcal{H}$ can also refer to their underlying LTSs in this section.

Most of the presented relations are used in model-based engineering where a given implementation $Impl$ must be validated regarding a specification $Spec$. Regarding human-machine interaction, $Spec$ can be related to mental model and $Impl$ to system model. In that case, a system model is viewed as an implementation of the mental model. The implementation, namely the system model, is valid regarding a given specification, namely the mental model, if all the operations that are possible on the system are authorised by the specification. The specification acts like an envelope of valid behaviour inside which all the valid implementations lie.

The analysed relations can be split into two categories. The first ones have been defined on classical LTSs and are referred to as being part of the *linear time – branching time spectrum* [vG01]. The relations from the second category are considering models where inputs and outputs play a role [Tre08], and they are therefore closer to the work presented in this thesis.

4.4.1 Trace Preorder

The *trace preorder* [Hoa85] is the simplest relation that can be established between two models. According to that relation, an implementation is said to be valid if any of its traces belongs to the specification. In other words, the set of traces of any implementation must be a subset of the set of traces of the specification.

Definition 4.14 (Trace preorder)**.** *Given two LTSs* $\mathcal{S} = \langle S_S, \mathcal{L}, s_{0_S}, \rightarrow_S \rangle$ *and* $\mathcal{H} = \langle S_H, \mathcal{L}, s_{0_H}, \rightarrow_H \rangle$*, the* trace preorder*, denoted* $\leq_{tr}$*, is defined as follows:*

$$\mathcal{S} \leq_{tr} \mathcal{H} \iff \mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H})$$

The full-control property allows the mental model to contain more behaviour than the one contained in the system model. This is precisely due to the flexibility brought by observations in the definition of full-control. Generally speaking, the set of traces of a full-control mental model is always a superset of the set of traces of the system model for which it allows full-control.

Property 4.15. *Given two HMI-LTSs* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ *and* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$*:*

$$\mathcal{H} \, \mathbf{fc} \, \mathcal{S} \implies \mathcal{S} \leq_{tr} \mathcal{H}$$

Proof. Let $\sigma \in \mathcal{L}^*$ be a trace that is supposed to belong to both $\mathbf{Tr}(\mathcal{S})$ and $\mathbf{Tr}(\mathcal{H})$. By definition of full-control, it means that there would exist two executions $s_{0_S} \overset{\sigma}{\Longrightarrow} s_S$ and $s_{0_H} \overset{\sigma}{\longrightarrow} s_H$ such that $A^c(s_S) = A^c(s_H)$ and $A^o(s_S) \subseteq A^o(s_H)$. Consequently, any extension σa, with $a \in \mathcal{L}$, that belongs to $\mathbf{Tr}(\mathcal{S})$ would also belongs to $\mathbf{Tr}(\mathcal{H})$. By induction, since the empty trace belongs to both $\mathbf{Tr}(\mathcal{S})$ and $\mathbf{Tr}(\mathcal{H})$, it means that $\mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H})$ and therefore $\mathcal{S} \leq_{tr} \mathcal{H}$. $\qquad\square$

A contrario, trace preorder does not imply full-control as illustrated by the example of Figure 4.16. In this example, the set of traces of $\mathcal{S}$ is contained in the set of traces of $\mathcal{H}$ (that is, $\mathcal{S} \leq_{tr} \mathcal{H}$), but it is not the case that $\mathcal{H}$ allows full-control of $\mathcal{S}$. Indeed, the two models do not have the same set of possible commands after the empty trace, $\mathcal{H}$ has an additional c_2 command.
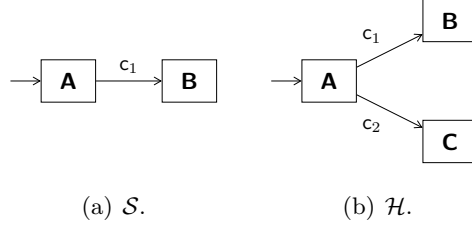
(a) $\mathcal{S}$. (b) $\mathcal{H}$.

Figure 4.16. Trace preorder between a mental and a system model does not imply full-control property between both models: $\mathcal{S} \leq_{tr} \mathcal{H}$ and $\neg(\mathcal{H} \,\mathbf{fc}\, \mathcal{S})$.

The additional traces that are present in the mental model all follow the same pattern: $\sigma o \sigma'$. They have an observation in them such that σ is also in the system model, but not σo. Such traces can be in a full-control mental model since, as already detailed, the full-control mental model can have observations that will never occur on the system.

Definition 4.16 (Observation-Closure)**.** *Given an HMI-LTS* $\mathcal{S} = \langle S_S,$ $\mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ *and a set of traces* $T \in \mathcal{L}^*$, *the* observation-closure *of* T, *denoted* $Cl^o(T)$, *is defined as:*

$$Cl^o(T) = T \cup \{\sigma o \sigma' \mid \sigma o \notin T, o \in \mathcal{L}^o, \sigma' \in \mathcal{L}^*\}$$

Property 4.17. *Given an fc-deterministic HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o,$ $s_{0_S}, \rightarrow_S \rangle$ *and a deterministic non-divergent HMI-LTS* $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o,$ $s_{0_H}, \rightarrow_H \rangle$:

$$\mathcal{H} \,\mathbf{fc}\, \mathcal{S} \iff \mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H}) \text{ and } \mathbf{Tr}(\mathcal{H}) \subseteq Cl^o(\mathbf{Tr}(\mathcal{S}))$$

Proof. Let first suppose that $\mathcal{H} \,\mathbf{fc}\, \mathcal{S}$; thus, by Property 4.15, it means that $\mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H})$. Let $\sigma \in \mathcal{L}^*$ be a trace that belongs to both $\mathbf{Tr}(\mathcal{S})$ and $\mathbf{Tr}(\mathcal{H})$. By definition of full-control, it means that there would exist two executions $s_{0_S} \overset{\sigma}{\Longrightarrow} s_S$ and $s_{0_H} \overset{\sigma}{\longrightarrow} s_H$ such that $A^c(s_S) = A^c(s_H)$ and $A^o(s_S) \subseteq A^o(s_H)$. Consequently, any extension σc, with $c \in \mathcal{L}^c$, that belongs to $\mathbf{Tr}(\mathcal{H})$ also belongs to $\mathbf{Tr}(\mathcal{S})$. Extensions σo, with $o \in \mathcal{L}^o$, that belongs to $\mathbf{Tr}(\mathcal{H})$ do not necessarily belong to $\mathbf{Tr}(\mathcal{S})$, but if it is not the case, it does belong to $Cl^o(\mathbf{Tr}(\mathcal{S}))$, by definition of the observation-closure. Therefore, by induction, since the empty trace belongs to both $\mathbf{Tr}(\mathcal{H})$ and $\mathbf{Tr}(\mathcal{S})$, $\mathbf{Tr}(\mathcal{H}) \subseteq Cl^o(\mathbf{Tr}(\mathcal{S}))$.

Let this time suppose that $\mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H})$ and $\mathbf{Tr}(\mathcal{H}) \subseteq Cl^o(\mathbf{Tr}(\mathcal{S}))$. Let $\sigma \in \mathcal{L}^*$ be a trace that belongs to both $\mathbf{Tr}(\mathcal{S})$ and $\mathbf{Tr}(\mathcal{H})$, which means that there would exist two executions $s_{0_S} \overset{\sigma}{\Longrightarrow} s_S$ and $s_{0_H} \overset{\sigma}{\longrightarrow} s_H$. Let the extension σa, with $a \in \mathcal{L}$, belong to $\mathbf{Tr}(\mathcal{S})$. By hypothesis, it also belongs to $\mathbf{Tr}(\mathcal{H})$. Consequently, $A(s_S) \subseteq A(s_H)$. Let now the extension σa, with $a \in \mathcal{L}$, belong to $\mathbf{Tr}(\mathcal{H})$. By hypothesis, it belongs to $Cl^o(\mathbf{Tr}(\mathcal{S}))$, which means that either it belongs to $\mathbf{Tr}(\mathcal{S})$, or not, in which case a is an observation. Consequently, $A^c(s_H) \subseteq A^c(s_S)$. Therefore, by induction, since the empty trace belongs to both $\mathbf{Tr}(\mathcal{H})$ and $\mathbf{Tr}(\mathcal{S})$, $A^c(s_S) = A^c(s_H)$ and $A^o(s_S) \subseteq A^o(s_H)$, that is, $\mathcal{H} \mathbf{\,fc\,} \mathcal{S}$. $\qquad\square$

4.4.2 Testing Preorder

The *testing preorder* [DH84] adds a constraint to the trace preorder. It takes into account the possibility for the models to deadlock. A state of an LTS is said to *refuse* an action if that action is not possible in that state. By extension, a state of an LTS is said to refuse a set of actions if it refuses every action of the set. The *refusal sets* of a state of an LTS are all the sets of actions that are refused by the state.

Definition 4.18 (Refusal sets). *Given an LTS $\langle S, \mathcal{L}, s_0, \to \rangle$, a state $s \in S$* refuses *an action $\alpha \in \mathcal{L}$ if and only if $\alpha \notin A(s)$. A state $s \in S$* refuses *a set of actions $X \subseteq \mathcal{L}$, which is denoted $s \mathbf{\,refuses\,} X$, if and only if s refuses all the actions of X. The* refusal set *of a state $s \in S$ for a given trace $\sigma \in \mathcal{L}^*$, denoted $\mathbf{Ref}(s, \sigma)$, is the set of refused sets of the states $s' \in S$ that can be reached after the trace σ:*

$$\mathbf{Ref}(s, \sigma) = \{ X \subseteq \mathcal{L} \mid \exists s' \in (s \mathbf{\,after\,} \sigma) : s' \mathbf{\,refuses\,} X \}$$

For an implementation to be valid according to testing preorder, it cannot refuse an action that is allowed by the specification, for any execution trace. In other words, the refusal sets in the implementation must be subsets of the refusal sets on the specification, for any sequence of actions.

Definition 4.19 (Testing preorder). *Given two LTSs $\mathcal{S} = \langle S_S, \mathcal{L}, s_{0_S}, \rightarrow_S \rangle$ and $\mathcal{H} = \langle S_H, \mathcal{L}, s_{0_H}, \rightarrow_H \rangle$, the testing preorder, denoted $\leq_{te}$, is defined as follow:*

$$
\begin{aligned}
\mathcal{S} \leq_{te} \mathcal{H} \iff & \forall \sigma \in \mathcal{L}^* : \mathbf{Ref}(s_{0_S}, \sigma) \subseteq \mathbf{Ref}(s_{0_H}, \sigma) \\
\iff & \forall \sigma \in \mathcal{L}^* : \forall X \subseteq \mathcal{L} : s_{0_S} \textbf{ after } \sigma \textbf{ refuses } X \\
& \qquad\qquad\qquad \implies s_{0_H} \textbf{ after } \sigma \textbf{ refuses } X
\end{aligned}
$$

This relation could be useful to compare a mental model against a system model. The testing preorder requires that the system model cannot refuse an action that is possible according to the mental model. In an HMI perspective, it would mean that at any time during the interaction, there can be more actions in the system model than those which are present in the mental model. If the action is a command, it would mean that the operator does not need to know exactly all the possible ones at any time during the interaction, which does not correspond to what is captured by full-control, but is a sensible alternative. The situation is different if the action is an observation since they are controlled by the machine. It would mean that the user may observe things that would just be meaningless to him and that will get him surprised.

The full-control property and the testing preorder are not directly related at all, as shown by the two following examples referring to Figure 4.17:

- $\mathcal{H}_1 \textbf{ fc } \mathcal{S}$ and $\neg(\mathcal{S} \leq_{te} \mathcal{H}_1)$

  The mental model $\mathcal{H}_1$ does allow full-control of the system $\mathcal{S}$, the additional behaviour that is present in the mental model is related to the o_1 observation, which is not an issue with respect to the full-control property. However it is not the case that $\mathcal{S} \leq_{te} \mathcal{H}_1$, since for the empty trace: $\mathbf{Ref}(s_{0_S}, \varepsilon) = \{\emptyset, \{o_1\}\} \not\subseteq \mathbf{Ref}(s_{0_{H_1}}, \varepsilon) = \{\emptyset\}$.

- $\mathcal{S} \leq_{te} \mathcal{H}_2$ and $\neg(\mathcal{H}_2 \textbf{ fc } \mathcal{S})$

  In this second example, $\mathcal{S} \leq_{te} \mathcal{H}_2$. Indeed:

  - $\mathbf{Ref}(s_{0_S}, \varepsilon) = \{\emptyset\} \subseteq \mathbf{Ref}(s_{0_{H_2}}, \varepsilon) = \{\emptyset\}$;
  - $\mathbf{Ref}(s_{0_S}, c_1) = \{\emptyset\} \subseteq \mathbf{Ref}(s_{0_{H_2}}, c_1) = \{\emptyset\}$;
  - and $\mathbf{Ref}(s_{0_S}, c_1^*) = \emptyset \subseteq \mathbf{Ref}(s_{0_{H_2}}, c_1^*) = \{\emptyset\}$.

But $\mathcal{H}_2$ does not allow full-control of $\mathcal{S}$, since after the sequence α, the system and mental model reach two states with different sets of possible commands: $A^c(\mathsf{B}_S) = \emptyset \neq A^c(\mathsf{A}_{H_2}) = \{\mathsf{c}_1\}$.

(a) $\mathcal{S}$.   (b) $\mathcal{H}_1$ (c) $\mathcal{H}_2$.
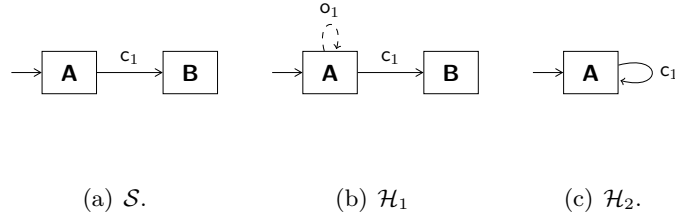
Figure 4.17.   Full-control property and testing preorder are not equivalent: $\neg(\mathcal{H}_1 \mathbf{\,fc\,} \mathcal{S} \implies \mathcal{S} \leq_{te} \mathcal{H}_1)$ and $\neg(\mathcal{S} \leq_{te} \mathcal{H}_2 \implies \mathcal{H}_2 \mathbf{\,fc\,} \mathcal{S})$.

4.4.3 Conformance

The testing preorder requires that the implementation agrees on deadlocks with the specification for any trace. The *conformance relation* [Bri88] is the same than the testing preorder, except that the condition on deadlocks is only to be verified on the traces of the specification.

Definition 4.20 (Conformance relation)**.** *Given two LTSs* $\mathcal{S} = \langle S_S, \mathcal{L},$ $s_{0_S}, \rightarrow_S \rangle$ *and* $\mathcal{H} = \langle S_H, \mathcal{L}, s_{0_H}, \rightarrow_H \rangle$, *the* conformance relation*, denoted* **conf***, is defined as follow:*

$$\mathcal{S} \mathbf{\,conf\,} \mathcal{H} \quad \iff \quad \forall \sigma \in \mathbf{Tr}(\mathcal{H}) : \mathbf{Ref}(s_{0_S}, \sigma) \subseteq \mathbf{Ref}(s_{0_H}, \sigma)$$

The intuition is the same than the one of testing preorder, except that only those traces belonging to the specification are considered. In an HMI perspective, it would mean that only for the execution that the human is likely to perform, the system cannot refuse an action that is present in the mental model. Said in the other way, it would mean that no matter what the operator executes, it is always accepted by the system.

Input-Output Conformance

The conformance relation has been extended in order to take into account input and output actions. The *input-output conformance* relation has been defined on IOTS (Definition 3.17 on page 84). One particularity of the relation is that it takes into account *quiescent states*, which are states from which there are no outputs.

Definition 4.21 (Quiescent state and traces)**.** *Given an IOTS $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$, a state $s \in S$ is a* quiescent state, *which is denoted $\delta(s)$, if there is no enabled output or outgoing τ-transition from that state, that is, $\delta(s) \iff \forall \alpha \in \mathcal{L}^o \cup \{\tau\} : s \not\xrightarrow{\alpha}$.*

The set of quiescent traces *of $\mathcal{M}$ is the set of traces that lead to a quiescent state. It is defined and denoted as $\mathbf{QTr}(\mathcal{M}) = \{\sigma \in \mathcal{L}^* \mid \exists s' \in (s_0 \, \mathbf{after} \, \sigma) : \delta(s')\}$.*

The hypothesis is that making no observation is in some sense an observation about the system. Since quiescence is a kind of observation, it is denoted as the δ action (with $\delta \notin \mathcal{L} \cup \{\tau\}$). Given that additional observation, it is possible to explicitly identity quiescent states by defining *suspension traces*.

Definition 4.22 (Suspension traces)**.** *Given an IOTS $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$, it is extended to the IOTS $\mathcal{S}_\delta = \langle S, \mathcal{L}^c, \mathcal{L}^o \cup \{\delta\}, s_0, \rightarrow \cup \rightarrow_\delta \rangle$ where $\rightarrow_\delta = \{s \xrightarrow{\delta} s \mid s \in S \text{ and } \delta(s)\}$.*

The set of suspension traces *of $\mathcal{S}$ is the set of traces of $\mathcal{S}_\delta$. It is defined and denoted as $\mathbf{STr}(\mathcal{M}) = \{\sigma \in \mathcal{L}_\delta^* \mid s_0 \xRightarrow{\sigma}\}$.*

For an implementation to be input-output conforming to a specification, it cannot produce an output that is not foreseen according to the specification.

Definition 4.23 (Input-output conformance relation)**.** *Given two IOTSs $Impl = \langle S_I, \mathcal{L}^c, \mathcal{L}^o, s_{0_I}, \rightarrow_I \rangle$ and $Spec = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, the* input-output conformance relation, *denoted* **ioco***, is defined as follow:*

$$Impl \, \mathbf{ioco} \, Spec \iff \forall \sigma \in \mathbf{STr}(Spec) :$$
$$out(Impl \, \mathbf{after} \, \sigma) \subseteq out(Spec \, \mathbf{after} \, \sigma)$$

$$where \, out(S) = \bigcup_{s \in S} \Big(\Gamma^o(s) \cup \{\delta \mid \delta(s)\} \Big).$$

The **ioco** relation does not impose anything on input actions, which makes it different from the full-control property. Indeed, according to the **ioco** relation, commands may be available in the implementation even if not present in the specification. The same results that those obtained with the testing preorder are still valid for **ioco** (Figure 4.17):

- $Impl$ **ioco** $Spec$, but $Impl$ does not allow full-control of Spec, since they do not have the same set of possible commands for the empty trace $\sigma = \varepsilon$: $A^c(A_I) = \{\alpha\} \neq A^c(A_S) = \emptyset$.

- $\mathcal{H}$ allows full-control of $Impl$, but $lts(\mathcal{H})$ **io̸co** $Impl$ since they do not agree on outputs for the empty trace $\sigma = \varepsilon$: $out(A_{\mathcal{H}}) = \{\beta\} \nsubseteq out(A_I) = \{\delta\}$.

Chapter 5

Generating Full-control Conceptual Models

This chapter proposes three approaches to automatically generate minimal full-control conceptual models. Section 5.1 states precisely the minimal full-control conceptual model generation problem and discusses characteristics of its solution. Section 5.2 presents Three-Valued Deterministic Automata (3DFAs) and a trace characterisation of the full-control property based on 3DFAs. The three next sections each presents an algorithm to solve the generation problem. Firstly, Section 5.3 presents a direct approach based on the 3DFA characterisation. Then, Section 5.4 presents an algorithm based on a reduction approach inspired by the work of Degani et al. [HD07] where states of the system model are merged together to build the full-control conceptual model. Finally, Section 5.5 follows with a third algorithm based on an active learning approach where the full-control conceptual model is built incrementally. Last but not least, Section 5.6 compares the proposed approaches and identifies the advantages and disadvantages of each approach.

5.1 The Minimal Full-control Conceptual Model Generation Problem

As introduced in the general overview in Chapter 2.1.2 on page 14, a distinction is made between the notions of mental model and conceptual model. Whereas the mental model corresponds to the model that the operator has in his mind and that can evolve with time, conceptual model is used in this thesis to refer to a model that has been generated based on the system model. The conceptual model can be seen as a kind of perfect mental model that if used by the operator, guarantees some

properties. In this work, the focus is on conceptual models that allow full-control of the system model.

Generating a conceptual model from a system model can serve various purposes, including the analysis of properties of the system such as its complexity or usability, and the generation of training materials. Those different uses of a conceptual model are presented in details in Chapter 6. The *minimal full-control conceptual model generation problem* consists in finding, given a system model, a conceptual model that allows full-control of it and which is minimal, in terms of the number of states.

Definition 5.1 (The minimal full-control conceptual model generation problem (MFCCG))**.** *Given an HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ *representing a system model, the* minimal full-control conceptual model generation problem *consists in finding an HMI-LTS* $\mathcal{A} = \langle S_A, \mathcal{L}^c, \mathcal{L}^o, s_{0_A}, \rightarrow_A \rangle$ *representing a conceptual model such that:*

1. *$\mathcal{A}\,\mathbf{fc}\,\mathcal{S}$*

2. *$\forall \mathcal{A}' = \langle S'_A, \mathcal{L}^c, \mathcal{L}^o, s'_{0_A}, \rightarrow'_A \rangle : \mathcal{A}'\,\mathbf{fc}\,\mathcal{S} \implies |S_A| \leq |S'_A|$*

The intuitive idea to generate a full-control conceptual model is to identify states of the system model whose behaviours are similar. Similar behaviours can be seen as equivalent from the point of view of the user. Differences in the behaviours can be ignored by the user without affecting his controllability of the system. Those similar states can be merged together to reduce the system model and to get a simplified one that can serve as a conceptual model.

This idea is pretty much the same as the one lying behind the operationally deterministic LTS determinisation algorithm presented in [HV06] and illustrated by Figure 5.1. The system model of Figure 5.1(a) is not (structurally) deterministic, but is operationally deterministic according to Definition 3.8 on page 66. Indeed, states **A** and **B** both only have one enabled action and triggering it leads to states with the same behaviour. The model of Figure 5.1(b) is a reduced version of that system model, where states **A** and **B** have been merged together. It is not the most reduced model that can be obtained since states **C** and **D** also exhibit the same behaviour; they and can thus also be merged together.

The approach of [HV06] cannot be directly applied for the MFCCG problem. The fact that the mental model can have more observations than

(a) A system model $\mathcal{S}$.   (b) A reduced version of $\mathcal{S}$.
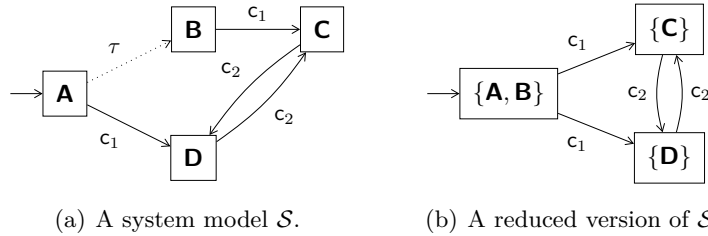
Figure 5.1. A system model that is operationally deterministic and a corresponding reduced model where states **A** and **B** have been merged together. The reduced model is not the minimal once and can thus be further reduced.

those that can actually occur on the system, according to the full-control property, is not considered by their approach. Figure 5.2 illustrates that issue. The system model of Figure 5.2(a) is not operationally deterministic since the empty trace can lead to several states with different behaviours. The system cannot be reduced with the approach of [HV06] but, as shown on Figure 5.2(b), there does exist a reduced full-control conceptual model for it.

(a) A system model $\mathcal{S}$.   (b) The minimal full-control conceptual model for $\mathcal{S}$.
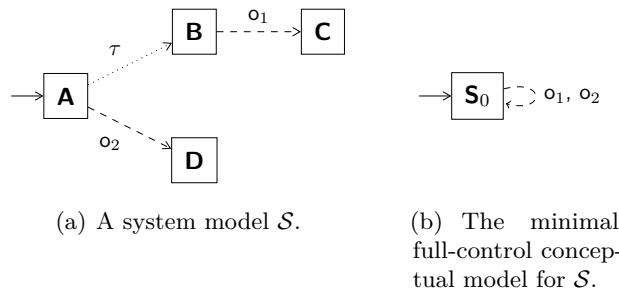
Figure 5.2. A system model that is not operationally deterministic and cannot be reduced with the approach of [HV06], but for which there exists a reduced full-control conceptual model.

5.1.1 Unicity

Given a system model, there does not always exist only one single possible minimal full-control conceptual model for it. Figure 5.3 shows an example of a system model for which there exists two different minimal full-control

conceptual models. State **C** is fc-similar to both states **B** and **D**. However, states **B** and **D** are not fc-similar, because they do not have the same sets of possible commands after the execution of $\langle o_1 \rangle$ trace. This is a consequence of the fact that the fc-similarity relation is not transitive. The first possible full-control minimal conceptual model (on the left) is obtained by merging together states **B** and **C**, whereas states **C** and **D** have been put together for the second one (on the right).
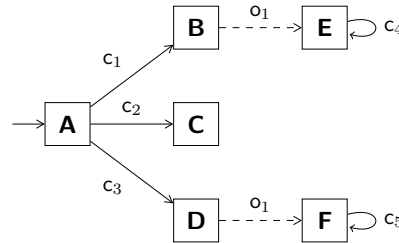
(a) A system model $\mathcal{S}$.
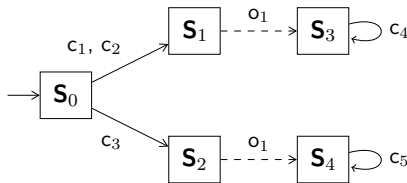
(b) A possible minimal full-control conceptual model for $\mathcal{S}$.
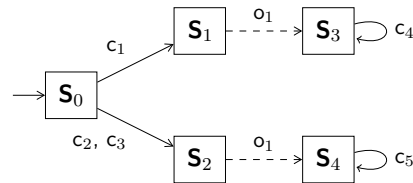
(c) Another possible minimal full-control conceptual model for $\mathcal{S}$.

Figure 5.3. Example of a system model for which there exists two possible different minimal full-control conceptual models.

It is possible to find all the minimal full-control conceptual models, by enumerating them all. The time taken to generate all of them is of course dramatically increased. The MFCCG problem does not require to find all the possible solutions, algorithms proposed in this thesis only produce one conceptual model as a result. It is however possible to represent all the possible solutions of the MFCCG problem by a single model, using three-valued deterministic finite automata (3DFA). It is only a single representation of the solutions, but if a particular solution has to be provided, it would requires time to compute one from the 3DFA. Those considerations are discussed further in Section 5.2.

5.1.2 Generation Algorithms

Three different algorithms to automatically generate a minimal full-control conceptual model from a given system model are proposed in this thesis. Figure 5.4 shows a global view of the three proposed algorithms, that all start with a system model $\mathcal{S}$ and generate a minimal full-control conceptual model $\mathcal{H}$. The first one of the top is the 3DFA-based, the middle one is the reduction-based and the bottom one is the learning-based algorithm. Several operations are used by more than one algorithm, including DFA-minimisation and $\tau^*a\tau^*$-completion.
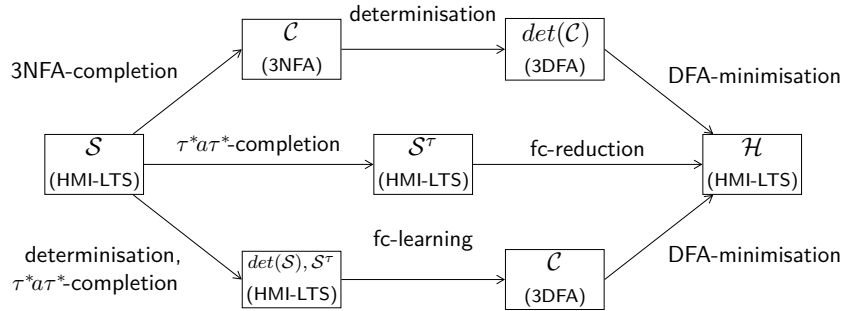
Figure 5.4. Steps of the three generation algorithms proposed in this thesis.

None of the algorithms makes the hypothesis that the input system model is fc-deterministic. They can all detect whether there is an fc-determinism issue, while processing the system model. However, fc-determinism can be checked before running the generation algorithms, in which case some of three proposed algorithms can be simplified. This chapter describes the general version of the algorithms, without assuming fc-deterministic system models.

5.2 Three-Valued Deterministic Finite Automata

As introduced in Chapter 3 on page 59, labelled transition systems (LTS) are very close to *deterministic finite automata* (DFA). The difference is that DFAs are used to represent *languages*, that is, sets of finite words over a given alphabet.

Definition 5.2 (Deterministic Finite Automaton (DFA)). *A Deterministic Finite Automaton (DFA) $\mathcal{A}$ is a tuple $\langle \Sigma, S, s_0, \delta, Acc \rangle$ where Σ is a finite set of labels (the alphabet), S is a finite set of states, $s_0 \in S$ is the initial state, $\delta : S \times \Sigma \to S$ is the transition function and $Acc \subseteq S$ is the set of accepting states.*

Figure 5.5(a) shows a graphical representation of a DFA example. That example represents the language consisting of words made of a finite repetition of $\langle ab \rangle$ (including the empty word). The words of the language are those satisfying the following regular expression: $(ab)^*$.

Not all the transitions are shown on the figure to keep is readable. Indeed, every state of a DFA must have exactly one transition for every label of the alphabet. The missing transitions are all leading to one rejecting state from which everything is rejected. Figure 5.5(b) shows the complete version of the DFA example.

(a) DFA example (simplified version).   (b) DFA example (complete version).

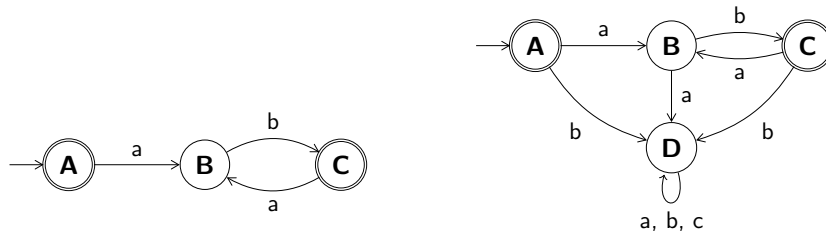**Figure 5.5.** Graphical representation of a DFA example. States with double lines are accepting states and those with simple lines are rejecting states.

For DFAs, the notion of words is similar to the notion of traces for LTSs. A word is accepted by a DFA if executing it leads to an accepting state, otherwise the word is rejected. The *language* recognised by a given DFA is the set of words that are accepted by it.

Definition 5.3 (Language recognised by a DFA). *Given a DFA $\mathcal{A} = \langle \Sigma, S, s_0, \delta, Acc \rangle$, the language that it recognises, denoted $L(\mathcal{A})$, is defined as $L(\mathcal{A}) = \{\sigma \in \Sigma^* \mid s \in Acc, where\ s_0 \xrightarrow{\sigma} s\}$.*

In the sequel, the set of words that are accepted by a DFA $\mathcal{A}$ is denoted L_{Acc} and corresponds to the recognised language $L(\mathcal{A})$. The set of words that are rejected is denoted L_{Rej} and corresponds to $\Sigma^* \setminus L(\mathcal{A})$.

DFAs can be extended so that their sets of states are partitioned into three different disjoints sets. *Three-valued deterministic finite automaton* (3DFA) [CFC$^+$09] is a variant of classical DFA where each state is either accepting, rejecting or can also be classified as a *don't care* state. That third category is used to identify states that can act either as accepting or rejecting state. Intuitively, it means that the same don't care state can sometimes accept and sometimes reject words.

Definition 5.4 (Three-Valued Deterministic Finite Automaton (3DFA))**.**
A Three-Valued Deterministic Finite Automaton *(3DFA) $\mathcal{C}$ is a tuple $\langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$ where Σ is a finite set of labels (the alphabet), S is a finite set of states, $s_0 \in S$ is the initial state and $\delta : S \times \Sigma \to S$ is the transition function. The set of states S is partitioned into three disjoint sets: Acc is the set of accepting states, Rej is the set of rejecting states and finally DC is the set of don't care states.*

Figure 5.6 shows a graphical representation of a 3DFA example. The example is an extension of the DFA example of Figure 5.5(a). With that 3DFA example, it is possible to have a finite sequence of c, between repetitions of the $\langle$ab$\rangle$ sequence, to be accepted or rejected. Again, not all the transitions are shown on the figure to keep it readable. Just as it is done with DFAs, missing transitions are leading to a state from which everything is rejected. For example, the words $\langle$ab$\rangle$ and $\langle$abcccab$\rangle$ are accepted; the words $\langle$a$\rangle$ and $\langle$abccca$\rangle$ are rejected; and finally the words $\langle$abc$\rangle$ and $\langle$abccc$\rangle$ are don't care words; they can therefore either be accepted or rejected.
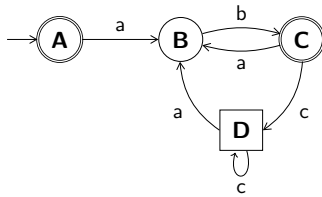

Figure 5.6. Graphical representation of a 3DFA example. States depicted as squares are don't care states.

5.2.1 Consistent DFA

A 3DFA encodes a set of DFAs. A DFA is said to be *consistent* with a 3DFA if the words that it accepts and rejects are also accepted and rejected by the 3DFA, or are don't care words. Said in the other way, it means that all the words that are accepted (resp. rejected) by the 3DFA must also be accepted (resp. rejected) by the DFA.

Definition 5.5 (Consistent DFA). *Given a 3DFA $\mathcal{C} = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$, a DFA $\mathcal{A} = \langle \Sigma, S_A, s_{0_A}, \delta_A, Acc_A \rangle$ is said to be* consistent *with $\mathcal{C}$ if and only if for any $\sigma \in \Sigma^*$:*

- $\sigma \in L_{Acc} \implies \sigma \in L_{Acc_A}$

- $\sigma \in L_{Rej} \implies \sigma \in L_{Rej_A}$

Figure 5.7 shows a 3DFA example with a DFA example that is consistent with it. This example illustrates a situation where a don't care state can act both as an accepting or a rejecting state, depending on the word that ends on it. For example, the $\langle \mathsf{a} \rangle$ don't care word is accepted by the DFA, but the $\langle \mathsf{a}, \mathsf{b}, \mathsf{a} \rangle$ word don't care is rejected by the DFA.

(a) A 3DFA example $\mathcal{C}$.   (b) A DFA example that is consistent with $\mathcal{C}$.
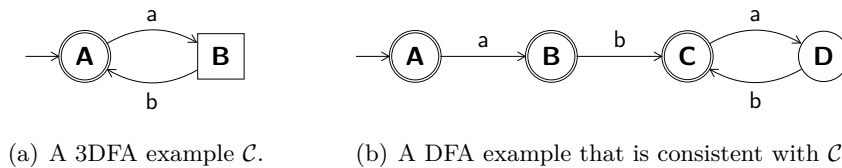
Figure 5.7. A 3DFA example with one possible consistent DFA for it.

A 3DFA $\mathcal{C}$ characterises a a set of languages, namely the languages recognised by the DFAs consistent with $\mathcal{C}$. Among all the consistent DFAs, two are of particular interest as they are acting as lower and upper bounds for the languages characterised by the 3DFA. The $\mathcal{C}^+$ DFA is the one obtained by considering the don't care states as accepting states. Similarly, the $\mathcal{C}^-$ DFA is obtained by considering the don't care states as rejecting states. The language of any DFA consistent with $\mathcal{C}$ lies between the languages recognised by $\mathcal{C}^-$ and $\mathcal{C}^+$.

Property 5.6 (3DFA language characterisation). *Given a 3DFA $\mathcal{C} = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$, and the two DFAs $\mathcal{C}^+ = \langle \Sigma, S, s_0, \delta, Acc \cup DC \rangle$ and $\mathcal{C}^- = \langle \Sigma, S, s_0, \delta, Acc \rangle$, a DFA $\mathcal{A}$ is consistent with $\mathcal{C}$ if and only if:*

$$L(\mathcal{C}^-) \subseteq L(\mathcal{A}) \subseteq L(\mathcal{C}^+)$$

Figure 5.8 illustrates the *language characterisation* of 3DFAs. A given 3DFA partitions the set of all possible words over its alphabet into three disjoint subsets. Any DFA that is consistent with the 3DFA is a subset of Σ^* that must contain completely L_{Acc} and any subset of L_{DC}. The dashed line on Figure 5.8 delimitates one possible consistent DFA.
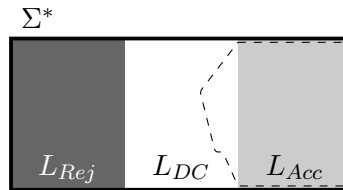
Figure 5.8. A 3DFA partitions the set of words over its alphabet into three disjoint subsets. Any DFA that is consistent with the DFA accepts all the words of L_{Acc}, and can accept any subset of the words of L_{DC}. The set marked out with a dashed line represents one possible consistent DFA.

5.2.2 DFA-minimisation

The *DFA-minimisation problem* consists in finding a DFA that is consistent with the 3DFA, and which is minimal in terms of the number of states. One possible algorithm to solve that problem is the one used by Degani et al. [HD07] and which originally comes from Paull and Unger [PU59]. The idea of the algorithm is to compute sets of states that are compatible, and then to merge compatible states together in order to get a reduced DFA. The particularity is that states from the *DC* set can be compatible with both *Acc* states or *Rej* states, although states from *Acc* and *Rej* cannot be compatible.

Compatible Pairs

The first step of the algorithm is to find the pairs of states that are compatible. A practical way to find those pairs of *compatible states* is to

build a so-called *implication table* as shown on Figure 5.9(b). The table encodes pairwise state compatibilities for the 3DFA of Figure 5.9(a). The cells of the table are either compatible (empty cell), incompatible (crossed cell) or are marked with a set of pairs of states.

The implication table is initialised by marking all the cells (X, Y) where one state is in *Acc* and the other one in *Rej* as incompatible. Then, for all the other pairs of states (X, Y), the cell is marked with all the pairs of states (X', Y') such that there exists a coupled transition $(X, Y) \xrightarrow{\alpha} (X', Y')$ that is not a loop and such that $X' \neq Y'$.
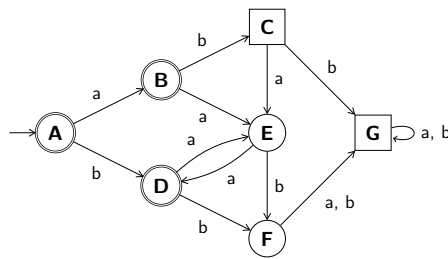
(a) A 3DFA example $\mathcal{C}$.   (b) The initial implication table for $\mathcal{C}$.

Figure 5.9. A 3DFA example with the initial implication table containing pairwise compatibilities that are used to compute the sets of compatibles.

The implication table then goes through several iterations until it reaches a fix point. At each step, each cell containing some pairs of states is examined. If there is any pair leading to an incompatible pair, the cell becomes incompatible. If all the pairs are leading to compatible pairs, the cell becomes compatible. In any other case, the cell remains to be examined for the next iteration. Figure 5.10 shows the second and third iterations that lead to the fix point. Algorithm 6 in Appendix C computes the set of pairs of compatible states.

Maximal Compatibles

Once the pairwise compatibilities have been computed, the next step of the DFA-minimisation consists in finding the sets of compatible states, simply called the *compatibles*. A compatible is a set of states that are
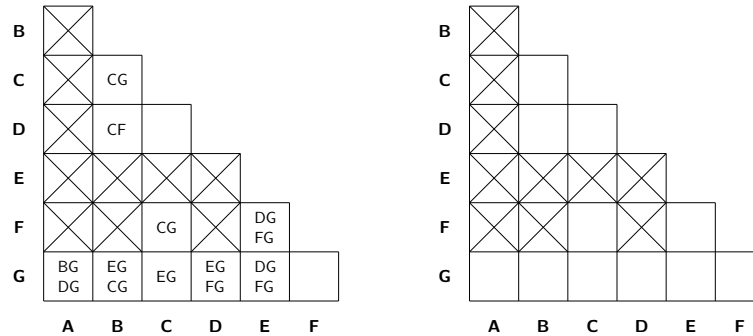
(a) The second iteration for the implication table. Cells leading to (**B**, **E**) or (**D**, **E**) are marked incompatible, the cell leading to (**F**, **G**) becomes compatible and the other not compatible cells are still to be decided.

(b) The third iteration does not change anything to the implication table, which make it stable. All the cells that were still to be decided becomes compatible.

Figure 5.10. The two iterations leading the implication table to a fix point, for the example of Figure 5.9.

all compatible to each other. The idea of that step is to maintain a list of sets that are supposed to be compatibles. Running through the implication table from the left to the right, and examining it column by column, provides information about how the list can be updated so as to finally obtain the list of compatibles.

The idea is to start with a list that only contains one set with all the states of the 3DFA. For each examined column j of the implication table, all the sets of the list containing j are considered. Each of those sets is replaced by two sets: the original set without j and the original set from which the states that are incompatibles in the examined column are removed. After each iteration, the sets of states that are subset of another set from the list are eliminated. Algorithm 7 on Appendix C computes the set of compatibles.

The last step of the DFA-minimisation consists in finding the minimal closed set of compatibles. That set must cover entirely the states of the 3DFA and must also be closed, which intuitively means that all the transitions going from any state of one compatible must lead to one another unique compatible for a given action. The solution may not be unique, and in order to get the one with the minimal number of compatibles, all the possible solutions have to be enumerated.

The successive iterations to find the compatibles for the example of Figure 5.9 are:

1. $\{\{\textbf{BCDEFG}\}, \{\textbf{AG}\}\}$

2. $\{\{\textbf{CDEFG}\}, \{\textbf{BCDG}\}, \{\textbf{AG}\}\}$

3. $\{\{\textbf{DEFG}\}, \{\textbf{CDFG}\}, \{\textbf{BCDG}\}, \{\textbf{AG}\}\}$

4. $\{\{\textbf{EFG}\}, \{\textbf{CFG}\}, \{\textbf{BCDG}\}, \{\textbf{AG}\}\}$

5. $\{\{\textbf{EFG}\}, \{\textbf{CFG}\}, \{\textbf{BCDG}\}, \{\textbf{AG}\}\}$

There is thus a total of four compatibles. One possible solution to cover all the states of the 3DFA is to chose the three following compatibles: $\{\textbf{AG}\}$, $\{\textbf{BCDG}\}$ and $\{\textbf{EFG}\}$. That choice is not closed, since for example, action b outgoing from $\{\textbf{BCDG}\}$ can either lead to $\{\textbf{EFG}\}$ or loop on $\{\textbf{BCDG}\}$. Therefore, the minimal closed set of compatibles just contains all the four compatibles. The corresponding minimal consistent DFA is shown on Figure 5.11.
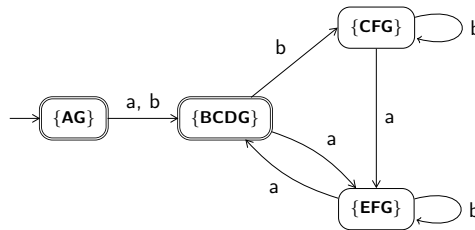
Figure 5.11. Minimal consistent DFA for the 3DFA example of Figure 5.9(a).

5.2.3 Trace Characterisation of Full-control Property

The set of conceptual models that allow full-control of a given system model can be represented as a single 3DFA. Property 4.17 proposed in Chapter 4 on page 117 relates the full-control property to a relation between the sets of traces of the system and the conceptual models. Given an fc-deterministic system model, it is possible to build a 3DFA encoding all the possible conceptual models that allow full-control of it. All the traces of the system model must be accepted by the 3DFA. From

those accepted traces, two scenarios are possible. Either those accepted traces are extended with a command that is not possible for one of the underlying execution in the system, and then, the extended traces must be rejected by the 3DFA. Or those accepted traces are extended with an observation that is not possible on all the underlying executions, in which case the extended traces are don't care traces. Figure 5.12(a) illustrates that characterisation of the traces of an HMI-LTS system model as a 3DFA. It can be observed that whenever a trace is in Rej (resp. in DC), any extension of it remains in Rej (resp. DC).
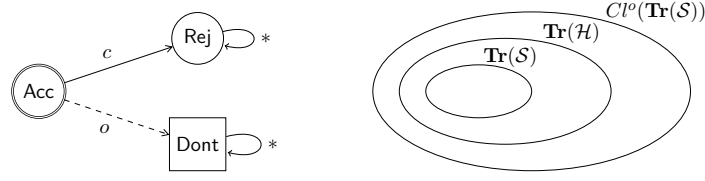
(a) Trace characterisation as 3DFA. (b) Relation between full-control and trace preorder.

Figure 5.12. Traces characterisation of full-control conceptual models.

Definition 5.7 (3DFA characterisation of full-control conceptual models). *Given an fc-deterministic HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, all the HMI-LTSs that allow full-control of the system can be represented as a single 3DFA $\mathcal{C}(\mathcal{S}) = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$ such that, given a sequence $\sigma \in \Sigma^*$ and an action $\alpha \in \Sigma$, we have $\varepsilon \in L_{Acc}$ and:*

- *$\sigma\alpha \in L_{Rej}$ if either $\sigma \in L_{Rej}$ or $\sigma \in L_{Acc}$, $\alpha \in \mathcal{L}^c$ and $\exists s \in (s_{0_S} \textbf{ after } \sigma)$ such that $\alpha \notin A^c(s)$;*

- *$\sigma\alpha \in L_{Acc}$ if either $\sigma \in L_{Acc}, \alpha \in \mathcal{L}^c$ and $\forall s \in (s_{0_S} \textbf{ after } \sigma) : \alpha \in A^c(s)$ or $\sigma \in L_{Acc}, \alpha \in \mathcal{L}^o$ and $\exists s \in (s_{0_S} \textbf{ after } \sigma) : \alpha \in A^o(s)$;*

- *$\sigma\alpha \in L_{DC}$ if either $\sigma \in L_{DC}$ or $\sigma \in L_{Acc}$, $\alpha \in \mathcal{L}^o$ and $\forall s \in (s_{0_S} \textbf{ after } \sigma) : \alpha \notin A^o(s)$.*

As a reminder, an HMI-LTS $\mathcal{H}$ allows full-control of a system $\mathcal{S}$ if and only if $\textbf{Tr}(\mathcal{S}) \subseteq \textbf{Tr}(\mathcal{H})$ and $\textbf{Tr}(\mathcal{H}) \subseteq Cl^o(\textbf{Tr}(\mathcal{S}))$, as illustrated by Figure 5.12(b). That property means that all the traces of $\mathcal{S}$ must be present in $\mathcal{H}$ and that all the traces of $\mathcal{H}$ must be contained in

the observation-closure of the traces of $\mathcal{S}$. That is precisely the kind of characterisation that is made possible with the 3DFA, with $\mathbf{Tr}(\mathcal{S})$ corresponding to the accepted traces, $\overline{Cl^o(\mathbf{Tr}(\mathcal{S}))}$ being the rejected traces and finally $Cl^o(\mathbf{Tr}(\mathcal{S})) \setminus \mathbf{Tr}(\mathcal{S})$ being the don't care traces.

Theorem 5.8. *Given an fc-deterministic HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, a deterministic and non-divergent HMI-LTS $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$ and the 3DFA $\mathcal{C}(\mathcal{S}) = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$:*

$$\mathcal{H} \, \mathbf{fc} \, \mathcal{S} \iff L_{Acc} \subseteq \mathbf{Tr}(\mathcal{H}) \subseteq \overline{L_{Rej}}$$

Proof. Let us first suppose that $\mathcal{H} \, \mathbf{fc} \, \mathcal{S}$; thus, by Property 4.17, it means that $\mathbf{Tr}(\mathcal{S}) \subseteq \mathbf{Tr}(\mathcal{H})$ and $\mathbf{Tr}(\mathcal{H}) \subseteq Cl^o(\mathbf{Tr}(\mathcal{S}))$. Moreover, it implies that $\mathcal{S}$ is fc-deterministic, and so $L_{Acc} = \mathbf{Tr}(\mathcal{S})$. Consequently, $L_{Acc} \subseteq \mathbf{Tr}(\mathcal{H})$. As a reminder, $Cl^o(\mathbf{Tr}(\mathcal{S})) = \mathbf{Tr}(\mathcal{S}) \cup \{\sigma o \sigma' \mid \sigma o \notin \mathbf{Tr}(\mathcal{S}), o \in \mathcal{L}^o$ and $\sigma' \in \mathcal{L}^*\}$, which corresponds to $\overline{L_{Rej}} = L_{Acc} \cup L_{DC}$ and consequently $\mathbf{Tr}(\mathcal{H}) \subseteq \overline{L_{Rej}}$. $\square$

5.3 3DFA-based Approach

As presented in the previous chapter, Theorem 4.6 states that if a system model is fc-deterministic, its determinisation is a full-control conceptual model for it. However, the obtained conceptual model is not minimal in the general case. This section presents an algorithm that is based on the determinisation of a completed version of the system model, that built a 3DFA from which a minimal consistent DFA can be computed. That DFA corresponds to a minimal full-control conceptual model for the system model. The *3DFA-based generation algorithm* is composed of the three steps summarised by Figure 5.13.
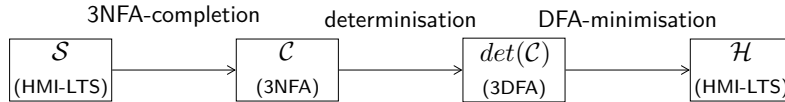
Figure 5.13. Three steps of the 3DFA-based minimal full-control conceptual model generation algorithm.

The first step of the algorithm consists in completing the system model so as to make explicit the 3DFA characterisation of its full-control conceptual models. For that, two states are added to the system model: one *error state* Π and one *don't care* state Δ. Those two states have loop transitions for every action of the alphabet. Transitions are added for the states of the system model so that every action of the alphabet is possible for all the states. Missing commands lead to the error state and missing observations lead to the don't care state. The model that is obtained is not a 3DFA since it can exhibit some non-determinism, it is in fact a *three-valued non-deterministic finite automaton* (3NFA).

Definition 5.9 (HMI-LTS 3NFA-completion)**.** *Given an fc-deterministic HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \to_S \rangle$, *its* 3NFA-completion *is the 3NFA* $\mathcal{C} = \langle \mathcal{L}^c \cup \mathcal{L}^o, S_S \cup \{\Pi, \Delta\}, s_{0_S}, \delta, S_S, \{\Pi\}, \{\Delta\} \rangle$ *where:*

$$
\begin{aligned}
\delta \quad = \quad & \to_S \cup \{(\Pi, a, \Pi) \mid a \in \mathcal{L}\} \cup \{(\Delta, a, \Delta) \mid a \in \mathcal{L}\} \\
& \cup \{(s, c, \Pi) \mid c \in \mathcal{L}^c \setminus A^c(s)\} \cup \{(s, o, \Delta) \mid o \in \mathcal{L}^o \setminus A^o(s)\}
\end{aligned}
$$

Algorithm 8 in Appendix C shows the 3NFA-completion algorithm. Figure 5.14 shows an HMI-LTS system model example, along with its 3NFA-completion. The 3NFA-completed system model clearly exhibits the 3DFA characterisation of all the full-control conceptual models for the system model, as illustrated above on Figure 5.12.

(a) A system model $\mathcal{M}$.   (b) The 3NFA-completion of $\mathcal{M}$.

Figure 5.14. An HMI-LTS system model example and its 3NFA-completion.

Once the system model has been 3NFA-completed, the second step of the algorithm consists in determinising it. That operation can be done with the classical subset construction presented in Section 3.1.5,

slightly adapted to handle 3NFA. The model that is produced after this step is thus a 3DFA that contains exactly the same traces than the 3NFA-completed system model, by definition of the determinisation. Algorithm 9 in Appendix C shows the 3NFA determinisation algorithm.

Figure 5.15 shows the determinisation of the 3NFA-completed system model example. During the determinisation, states containing the Π state are considered as *Rej* states and states containing at least one accepting state are considered as *Acc* states.
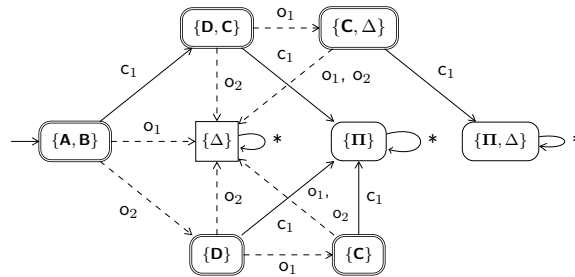

Figure 5.15. The determinisation of the 3NFA-completed system model example of Figure 5.14(b).

The last step of the algorithm is the minimisation of the obtained 3DFA, in order to get a minimal consistent DFA. The minimal DFA can be obtained with the algorithm presented in Section 5.2.2. Before minimisation, all the error states of the determinised 3NFA are merged together as a single error state Π, which results in a model with one error state and one don't care state. Once minimised, the error state is simply removed to get a minimal full-control conceptual model which is an HMI-LTS.

Figure 5.16(a) shows the final implication table for the example. The DFA-minimisation algorithm then produces three compatibles that is also the set of maximal compatibles from which the minimal consistent DFA shown on Figure 5.16(b) is produced.

Correctness, Termination and Time Complexity

Given that the system model is fc-deterministic, the 3DFA-based algorithm is guaranteed to find the minimal full-control conceptual model.
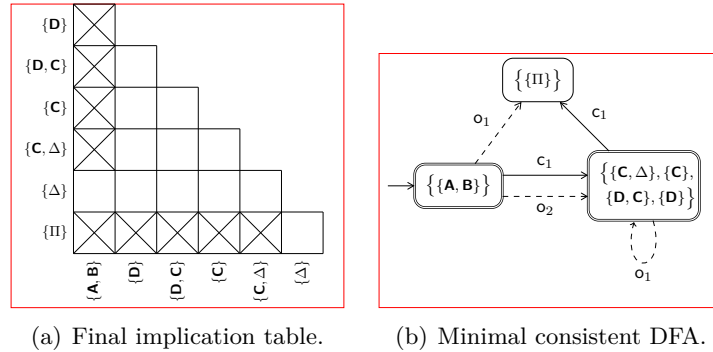
(a) Final implication table.   (b) Minimal consistent DFA.

Figure 5.16. Minimisation of the 3DFA obtained after determinisation of the 3NFA-completed system model, for the example of Figure 5.14(a).

Property 5.7 provides the 3DFA characterisation of full-control conceptual models. The 3NFA-completion algorithm exactly builds a 3NFA that satisfies this characterisation. Indeed, all the traces of the system model are kept, while missing traces are either added to be rejecting or don't care traces, according to the characterisation. The second step of the algorithm is a determinisation that keeps the traces. The resulting 3DFA has therefore all the traces characterising full-control conceptual models. Finally, the DFA-minimisation step is guaranteed to explore all the possible consistent DFA and to find the minimal one.

The time complexity of the algorithm is exponential in the number of states of the system model. Indeed, the determinisation and DFA-minimisation steps both correspond to an exponential time algorithm.

5.4 Reduction-based Approach

A full-control conceptual model can be automatically generated for fc-deterministic system models using an algorithm based on a variant of the Paige-Tarjan algorithm [PT87] that is used to solve the coarsest partition refinement problem. This section presents the *reduction-based generation algorithm.* In the general case, the proposed algorithm will not be able to compute the minimal full-control conceptual model, but will get one that is as small as possible. The algorithm is composed of two steps, as illustrated by Figure 5.17.
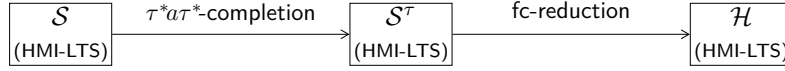
$\mathcal{S}$	$\tau^*a\tau^*$-completion	$\mathcal{S}^\tau$	fc-reduction	$\mathcal{H}$
(HMI-LTS)		(HMI-LTS)		(HMI-LTS)

Figure 5.17. Two steps of the reduction-based minimal full-control conceptual model generation algorithm.

5.4.1 Eliminating τ-transitions

The first step of the reduction-based approach is to eliminate τ-transitions from the system model, while keeping the same behaviour. A first way to eliminate τ-transitions is to build a new LTS whose transition relation makes explicit the weak transitions. This involves the standard $\tau^*a\tau^*$-*completion* construction [KS83] which computes the reflexive and transitive closure with respect to τ's. That construction is also referred to as the normal-form with respect to observation equivalence in [FM91].

Definition 5.10 ($\tau^*a\tau^*$-completion)**.** *Given an LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, *its* $\tau^*a\tau^*$-completion *is an LTS defined as* $\mathcal{M}^\tau = \langle S, \mathcal{L}, s_0, \rightarrow^\tau \rangle$, *with* $\rightarrow^\tau = \{(s, a, s') \mid s \xrightarrow{\tau^*a\tau^*} s', a \in \mathcal{L} \cup \{\tau\}\}$.

The $\tau^*a\tau^*$-completion makes directly available the sets of possible actions for all the states of the LTS, that is, $\Gamma(s) = A(s)$. Another possible way to eliminate τ-transitions from an LTS is with the τ^*a-*completion*. A difference between the two kinds of completion is that the latter does not make directly available the set of post-states, that is, states reachable with one single strong transition.

Definition 5.11 (τ^*-completion)**.** *Given an LTS* $\mathcal{M} = \langle S, \mathcal{L}, s_0, \rightarrow \rangle$, *its* τ^*a-completion *is an LTS defined as* $\mathcal{M}^{\tau^*} = \langle S, \mathcal{L}, s_0, \rightarrow^{\tau^*} \rangle$ *and whose transition relation is defined as* $\rightarrow^{\tau^*} = \{(s, a, s') \mid s \xrightarrow{\tau^*a} s'\}$.

Figure 5.18 shows the two kinds of completion on a simple LTS. In general, the $\tau^*a\tau^*$-completion results in an addition of more transitions than the τ^*a-completion. Another difference between the two completions is that the τ-transitions can be removed from the first one while preserving the set of reachable states, which is not the case with the τ^*a-completion.
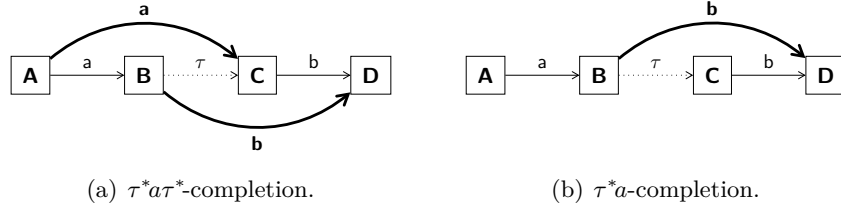
(a) $\tau^*a\tau^*$-completion. (b) τ^*a-completion.

Figure 5.18. Two examples illustrating the elimination of τ-transitions.

5.4.2 Partition Refinement

Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ representing a system model, the reduction-based algorithm works by iteratively refining an initial partition of the states S_S until that partition becomes stable. The elements of that stable partition are the states of the reduced model which is a full-control conceptual model. The τ-transitions are first removed from the system model, completed with the $\tau^*a\tau^{\underline{*}}$completion. For the remainder of this section, *system model* will refer to the model without τ-transitions.

Partition

A *partition* of the states of a system model $\mathcal{S}$ is a set $P = \{B_i \subseteq S_S \mid \bigcup_i B_i = S_S$ and $B_i \cap B_j = \emptyset$ when $i \neq j\}$. The elements B_i of a partition are called *blocks* of the partition. Each state $s \in S_S$ belongs to one block of the partition P. The notation $[s]_P$ is used to represent the unique block of the partition P containing s.

A partition P' *refines* a partition P if for every block of P', there exists a block of P containing it. Intuitively, a partition P' is a refinement of a partition P if it has the same blocks than those of P with the exception that some of those blocks have been cut into smaller blocks.

Definition 5.12 (Refined partition)**.** *Let P and P' be two partitions of a set of states S. The partition P' refines P, which is denoted $P' \sqsubseteq P$, if and only if:*

$$\forall B' \in P' : \exists B \in P : B' \subseteq B$$

The partition P' is said to be finer *than P and P is said to be* coarser *than P'. If the two partitions are different ($P \neq P'$), P' is said to be* strictly finer *than P and P is said to be* strictly coarser *than P'.*

Full-control Stability

The idea of the reduction-based algorithm is to start from an initial partition and to refine it until it becomes stable. At that point, a reduced model can be built from the resulting stable partition. A partition is *fc-stable* (full-control stable) if all its blocks are *fc-stable* according to the Definition 5.13. The notation $B\,\mathbf{after}\,a$ is used to refer to the set of states that can be reach from any state in B with the action a, that is, $B\,\mathbf{after}\,a = \{s' \in S \mid s \stackrel{a}{\Longrightarrow} s' \wedge s \in B\}$.

Definition 5.13 (Fc-stable block)**.** *Let P be a partition of the set of states S_S of a system model $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$. A block $B \in P$ is fc-stable according to an action $a \in \mathcal{L}$ if and only if:*

$$
\begin{aligned}
a \in \mathcal{L}^c &\implies (B\,\mathbf{after}\,a) = \emptyset \\
&\qquad \vee\ (\forall s \in B : a \in \Gamma(s)\ \wedge\ \exists B' \in P : (B\,\mathbf{after}\,a) \subseteq B') \\
\wedge\quad a \in \mathcal{L}^o &\implies \exists B' \in P : (B\,\mathbf{after}\,a) \subseteq B'
\end{aligned}
$$

The block B is fc-stable *if it is fc-stable according to all the actions in $\mathcal{L}$.*

   Intuitively, it means that all the states of an fc-stable block have the same sets of enabled commands. There are no constraints for the enabled observations. In both cases, for each action, triggering it from any state of the fc-stable block B always leads into the same block B'.

Splitting Pair

The reduction-based algorithm works by successively refining the partition until it gets fc-stable. In order to refine the partition, a *splitting pair* is used to split one block of the partition that is not fc-stable into new blocks.

   Given two blocks $B, B' \in P$ and an action $a \in \mathcal{L}$, B' is a *splitter* of B according to a if there exists states $s \in B$ such that $a \notin \Gamma(s)$ or such that $s \stackrel{a}{\longrightarrow} s'$ and $[s']_P \neq B'$.

As illustrated by Figure 5.19, the states of the block B can be partitioned according to the action a, into three sets of states:

- $B_1 = \{s \in B \mid \exists s \xrightarrow{a} s' : [s']_P = B'\}$;
- $B_2 = \{s \in B \mid \exists s \xrightarrow{a} s' : [s']_P \neq B'\}$;
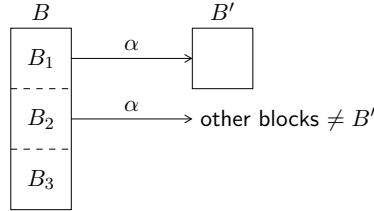- $B_3 = \{s \in B \mid a \notin \Gamma(s)\}$.

Figure 5.19. A block splitter for the reduction-based algorithm.

The pair (B, B') is a *splitting pair* according to the action a and is used in the refinement step of the reduction-based algorithm to refine the current partition. If the action is a command, the refinement step consists in replacing the block B with the blocks B_1, B_2. If B_3 is not empty, the refinement step is not unique anymore. Such a situation can occur when a is an observation and it is precisely due to the fact that two states that do not have the same observations may be considered as the same state by the operator in the conceptual model. It is precisely if such a situation occurs that the algorithm is not guaranteed to find a minimal full-control conceptual model. The refinement step consists in replacing the block B with the blocks B_1 and B_2. The states from the block B_3 may in fact either go with the block B_1 or with the block B_2, which can lead to different final fc-stable partitions, some of them not leading to a minimal full-control conceptual model.

5.4.3 The Reduction-based Algorithm

Given the definition of fc-stable partition and splitting pair, Algorithm 3 shows the reduction-based algorithm in a high-level fashion. The first step consists in removing the τ transitions, with the $\tau^*a\tau^*$-elimination algorithm. The next step computes the initial partition P_0. As all the

states from an fc-stable block must have exactly the same set of enabled commands, the initial partition is built according to commands:

$$P_0 = \{[s]_c \mid s \in S\} \text{ with } [s]_c = \{s' \in S \mid \Gamma^c(s) = \Gamma^c(s')\}.$$

From the initial partition, the algorithm proceeds by steps. At each step, an unstable block B along with a splitting pair is found and refined. The algorithm continues until the partition becomes stable. The $refine\,(P, B, B', \alpha)$ procedure handles the block splitting strategy for the case of a splitting pair according to an observation.

Algorithm 3: Reduction-based algorithm to compute full-control conceptual model.

Input: $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ a system model
Output: $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$ a full-control conceptual model

$\mathcal{S} \leftarrow eliminate\_taus\,(\mathcal{S})$
$P \leftarrow initial\_partition\,(S_S)$
while not P **is** *fc-stable* **do**
    $(B, B', \alpha) \leftarrow find\_splitting\_pair\,(P)$
    $P \leftarrow refine\,(P, B, B', \alpha)$
return $generate\_model\,(\mathcal{S}, P)$

Correctness, Termination and Time complexity

The algorithm is guaranteed to terminate in at most $|S_S| - |P_0|$ refinement steps. At each iteration, the partition is not fc-stable, which means that there exists at least one non fc-stable block in the partition. Moreover, the refinement step always split a block into two new blocks giving a strictly finer partition for the next iteration, with the total number of blocks increased by one.

In order to prove the correctness of the algorithm, it suffices to prove that any fc-stable refinement of the initial partition is also an fc-stable refinement of the current partition, by induction on the number of refinement steps. Let us suppose that the current partition is P and that this latter will be refined to a partition Q, using the (B, B') splitting pair, in the next refinement step of the algorithm. Let R be any fc-stable refinement of P_0, which is also a refinement of P, by induction hypothesis. Let C be any block of R, it suffices to show that C is included in a block
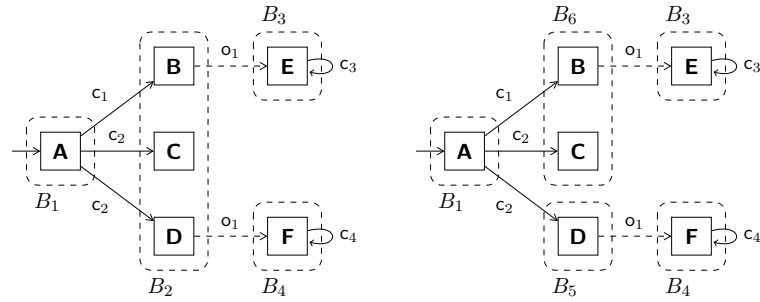
of Q. Since R is a refinement of P, it means that C is included in a block D of P. If $D \neq B$, the D block still is in Q and thus C is included in a block of Q, so let us suppose that $D = B$. In the next refinement step, the B block is split into two blocks B_1 and B_2 and the block C should be included in one of the two blocks. Let us suppose that it is not the case, that is there exist two states $s_1, s_2 \in C$ such that $s_1 \in B_1$ and $s_2 \in B_2$. That would mean that:

- Either there is a action a such that both s_1 and s_2 have that action enabled, leading to different blocks in P, and thus also to different blocks in R, which is impossible since R is fc-stable.

- Or there is an observation o that is enabled in s_1 but not in s_2. In such a situation, it may be possible that s_1 and s_2 are not both in B_1 or B_2 since the splitting operation is not unique anymore and s_2 could go in either B_1 or B_2. But since the algorithm have to explore all the possible splitting operations, there is indeed one splitting operation that guarantees that s_1 and s_2 will not be separated.

Without considering the exploration of all the possible ways to split the states from the third block B_3 between the B_1 and B_2 blocks, the time complexity of the algorithm is $\mathcal{O}(|S_S| \cdot | \to_S |)$, following an implementation similar to the one proposed in [GV90]. Considering the algorithm with the exploration of all possible splits, the time complexity increases much. The worst situation is a system that only contains observations, and for which all the states have different sets of enabled observations. In that case, the additional time complexity is exponential. Indeed, in each refinement step, there will be 2^X different possible splits, X depending on the step.

Non-transitivity Situations

As already introduced above, given a splitting pair, the splitting operation is not always unique. Such a situation can occur when the action that is used for the splitting pair is an observation. Figure 5.20 shows a system model for which the splitting operation is not unique in the current situation, if the chosen splitting pair is (B_2, B_3). There are in fact two possible splits: the block B_2 is either split in $\{\mathbf{B}, \mathbf{C}\}$ and $\{\mathbf{D}\}$ or in $\{\mathbf{B}\}$ and $\{\mathbf{C}, \mathbf{D}\}$. Indeed, states $\mathbf{B}$ and $\mathbf{D}$ cannot be in the same block

since they share a common observation leading to different blocks. But the state **C** do not have this observation possible, which is allowed by full-control, and can thus go with either **B** or **D**.

Figure 5.20. Example of a system model for which the splitting operation is not unique in the current iteration of the reduction-based algorithm.

In order for the reduction-based algorithm to find the minimal full-control conceptual model, all the possible splitting operations must be considered, which results in an exploration of all the possible choices. Of course, doing so requires much more time since the number of different splitting at each step can be large. If the final goal is not to reach minimality, it is however possible to avoid exploring the possible splittings by setting up some heuristic. Examples of heuristics include choosing randomly where to put the states that do not have the observation of the splitting pair, or to put those states in the largest block.

But, it can be worse than just not getting the minimal full-control conceptual model. Indeed, in some situations, the algorithm can terminate by incorrectly reporting some fc-determinism issue. Figure 5.21(a) shows a system model where a wrong output can occur. The block B_2 has to be split, and states **B** and **D** must be separated. The state **C** can go either with **B** or **D**. If the choice is made to put it with **B** as shown on Figure 5.21(b). The issue is that the block B_1 is not fc-stable anymore, according to command c_2. In the next iteration of the algorithm, the B_1 block has to be split but it cannot be since it would require to split the state **A**. The reduction algorithm exits with an error and gives a false diagnostic of a non-fc-deterministic system model.

In summary, for systems containing observations, the splitting operation does not always result in one unique split. In order to find the

(a) Initial partition. A splitting pair is (B_2, B_3) with respect to the o_1 observation.

(b) Intermediate partition of the reduction algorithm which exhibits an fc-determinism issue.

Figure 5.21. Example of a system model for which some heuristics for the non-transitive situations may lead to a wrong non-full-control diagnostic.

minimal full-control conceptual model, all the possible splits must be explored. Given the potential high cost in term of execution time, it is possible not to explore all the possible splits and chose an heuristic. But there are two possible drawbacks: there is no more guarantees that the generated conceptual model is the minimal one and the algorithm can exit by wrongly indicating an fc-determinism issue.

Minimality

The minimal full-control conceptual model cannot always be computed with the reduction-based algorithm. Figure 5.22 shows an example of a system model whose corresponding minimal full-control conceptual model cannot be represented by partitioning the state of the system model.

At first, it looks like the minimal full-control conceptual model is obtained by merging together states **B** and **D** to get the state **1**. But it is not the case since if it was, there would be a loop transition $1 \xrightarrow{b} 1$, which would mean that the trace $\langle c_1, b, a, c_4 \rangle$ is wrongly accepted, meaning that the conceptual model does not allow full-control of the system.

Such system models cannot be addressed by the reduction-based algorithm. The reduction-based approach is therefore not guaranteed to find the minimal full-control conceptual model. To do so, either
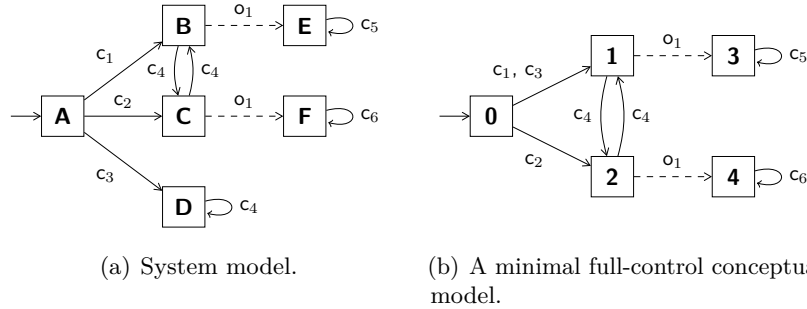
(a) System model.

(b) A minimal full-control conceptual model.

Figure 5.22. Example of a system model whose minimal full-control abstraction cannot be computed directly with a reduction approach (inspired from Fig 2 from [PO99]).

the 3DFA-based algorithm presented previously or the learning-based approach presented in the next section has to be used.

5.5 Learning-based approach

This section presents another technique based on a learning approach where the full-control conceptual model is grown incrementally, starting from a single-state conceptual model. The first part of the section explains the L^* algorithm which is the base of the proposed *learning-based generation algorithm*. The original L^* algorithm which is used to learn DFAs has been extended to learn 3DFAs and is presented in the second part of the section. A review of active learning can be found in [SHM10].

5.5.1 The L^* learning algorithm

The L^* *algorithm*, designed by Angluin [Ang87], is used to automatically learn an unknown regular language over a given alphabet, and to build a deterministic finite automaton (DFA) that accepts the learned language. This section summarises the important points of the approach. All the details are in the original paper of Angluin [Ang87].

The L^* algorithm is an *active* learning algorithm in the sense that it consists of two components interacting together: the *learner* interacts with the *teacher* in order to learn the unknown language U, that is a DFA.

The learner can ask two types of questions to the teacher. Figure 5.23 shows the global view of the whole learning framework. The box on the left if the learner and the dashed box on the right is the teacher. The two types of questions that the learner can ask to the teacher are:

- A *membership query* $MQ(\sigma)$ that asks whether the string σ belongs to the unknown regular language U or not. The teacher has to answer with either true (**T**) or false (**F**).

- A *conjecture check* $Conj(\mathcal{C})$ that asks whether the candidate DFA $\mathcal{C}$ accepts the unknown regular language U or not. If it is not the case, the teacher has to provide as a counterexample a string *cex* that is accepted by the candidate DFA $\mathcal{C}$ but does not belong to U, or vice-versa.

A teacher capable of answering correctly these two types of questions is referred to as a *minimally adequate teacher*. Provided such a teacher, the L^* algorithm is guaranteed to learn a correct DFA that accepts the unknown regular language U.
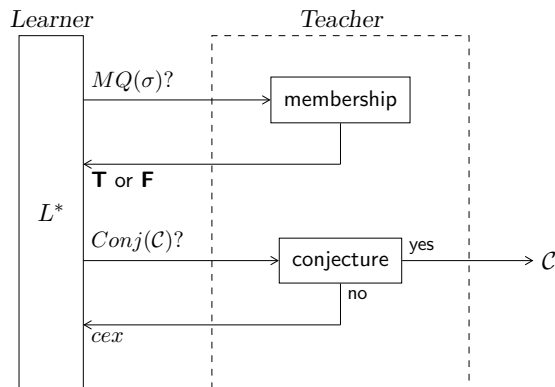
Figure 5.23. General overview of the learning framework used with the L^* algorithm to learn an unknown regular language U over a given alphabet. The L^* algorithm (on the left) interacts with a teacher (on the right).

Depending whether the teacher is automatic or requires the intervention of a human, the L^* learning algorithm is fully or semi-automatic. The L^* algorithm works in an iterative way consisting of two phases.

In the first phase, the learner questions the teacher with membership queries. Results from those queries are stored in an *observation table*. At some time, when the learner has enough new information, it produces a candidate DFA which is sent to the conjecture check. If the check succeeds, the algorithm finishes and output the candidate as its result. In the other case, a new iteration with membership queries begins, guided with the counterexample provided by the teacher.

Algorithm 4 presents the overall structure of the L^* algorithm. The first step is the initialisation of the observation table. After that, the main loop of the algorithm runs until a candidate DFA that passes the conjecture check is found. In the first phase, the observation table is updated with membership queries, until it gets closed and consistent. These two criteria are necessary to ensure that a candidate DFA covering the observation table can be generated. This precisely leads to the next step where a candidate DFA is built and passed through the conjecture check. If the teacher answers yes, the algorithm terminates and the candidate DFA is a valid automaton accepting U. If the teacher answers no, the observation table is updated to take into account information from the counterexample *cex* and then a new loop begins.

Algorithm 4: L^* algorithm.

Input: *Teacher* is a minimally adequate teacher for U, an unknown regular
language on an alphabet A

Output: $\mathcal{C}$, a DFA that accepts the language U

$T \leftarrow createObservationTable()$
repeat
   while T **is not** *closed* **or not** *consistent* **do**
      $T \leftarrow updateObservationTable\ (Teacher, T)$
   $\mathcal{C} \leftarrow buildCandidate\ (T)$
   if $Conj(\mathcal{C})\ =\ (\textbf{no, } cex)$ **then**
      $T \leftarrow updateObservationTable\ (Teacher, T, cex)$
until $Conj(\mathcal{C})\ =\ (\textbf{yes, } –)$
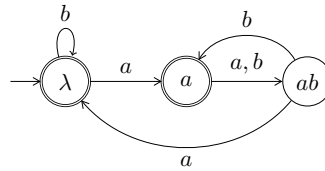

Observation Table

The L^* algorithm stores the results of the membership queries that it has done in an observation table. The observation table is a triple (S, E, T)

that is composed of two sets of strings S and E and of a function T which maps strings from the set $(S \cup S \cdot A) \cdot E$ to boolean values $\{true, false\}$, where A is the alphabet of the DFA. The set S (resp. E) is a non-empty prefix-closed (resp. suffix-closed) set of strings, meaning that any prefix (resp. suffix) of an element of the set also belongs to the set.

The observation table is initialised with $S = E = \{\lambda\}$. It can be viewed as a two dimensional table as the one shown on Figure 5.24(a). The rows of the table correspond to elements from S and $S \cdot A$ while columns correspond to elements from E. The values indicated in the body of the table are the value of the T mapping function indicating whether the corresponding string belongs or not to the unknown regular language U.

		E	
		λ	b
S	λ	true	false
	a	true	true
	ab	false	true
$S \cdot A$	b	true	false
	aa	false	true
	aba	true	false
	abb	true	true

(a) Observation table.   (b) Acceptor DFA.

Figure 5.24. Example of an observation table (S, E, T) that is closed and consistent along with the corresponding DFA that accepts the language defined by the table. The alphabet of the unknown language to be learned is $A = \{a, b\}$.

An observation table is *closed* if for every element $t \in S \cdot A$, there is an element $s \in S$ such that both rows of T corresponding to t and s are the same. An observation table is *consistent* if for any two elements $s_1, s_2 \in S$ such that their corresponding rows in T are the same, the rows corresponding to the elements $s_1 \cdot a$ and $s_2 \cdot a$ must also be the same. Given an observation table that is closed and consistent, it is possible to build a DFA that accepts the language defined by the T mapping function. That DFA is called an *acceptor* for the observation table. Figure 5.24(b) shows an acceptor corresponding to the observation table example.

Every row of the observation table, corresponding to an element $s \in S$, corresponds to a state of the acceptor. A state of the acceptor

is an accepting state if $T(s) = true$. Rows for elements $s \in (S \cup S \cdot A)$ represent the transition relation.

Correctness, Termination and Time complexity

The L^* algorithm is correct and always terminates provided that the teacher is minimally adequate. Moreover the L^* algorithm terminates after making at most n conjectures, that is, executing its main loop at most $n - 1$ times where n is the number of states in the minimum acceptor for the unknown regular language U.

The time complexity of the L^* algorithm also depends on m, the length of the counterexamples provided by the teacher in response of a conjecture query that fails. If such counterexamples are too long, it takes indeed more time for them to be processed by the algorithm to update the observation table.

In summary, given that the teacher is a minimally adequate teacher, the L^* algorithm eventually terminates and produces a DFA that is isomorphic to the minimal DFA accepting the regular unknown language U. The total running time of the algorithm is bounded by a polynomial function in m and n.

5.5.2 Learning a 3DFA

The L^* algorithm of Angluin has been extended by Chen et al. [CFC$^+$09] in order to learn a 3DFA. More precisely, the variant of L^* proposed by Chen et al. learns a minimal separating DFA for two disjoint regular language. A *separating DFA* for two languages L_1 and L_2 is one that accepts everything that L_1 accepts and rejects everything that L_2 rejects. For those strings that are neither in L_1 or in L_2, they can belong to the language of the separating DFA or not. In fact, those strings correspond to don't care strings.

The goal of Chen et al. is to find a minimal separating DFA that separates two given languages L_1 and L_2. In other words, they want to find a DFA $\mathcal{A}$ with the minimal number of states, so that $L_1 \subseteq L(\mathcal{A}) \subseteq \overline{L_2}$. Their approach to solve the problem is to use a variant of the L^* algorithm that uses a 3DFA to collect the samples coming from the L_1 and L_2 languages. The set L_1 is considered as Acc, and the set L_2 as Rej.

For a 3DFA to encode correctly separating DFAs for two given languages L_1 and L_2, it must be *sound* and *complete* with respect to both L_1 and L_2. Figure 5.25 illustrates those two conditions. The rows of the table represent the languages L_1 and L_2 that have to be separated, along with the set of don't care traces. The columns of the table represent the 3DFA candidate that is input to the conjecture check algorithm.

- The 3DFA $\mathcal{C}$ is sound with respect to L_1 and L_2 if any DFA that is consistent with $\mathcal{C}$ separates L_1 and L_2. In other words, the two following conditions must hold: $L_1 \subseteq L(\mathcal{C}^-)$ and $L(\mathcal{C}^+) \subseteq \overline{L_2}$.

  A string σ fails to satisfy the first condition if $\sigma \in L_1$ and $\sigma \notin L(\mathcal{C}^-)$ (cells 2 and 3 of the table) and fails to satisfy the second condition if $\sigma \in L_2$ and $\sigma \in L(\mathcal{C}^+)$ (cells 7 and 8 of the table).

- The 3DFA $\mathcal{C}$ is complete with respect to L_1 and L_2 if any DFA separating L_1 and L_2 is consistent with $\mathcal{C}$. In other words, the two following conditions must hold: $L(\mathcal{C}^-) \subseteq L_1$ and $\overline{L_2} \subseteq L(\mathcal{C}^+)$.

  A string σ fails to satisfy the first condition if $\sigma \notin L_1$ and $\sigma \in L(\mathcal{C}^-)$ (cells 4 and 7 of the table) and fails to satisfy the second condition if $\sigma \in \overline{L_2}$ and $\sigma \notin L(\mathcal{C}^+)$ (cells 3 and 6 of the table).

Figure 5.25. Representation of the set of traces of the languages L_1 and L_2 to be separated along with the 3DFA candidate that is input to the conjecture check algorithm of the L^{sep} learning algorithm.

The L^{sep} learning algorithm

The L^{sep} *algorithm* proposed by Chen et al. is a variation of the L^* algorithm from Angluin, except that it works with 3DFAs instead of classical DFAs. The first difference is that the observation table is now filled with three possible values corresponding to the three sets of a 3DFA: **T** for accepting traces, **F** for rejecting traces and **DC** for don't care traces. Figure 5.26 shows an example of such an observation table with the corresponding 3DFA.

		E	
		λ	b
	λ	**F**	**DC**
	b	**DC**	**DC**
S	ba	**F**	**T**
	bab	**T**	**T**
	a	**F**	**DC**
	bb	**DC**	**DC**
$S \cdot A$	baa	**F**	**T**
	$baba$	**F**	**DC**
	$babb$	**T**	**T**

(a) Observation table.
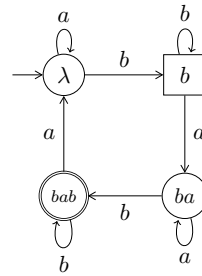
(b) 3DFA corresponding to the (S, E, T) observation table.

Figure 5.26. Example of an observation table (S, E, T) along with the corresponding 3DFA. The alphabet used is $A = \{a, b\}$ (taken from Fig 4 from [CFC$^+$09]).

The membership query simply replies **T** if the trace it is asked about belongs to L_1, **F** if it belongs to L_2 and **DC** for all the other traces. The conjecture check is a little more complicated since it has to check for the soundness and completeness conditions just presented here above. The conjecture check is composed of several checks performed in a specific order depicted on Figure 5.27.

- The completeness check is performed first, that is, the two containments queries $L_1 \supseteq L(\mathcal{C}^-)$ and $\overline{L_2} \subseteq L(\mathcal{C}^+)$ are performed. If either of the queries fails, a counterexample is produced and sent back to the learning algorithm so that to extend the observation table and refine the candidate.

- Once the candidate $\mathcal{C}$ is proved to be complete, that is, any DFA separating L_1 and L_2 is consistent with $\mathcal{C}$, it is minimised in order

to get a minimal consistent DFA $\mathcal{A}$, using the algorithm presented in Section 5.2.2, for example.

- Finally, the minimal consistent DFA $\mathcal{A}$ is checked for soundness, that is, the two containments queries $L_1 \subseteq L(\mathcal{A})$ and $L(\mathcal{A}) \subseteq \overline{L_2}$ are performed. If either of the queries fails, a counterexample is produced and the whole process starts again. Otherwise, $\mathcal{A}$ is a minimal separating DFA for L_1 and L_2.
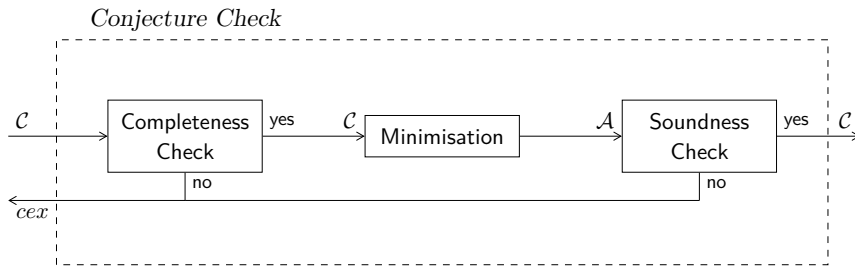

Figure 5.27. The conjecture check of the L^{sep} algorithm is composed of three successive checks. The completeness check is followed by the minimisation of the 3DFA that is then checked regarding soundness.

Correctness, Termination and Time Complexity

As it is the case with the L^* algorithm, the L^{sep} algorithm is also correct and always terminates outputting a minimal separating DFA for L_1 and L_2. The L^{sep} algorithm is also guaranteed to find a minimal 3DFA using at most $n-1$ loops of the learning algorithm, where n is the number of states of the minimal 3DFA representing the minimal separating DFA for L_1 and L_2. The time complexity also depends on the length m of the counterexamples provided by the teacher whenever a conjecture check fails. In summary, the candidate generator takes $\mathcal{O}(n^2 + n \log m)$ membership queries and calls $n-1$ times the conjecture check.

5.5.3 Learning a Minimal Full-control Conceptual Model

This section presents the $L^{\mathbf{fc}}$ *learning algorithm* that is used to generate a minimal full-control conceptual model for a given system model. The

proposed algorithm is inspired from the L^{sep} algorithm proposed by Chen et al. and presented here above. The algorithm is heavily asked to solve model checking problems, on the system model which is first completed in some different ways. The system model can be completed with the addition off an error state, just like the demonic completion presented in Section 3.4.1.

Definition 5.14 (Completion on error)**.** *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, its completion on error with respect to commands (resp. observation or both) is denoted $\mathcal{S}_{\rightarrow_c \Pi}$ and defined as the HMI-LTS $\langle S_S \cup \{\Pi\}, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \cup \{(\Pi, a, \Pi) \mid a \in \mathcal{L}\} \cup \{(s, a, \Pi) \mid a \in \mathcal{L}^c \setminus A^c(s)\}\rangle$.*
The completion on observation ($\mathcal{S}_{\rightarrow_o \Pi}$) and on both commands and observations ($\mathcal{S}_{\rightarrow \Pi}$) are defined in a similar way.

Membership Query Algorithm

The *membership query* (MQ) determines whether a given sequence of actions should be accepted, rejected or is a don't care sequence, in the 3DFA to be learned. The membership queries are answered according to the full-control criterion. The membership query algorithm operates on the system model from which τ-transitions have been eliminated with the $\tau^* a \tau^*$-completion algorithm, and then completed on error.

The sequence σ to be evaluated by the membership query algorithm is simulated on the system model and there are three possible outcomes:

1. σ *may* lead to the error state: $MQ(\sigma) = \mathbf{F}$;

2. σ can be simulated entirely and *never* leads to an error state: $MQ(\sigma) = \mathbf{T}$;

3. σ cannot be simulated entirely: $MQ(\sigma) = \mathbf{DC}$.

Definition 5.15 (Membership Query)**.** *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, the* membership query *is defined for any $\sigma \in \mathcal{L}$ as:*

$$MQ(\sigma) = \begin{cases} \boldsymbol{F} & \Longleftrightarrow & \Pi \in (s_{0_S} \mathbf{\ after\ } \sigma) \\ \boldsymbol{T} & \Longleftrightarrow & \sigma \in \mathbf{Tr}(\mathcal{S}) \wedge \Pi \notin (s_{0_S} \mathbf{\ after\ } \sigma) \\ \boldsymbol{DC} & \Longleftrightarrow & \sigma \notin \mathbf{Tr}(\mathcal{S}) \end{cases}$$

where the **after** *operation is defined on $\mathcal{S}^\tau_{\rightarrow_c \Pi}$.*

Figure 5.28 shows a system model and illustrates how the membership queries are answered thanks to $\mathcal{M}^{\tau}_{\to_c \Pi}$ the model:

- $MQ(c_1 o_1 c_1) = \mathbf{F}$, since the $\langle c_1 o_1 c_1 \rangle$ sequence can lead to the error state Π.

- $MQ(c_1 o_1) = \mathbf{T}$, since it can be entirely simulated (that is, it belongs to the traces of the system) and the only state that can be reached is $\mathbf{C}$, which is not the error state.

- $MQ(c_1 o_2) = \mathbf{DC}$, since it is impossible to simulate it entirely.

(a) A system model $\mathcal{M}$. (b) The $\mathcal{M}^{\tau}_{\to_c \Pi}$ model.
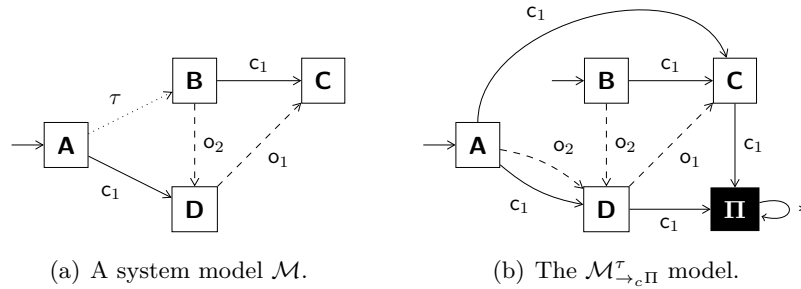
Figure 5.28. A system model example and its completion on commands that is used by the membership query algorithm.

Because the system model is not fc-deterministic in general, some trace may lead to different states. In particular, for the first case of the membership query algorithm, a given sequence may non-deterministically lead to the error state or not. In such situation, the membership query must answer $\mathbf{F}$ because any behaviour that may be harmful, even if it is not always the case, should not be present in the full-control conceptual model. Figure 5.29 shows an example of such a situation. The $\langle c_1 c_2 \rangle$ sequence leads to the $\mathbf{E}$ state, but can also reach the Π state through the $\mathbf{A} \xrightarrow{c_1} \mathbf{D} \xrightarrow{c_2} \Pi$ execution. Thus, the $MQ(c_1 c_2)$ query replies $\mathbf{F}$.

The proposed membership query algorithm replies exactly according to the 3DFA characterisation of the full-control conceptual model. It can thus be used to know how to classify the sequences that are asked by the learning algorithm.
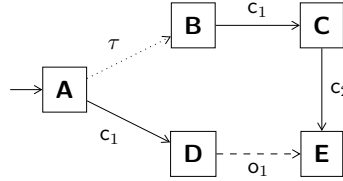
Figure 5.29. A non fc-deterministic system model for which $MQ(\mathsf{c_1}\mathsf{c_2}) = \mathbf{F}$.

Property 5.16. *Given an fc-deterministic HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o,$
$s_{0_S}, \rightarrow_S \rangle$ and its 3DFA characterisation $\mathcal{C}(\mathcal{S}) = \langle \Sigma, S, s_0, \delta, Acc, Rej,$
$DC \rangle$, MQ is consistent with $\mathcal{C}(\mathcal{S})$, that is, for $\sigma \in \mathcal{L}^*$:*

- $MQ(\sigma) = \boldsymbol{T} \iff \sigma \in Acc$
- $MQ(\sigma) = \boldsymbol{F} \iff \sigma \in Rej$
- $MQ(\sigma) = \boldsymbol{DC} \iff \sigma \in DC$

Proof. The proof follows from the 3DFA characterisation of a system
model described by Definition 5.7 on page 135. $\qquad\square$

Conjecture Check Algorithm

The *conjecture check* $(Conj)$ is used to test whether a given learned
3DFA candidate is acceptable or not. In the case that the candidate is
acceptable, the learning algorithm finishes and returns the candidate as
its result. If the candidate is not good, the conjecture check has to return
as a counterexample either a sequence that is in the candidate but should
not be part of it, or a sequence that is missing in the candidate. The
check that has to be performed is that all the DFAs that are consistent
with the 3DFA candidate allow full-control of the system model.

The conjecture check for $L^{\mathbf{fc}}$ follows the one used for the L^{sep} algo-
rithm presented in Section 5.5.2. The idea is to perform a model checking
on the synchronous parallel composition between some transformation
of the system model and the 3DFA candidate. If the error state can be
reached, the property fails to be satisfied and the trace leading to the
error state is output as a counterexample.

Let $\mathcal{C}(\mathcal{S}) = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$ be the 3DFA characterisation
of the system model, and $\mathcal{C}$ the 3DFA candidate to be analysed by

the conjecture check algorithm. The soundness check must ensure that $Acc \subseteq L(\mathcal{C}^-)$ and $L(\mathcal{C}^+) \subseteq \overline{Rej}$.

- A trace failing to satisfy the $Acc \subseteq L(\mathcal{C}^-)$ condition is one that belongs to Acc and do not belong to $L(\mathcal{C}^-)$. Such a trace can be found on the following model:

$$\mathcal{S}^\tau \parallel lts(\mathcal{C}^-)_{\to \Pi}$$

- A trace failing to satisfy the $L(\mathcal{C}^+) \subseteq \overline{Rej}$ condition is one that belongs to Rej and to $L(\mathcal{C}^+)$. Such a trace can be found on the following model:

$$\mathcal{S}^\tau_{\to_c \Pi} \parallel lts(\mathcal{C}^+)$$

Similarly as what is done for soundness, the completeness check also consists in verifying two conditions that are $L(\mathcal{C}^-) \subseteq Acc$ and $\overline{Rej} \subseteq L(\mathcal{C}^+)$.

- A trace failing to satisfy the $L(\mathcal{C}^-) \subseteq Acc$ condition is one that is not in Acc but belongs to $L(\mathcal{C}^-)$. Such a trace can be found on the following model:

$$det(\mathcal{S})_{\to \Pi} \parallel lts(\mathcal{C}^-)$$

- A trace failing to satisfy the $\overline{Rej} \subseteq L(\mathcal{C}^+)$ condition is one belonging to $\overline{Rej}$ but that is not in $L(\mathcal{C}^+)$. Such a trace can be found on the following model:

$$det(\mathcal{S})_{\to_o Sink} \parallel lts(\mathcal{C}^+)_{\to_{co} \Pi}$$

Table 5.1 summarises the four containment queries that have to be performed in order to check that a 3DFA candidate is sound and complete with respect to the Acc and Rej languages.
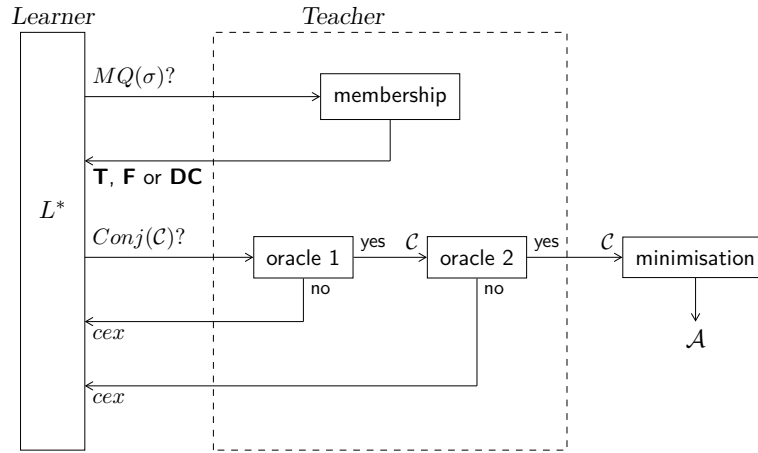
The $L^{\mathbf{fc}}$ algorithm

Given the membership query and the conjecture check algorithms just presented, Figure 5.30 shows the global overview of the $L^{\mathbf{fc}}$ algorithm,

Condition	Error trace	Containment query
$Acc \subseteq L(\mathcal{C}^-)$	$\sigma \in Rej$ and $\sigma \in L(\mathcal{C}^+)$	$\mathcal{S}^\tau \parallel lts(\mathcal{C}^-)_{\to_{co}\Pi}$
$L(\mathcal{C}^+) \subseteq \overline{Rej}$	$\sigma \in Acc$ and $\sigma \notin L(\mathcal{C}^-)$	$\mathcal{S}^\tau_{\to_c\Pi} \parallel lts(\mathcal{C}^+)$
$L(\mathcal{C}^-) \subseteq Acc$	$\sigma \notin Acc$ and $\sigma \in L(\mathcal{C}^-)$	$det(\mathcal{S})_{\to_{co}\Pi} \parallel lts(\mathcal{C}^-)$
$\overline{Rej} \subseteq L(\mathcal{C}^+)$	$\sigma \in \overline{Rej}$ and $\sigma \notin L(\mathcal{C}^+)$	$det(\mathcal{S})_{\to_o Sink} \parallel lts(\mathcal{C}^+)_{\to_{co}\Pi}$

Table 5.1. Summary of the containment queries that have to be performed in order to check whether a 3DFA candidate is sound and complete with respect to the *Acc* and *Rej* languages.

Figure 5.30. General overview of the learning framework used with the $L^{\mathbf{fc}}$ algorithm to learn a minimal full-control conceptual model for a given system model.

which is inspired from the L^{sep} algorithm of Chen et al. The difference with Chen et al. is that, for the $L^{\mathbf{fc}}$ algorithm, the completeness check is omitted. The two oracles correspond to the two conditions that have to be satisfied for the 3DFA candidate to be sound.

The first oracle corresponds to computing the $\mathcal{S}^\tau_{\to_c\Pi} \parallel lts(\mathcal{C}^+)$ model and to search for a trace reaching an error state. The second oracle checks the $\mathcal{S}^\tau \parallel lts(\mathcal{C}^-)_{\to_{co}\Pi}$ model and also search for a trace reaching an error state. In both cases, a composite state is considered as an error state if at least one of its component is the error state Π.

The main reason why the completeness check has been omitted is because of the high computational cost that may be necessary to determinise the system model. If the system model is fc-deterministic, the determinisation is not necessary anymore.

5.6 Comparison of the Generation Algorithms

This section compares the three generation algorithms proposed in this thesis. The generation algorithms are first compared regarding their performances. Their theoretical time complexities are compared as well as the effective time used to deal with a set of concrete examples. The second comparison presented is about how the three proposed algorithms handle situations where the input system model is not fc-deterministic.

5.6.1 Time Complexities and Execution Time

The three proposed algorithms have been run on the following seven different system models:

- **VTS** (Vehicle Transmission System) is a simple model of a semi-automatic transmission system of a large vehicle, proposed by Degani et al. [HD07]. That system model is described in Section 1.1 on page 4.

- **Therac-25** is a model of the Therac-25 medical device, taken from Bolton et al. [BBS08]. That system model is described in Section 2.4.5 on page 45.

- **FullAirConditionner** is a model of an air conditioner appliance, proposed in [CP09]. The model has been built from the description given in its user manual [Dan07]. The system has four operational modes: off, air cooling, humidity control and ventilation only. The user can cycle between those modes through the control panel of the appliance. For the air cooling mode (resp. humidity control mode), the user can select the target temperature (resp. humidity level) also through the control panel. The model has five different possible values for the temperature and humidity levels that the user can set.

- **AirConditionner** is a variant of **FullAirConditionner** with only two different possible values for the temperature and humidity levels that the user can set.

- **VCR** is a model of a Video-Cassette Recorder, that has been obtained from an ADEPT model, as proposed in [CGPF11a]. The model is described in Section 7.3.3 on page 7.3.3.

- **Countdown-2** is a model of a timer, presented in Section 3.3 on page 75, with a range of $N = 2$ values.

- **Countdown-12** is also a timer like **Countdown-2** except that it has a range of $N = 12$ values.

Table 5.2 summarises the results of the experiments. All the experiments have been run on the same machine, within the same conditions. The available memory has been limited to 4 Go and a time limit of 15 min has been set, for each test. The times that are reported in Table 5.2 are the total time needed to generate a minimal full-control conceptual model. The first observation that can be made is that no algorithm is uniformly better than the others in term of execution time. Depending on the particular example that is used, one algorithm can be more efficient than another one. For the seven tested examples, the learning-based algorithm is always the slowest.

Both the 3DFA-based and learning-based approaches have to minimise a 3DFA at some point. Since the minimisation algorithm has a high time complexity, it is better to have a 3DFA that is as small as possible. The **VCR** example shows that learning-based approach is better when the size learned 3DFA is small compared to the 3DFA-based approach, although the reduction-based approach still performs better. However, having a learned 3DFA that is smaller than the built 3DFA used by the 3DFA-based approach is not a guarantee that the learning-based approach will be faster as shown by the **Therac-25** example. For that example, the learning-based algorithm had to go through too many iterations which increases the total running time that is better for the 3DFA-based approach. The second observation that can be made is that when the system model is large, which means that the built 3DFA will also be large, and when a minimal full-control mental model is small, which means that the learned 3DFA will also be small, the learning-based approach tends to be faster than the 3DFA-based approach.

The reduction-based approach is most of the time faster than the two other approaches. The first reason is that the algorithm has been run without the full exploration of all possible minimal full-control conceptual models. It was indeed able to find directly the minimal full-control conceptual model. The only example where the reduction-based approach is not the fastest is the **Therac-25** example. The third observation that can be made is that the reduction-based approach scales better for large models, as shown with the **FullAirConditionner** and **AirConditionner** examples where the two other approaches was not able to provide a result within the fixed time and memory constraints.

5.6.2 Non-fc-deterministic System Models

The different algorithms do react differently whenever they are provided with a system model that is not fc-deterministic. Of course, one possible solution is to perform the fc-determinism check on the system model before running the algorithms, but as discussed in this section, that additional computation may be avoided in some situations.

Figure 5.31 shows a small example of a system model that is not fc-deterministic and that is used in this section to illustrate the difference between the three generation algorithms.
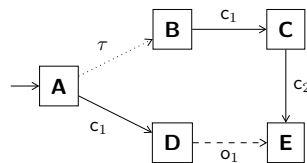
Figure 5.31. A system model example that is not fc-deterministic due to the trace $\langle c_1 \rangle$ that can lead to both state **C** or **D** with $A^c(\textbf{C}) = \{c_2\} \neq A^c(\textbf{D}) = \emptyset$.

3DFA-based Generation Algorithm

The 3DFA-based generation algorithm requires the system model to be fc-deterministic in order to correctly compute a minimal full-control conceptual model. If that constraint is not satisfied, the algorithm still is able to generate a reduced model, but that latter does not necessarily allows full-control of the system model. Figure 5.32 shows the models obtained after each step of the 3DFA-based algorithm.

System	Conceptual	3DFA		Reduction	Learning	
(States / Trans.)	(States / Trans.)	(3DFA states)	(Total)	(3DFA states)	(Total)	
VTS	8 / 20	5 / 14	10	26 ms	25 ms	10	75 ms
Therac-25	136 / 448	24 / 83	138	63 ms	375 ms	70	1,920 ms
FullAirConditionner	194,400 / 650,880	1,080 / 4,500	194,402	—	49,155 ms	—	—
AirConditionner	13,608 / 44,748	222 / 834	13,610	—	2,684 ms	—	—
VCR	3,352 / 15,082	2 / 9	3,354	22,595 ms	479 ms	6	988 ms
Countdown-2	11 / 27	8 / 18	10	33 ms	27 ms	10	155 ms
Countdown-12	55 / 137	30 / 73	32	113 ms	54 ms	32	3,576 ms

Table 5.2. Experimental results of the comparison of the three proposed full-control conceptual model generation algorithms.

(a) 3NFA-completion.

(b) Determinisation.

(c) Minimisation.
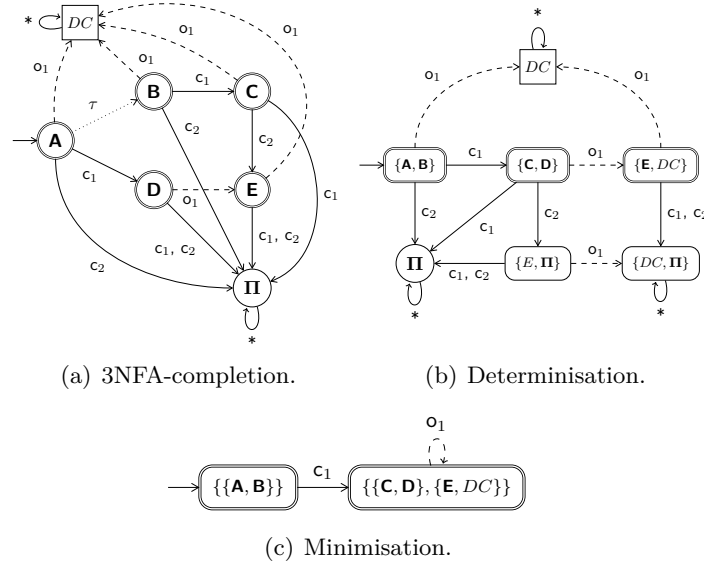
Figure 5.32. The three intermediate models built by the 3DFA-based algorithm for the example of Figure 5.31.

The non-fc-determinism in the system model example is due to the $\langle c_1, c_2 \rangle$ sequence which is at the same time accepted and rejected, as clearly identifiable on the 3NFA-completion by the two $\mathbf{A} \xrightarrow{\tau} \mathbf{B} \xrightarrow{c_1} \mathbf{C} \xrightarrow{c_2} \mathbf{E}$ and $\mathbf{A} \xrightarrow{c_1} \mathbf{D} \xrightarrow{c_2} \Pi$ executions. The situation can also be identified on the determinisation since the $\langle c_1, c_2 \rangle$ leads to the $(\mathbf{E}, \Pi)$ state, $\mathbf{E}$ being an accepting state and Π the error state. By identifying such composite states, containing the Π state, it is possible to detect whether the system model is not fc-deterministic.

As stated earlier, the HMI-LTS corresponding to the obtained DFA does not allow full-control of the system model, but the model that is generated still brings some interesting information about the system model. It represents, in some extent, the fc-controllable part of the system model.

Reduction-based Generation Algorithm

The reduction-based generation algorithm starts with a $\tau^*a\tau^*$-completion and then computes an initial partition, based on the possible commands.

The initial partition for the example of Figure 5.31 consists of three blocks as shown on Figure 5.33.

The B_1 block is not fc-stable according to the c_1 action, since triggering it from B_1 can lead to the two different blocks B_2 and B_3. The algorithm will thus try to split B_1, but it will face a difficulty since the state **A** must go in the two newly created blocks. When such a situation occurs, the algorithm just stops and produces an error message indicating that block B_1 must be split due to action c_1 for which the two following transitions are problematic: $\mathbf{A} \xrightarrow{\,c_1\,} B_2$ and $\mathbf{A} \xrightarrow{\,c_1\,} B_3$.
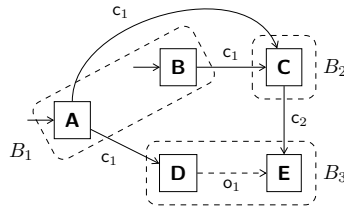
Figure 5.33. The initial partition for the reduction-based generation algorithm for the HMI-LTS system model example of Figure 5.31.

The reduction-based generation algorithm is thus capable of detecting non-fc-determinism issues, which is detected when a command from the same state s leads to different blocks. From that, useful diagnostic information can be provided by the algorithm, by finding an execution starting from the initial state and reaching the problematic state s.

Learning-based Generation Algorithm

The learning-based generation algorithm starts by a querying a set of traces about their membership. The initial observation table (S, E, T) is built by choosing $S = E = \{\lambda\}$. Figure 5.34 shows the system model which has been τ-completed and for which an error state has been added, to answer membership queries. From that, the learning algorithm computes the first closed and consistent observation table and derives from it the first 3DFA candidate.

The 3DFA candidate is then verified by the conjecture check algorithm. The first oracle fails and outputs the $cex = \langle c_1, c_1 \rangle$ sequence as a counterexample. That trace is accepted by the 3DFA candidate, but is

(a) Error completion.

(b) Observation table.
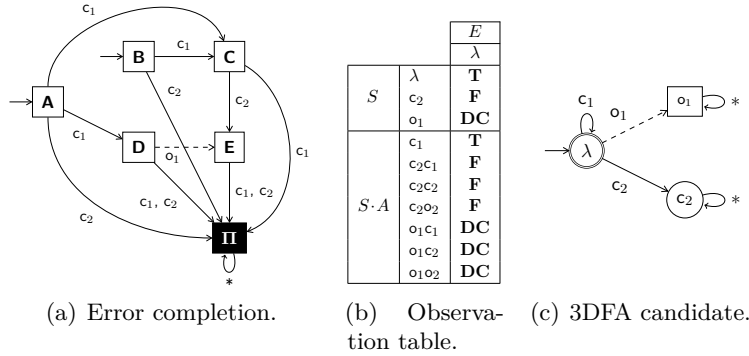
(c) 3DFA candidate.

Figure 5.34. First iteration of the learning-based generation algorithm for the system model example of Figure 5.31.

rejected by the system model (that is, $cex \in Rej$ and $cex \in L(\mathcal{C}^+)$). Figure 5.35 shows the two next 3DFA candidates produced by the algorithm. The one of Figure 5.35(a) still does not satisfy the conjecture check as the second oracle fails with the $cex = \langle c_1, o_1, o_1 \rangle$ counterexample, which is accepted by the 3DFA candidate, but is in DC in the system model (that is, $cex \notin Acc$ and $cex \in L(\mathcal{C}^-)$). Finally the last 3DFA candidate produced does satisfy the conjecture check, which allows the learning algorithm to produce a minimal consistent DFA as a result.

(a) Second 3DFA candidate.

(b) Last 3DFA candidate.

(c) Minimal consistent DFA.
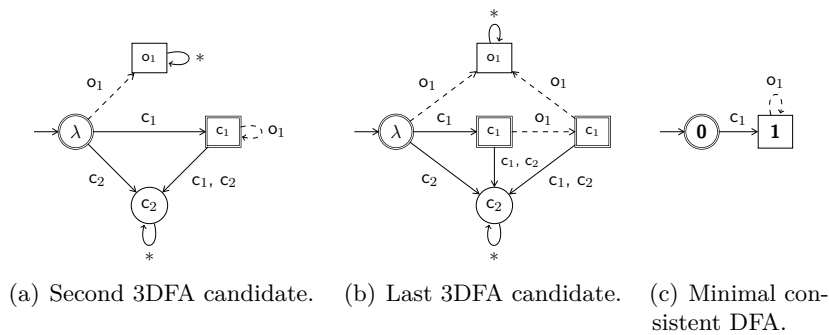
Figure 5.35. Successive 3DFA candidates produced by the $L^{\mathbf{fc}}$ algorithm for the system model example of Figure 5.31.

The obtained DFA does not allow full-control of the system model, and it corresponds in fact to the DFA generated by the 3DFA-based algorithm (see Figure 5.32(c)).

Chapter 6

HMI Analysis

This chapter presents how the concepts and techniques proposed in the previous chapters can be used in practice to perform analyses of human-machine interactions. Section 6.1 presents the `jpf-hmi` tool that has been developed to support the experiments. Section 6.2 presents how training materials for a given system can be analysed, for example to compare different sources of training. Section 6.3 describes how the full-control conceptual model generation algorithms can be used in order to analyse a given system and assess its controllability. Section 6.4 presents mode confusion potential and how detecting it can be directly handled with the algorithms proposed in the previous chapters. Section 6.5 presents a variant of the full-control property and how user task models can be taken into account and integrated in the proposed framework.

In the remainder of this chapter, *generation algorithm* will be used to refer to any of the minimal full-control conceptual model generation algorithms presented in the previous chapter.

6.1 The `jpf-hmi` Tool

As already introduced and highlighted in [Hol97], one crucial point to make formal methods based analyses used by people from other fields is to have a usable tool. The tool should be designed so that performing analyses do not require strong specific knowledge of formal methods, and so that obtained results make sense at the domain level for the user of the tool. Such a tool as not been developed in this work, although a prototype tool has been developed to support experiments. That prototype is the embryo of a tool that could be used by system designers and analysts, to perform human-machine interactions analyses.

6.1.1 Structure of the Tool

Figure 6.1 shows the general overview of the jpf-hmi tool. The top left part represents the input part, that is, how HMI-LTS models are imported in the tool. The bottom part represents the different analysis that can be performed. Finally, the right part shows the only way HMI-LTSs can be exported out of the tool. This section provides a general overview of the tool and focuses on the modelling part.

The jpf-hmi tool is an extension of the *Java Pathfinder* model checker (JPF) [VHBP00], and provides support for the algorithms developed in this thesis. For now, all the capabilities that are offered by the model checker have not been used in the current version of the prototype tool. Nevertheless, developing the proposed tool closed to a model checker from the start is motivated by the fact that most of the proposed algorithms are amenable to run a model checking algorithm.
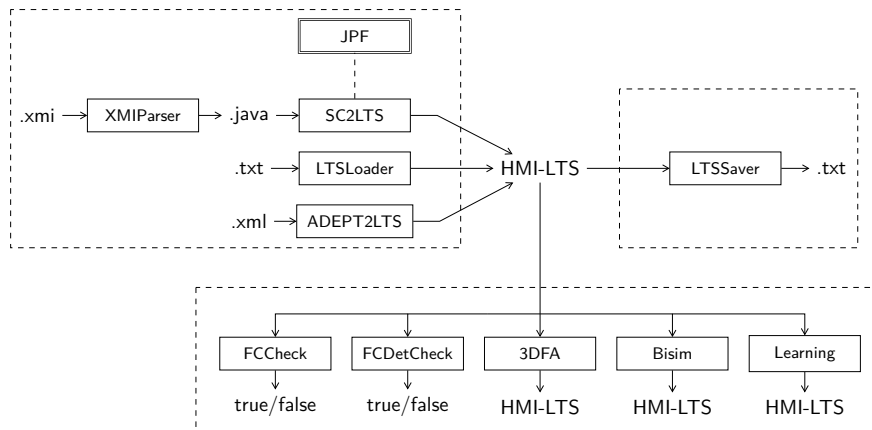
Figure 6.1. Overview of the jpf-hmi extension.

HMI-LTSs, representing system or mental models, can be imported into the tool by three different means:

- First of all, models can be imported from statecharts. Those can be described within the XMI standardised format, and then imported and translated into a Java program representing the statechart. The Java program is in fact described following the conventions of the jpf-statechart extension [Meh08]. Then, JPF is

used to explore entirely the behaviour of the statechart, so that to build and generate an HMI-LTS corresponding to the statechart.

- Model can also be imported from a raw text file, where the models are encoded with a non-standard simple format. This way to proceed is useful for the tool to be able to import files that have been exported from the tool.

- Finally, models can be obtained from ADEPT models. A precise semantics for those models and the translation algorithm are described in Chapter 7.

Of course any design language for which there exists or for which it is possible to define a semantics based on HMI-LTS can be used with the proposed tool. It suffices to write a translator from the design language towards HMI-LTS in order to get access to the proposed algorithms. However, in order to get a full support of a design language, more work has to be done in order to be able to explain the results provided by the algorithms, such as counterexample traces, back into the design languages.

As highlighted in the introduction about formal methods, it is important to provide a good tool support in order to make formal methods based techniques accepted by the users. This jpf-hmi extension is a first step towards such a good support, but as it is described in the perspectives in Chapter 8, work has still to be done for the tool support which is not the main focus of this thesis.

6.2 Analysis of Training Material

When a system designer thinks about the design of a system, he has in his mind a conceptual model of the system, referred to as the design model. The system designer has to communicate his design model to the user to allow them to properly use the system. One communication mean that is used between the system designers and the users are the user training manuals [BCC⁺11, Thi96]. More generally, the user has many other sources to learn how to operate a system and he will typically combine all those different sources to build his own mental model. Classical sources of information are user training manuals or training sessions

with possible demonstrations. Other important sources includes self-explanatory systems, such as ATMs, or knowledge transferred from similar systems. For example, a pilot used to fly with Airbus airplanes will transfer his knowledge about the autopilot when first flying a Boeing airplane.

The purpose of training material is to capture the behaviour of a system, or a part of it, and to communicate it to an operator so as to contribute to his mental model. It is therefore important that that training material satisfies some properties with respect to the system it describes. The remainder of the section will focus on user training manuals, simply referred to as *training manual* from now on. The analysis done can of course easily be extended to other sources of training as far as they can be described formally.

The full-control property can be used in order to assess whether a given training manual is complete enough regarding a given system. The idea is quite straightforward and consists in checking the training manual for full-control against the system model, using the full-control check algorithm 5 (presented on page 249). For such an analysis to be possible, the training manual has to be modelled as an HMI-LTS. Several techniques exist to (semi-)automatically get a formal model from a training manual. For example, [DLDvL05] proposes an algorithm to automatically generate an LTS from simple forms of *message sequence charts* (MSC) representing scenarios described in the training manual. The algorithm interactively presents MSCs to the end-user and asks him to classify them as examples or counterexamples of the desired behaviour. From those answers, the algorithm builds a corresponding LTS, representing the training manual.

When the full-control property is not satisfied, the full-control check algorithms provides a counterexample which is a trace whose execution can lead to a pair of states failing to satisfy the full-control criterion. When applied between a training manual and a system model, that counterexample can be interpreted as follows:

- The training manual describes a command corresponding to a functionality that is not available to the user on the system in a state. That situation can surprise the user since after performing the command, he will expect a reaction from the system that will never occur since the command is not available. One possible reason

for that possible confusion is that the training manual improperly describes conditions under which a command is applicable.

- A functionality of the system is not covered by the training manual. That situation is less problematic unless the goal of the training manual is to provide to the user a description of all the functionalities of the system. Section 6.5.2 presents a variant of the full-control property that is more flexible and does not require the training material to cover all the possible commands during the interaction.

- There are some observations that are possible on the system but are not described in the training manual. If such an observation occurs during the interaction, the user will be surprised since it was not described in the training manual. Such a situation can be very harmful when the signal sent by the system is an hazard alarm, for example.

If the model obtained from the training manual does not allow full-control of the system model, it means that potential automation surprises may occur. Either the training material has to be revised by adding necessary warnings for forbidden behaviours and completing it for missing observations, or the system has to redesigned and the training manual regenerated.

6.2.1 The Microwave Oven Example

Figure 6.2 shows a statechart of the mode part of the behaviour of a microwave oven. At a high level, the system is composed of two modes: disabled and operational. The system alternates between those two modes when the door is opened or closed. In the operational mode, the user can set the time of the system and can cook, in which case either he programs a duration or he asks for a one-minute cook. At any time, the user can press the stop button or open the door to stop the cooking. Whenever the system exits the operational mode when the user opens the door, whenever he closes the door, the system goes back in the same substate of the operational mode it was in. It is due to the fact that the operational state is an history state, as highlighted by the Ⓗ lying on the top right corner of the state.
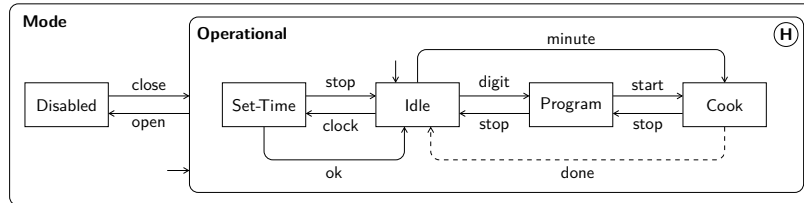
Figure 6.2. A statechart representing the mode part of the behaviour of a microwave oven (taken from [Luc93]).

As described in Section 3.4.3, statecharts can be converted into HMI-LTS. Figure 6.3 shows the translation of the microwave oven statechart into an HMI-LTS. The two main modes (disabled and operational) are easily identifiable on this HMI-LTS.
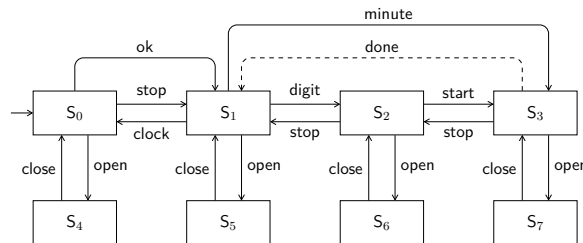

Figure 6.3. Translation of the microwave oven statechart into an HMI-LTS representing the same behaviour.

The description of the microwave oven example [1] contains a textual description of the cooking and time setting modes. In particular, for the cooking mode, it is stated that:

> "1. To cook, enter the amount of time digit-by-digit then press **Start**. As digits are entered, they are 'shifted' to the left; if more than four digits are entered, the left-most digit is 'shifted off' when a new digit is entered.
>
> 2. To set a power-level lower than 100%, press **Power** followed by the first digit of the percentage before pressing **Start**.

---

[1] http://chsm.sourceforge.net/examples/microwave/.

3. Alternatively, to cook for one minute at full power, or to add one minute to the remaining time, press **Minute**.

4. To stop cooking, press **Stop** or open the door; to continue cooking, close the door if it was opened, and press **Start**. To cancel cooking once stopped with the door closed, press **Stop** again."

The second rule is not relevant for the part of the model that has been chosen here. From that training manual, thus ignoring the second rule, it is possible to build an HMI-LTS representing the behaviour that is described. The adequacy of the training manual can then be assessed with the full-control check algorithm. Figure 6.4 shows a possible HMI-LTS corresponding to the partial training manual presented above.
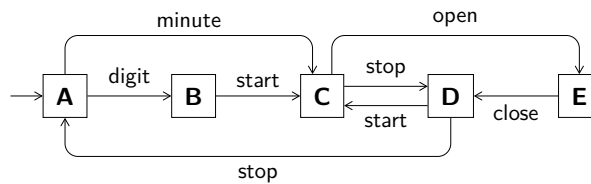
Figure 6.4. An HMI-LTS representing the behaviour described by the training manual of the microwave oven example, for the part related to cooking operations.

Checking the HMI-LTS representing the training manual against the system model, with the Set-Time state removed since that part is not covered by the training material, fails. The error trace that is produced as a counterexample is the empty trace, with the following condition that fails to be satisfied: $A^c(\mathsf{Idle}) = \{\mathsf{open}, \mathsf{digit}, \mathsf{minute}\} \neq A^c(\mathbf{A}) = \{\mathsf{digit}, \mathsf{minute}\}$. Indeed, the fragment of training manual presented above does not say anything about the possibility to open the door before doing anything else with the microwave oven. In fact, the opening of the door is described in another place of the training manual where it is possible to read:

"If the door is opened at any time during either sequence, the oven is disabled; closing the door resumes the sequence from where it left off."

The HMI-LTS corresponding to the training manual has to be adapted consequently. A new state s' is added for each state s along with the

two transitions $s \xrightarrow{\text{open}} s' \xrightarrow{\text{close}} s$, except for **E** where the door is already opened and **C** where the open command is already handled.

Checking that new model of the training manual still fails, this time with the sequence $\langle \text{minute} \rangle$ as a counterexample and with the following condition that fails to be satisfied: $A^o(\text{Cook}) = \{\text{done}\} \not\subseteq A^o(\textbf{C}) = \{\}$. The issue is that the training manual says nothing about the observation done that can occur in the system, when the cooking is finished. To solve that issue, the training manual must say that whenever the cooking is finished, a **Done** signal is produced by the microwave oven, resetting it to its initial state. That will induce a new transition $\textbf{C} \xrightarrow{\text{done}} \textbf{A}$ in the training manual.

The full-control property is still not satisfied with the last modifications. The provided counterexample is the sequence $\langle \text{digit} \rangle$ which leads to the following situation: $A^c(\text{Program}) = \{\text{open}, \text{start}, \text{stop}\} \neq A^c(\textbf{B}) = \{\text{open}, \text{start}\}$. The solution is to add the following transition to the training manual: $\textbf{B} \xrightarrow{\text{stop}} \textbf{A}$. Figure 6.5 shows the latest version of the HMI-LTS representing the training manual.
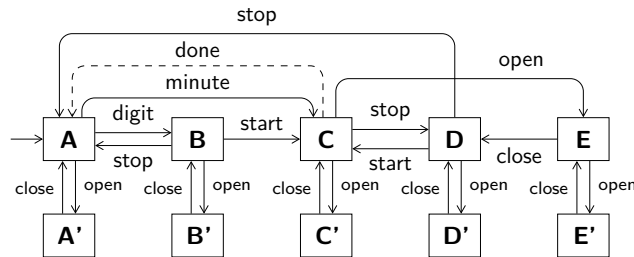
Figure 6.5. An HMI-LTS representing the behaviour of a training manual that allows full-control of the system, for the microwave oven example.

Checking that new training manual again produces an error. This time, the bad sequence $\langle \text{minute}, \text{open}, \text{close} \rangle$ can lead to the following bad situation: $A^c(\text{Cook}) = \{\text{open}, \text{stop}\} \neq A^c(\textbf{C}) = \{\text{open}, \text{start}, \text{stop}\}$. This time, the highlighted situation does not correspond to a missing information in the training manual. In fact, it does correspond to a faulty scenario described in the training manual. According to the training manual, if the cooking is interrupted because the door was opened, once closed, the cooking can be resumed by pressing **Start**. However, in the actual system, the cooking will resume automatically, without the

need to press any button. This is why the `start` command is possible according to the training manual but not in the system, in the presented counterexample.

This simple microwave oven example demonstrates how the full-control property with its check algorithm can be used to assess the adequacy of training manual with respect to a given system. The methodology illustrated here above shows how it is possible to use the information contained in the counterexample to modify the training manual, in order to make it adequate for the described system. Of course, another possibility would have been to modify the design of the system. Finally, as already stated before, using the full-control property for the proposed analysis enforces the training manual to cover completely all the behaviour of the system. Section 6.5.2 presents a variant of the full-control property to overcome that limitation and to be able to deal with training manuals that only cover part of the behaviour of a given system.

6.3 System Analysis

The previous section describes how a training manual can be checked against the system it describes. Generally, if issues are found, corrections will be done in the training manual by correcting it and adding warnings for potential dangerous situations. The reason is that the training manual is only available late in the design process. The design process of a new system in not a linear process. It involves successive iterations in which a test and/or verification phase is performed to check whether the designed system meets the specifications and some other properties such as usability properties or performance issues, for example. That phase is generally referred to as the *validation phase.*
The first stage is to model the desired system, either directly as HMI-LTS or through higher level formalisms such as statecharts or ADEPT models. From that, the candidate designed system is verified to satisfy the fc-determinism property, by going through a generation algorithm. Either the verification succeeds, in which case a minimal full-control conceptual model is produced, which can be used to generate training material, or a counterexample witnessing an issue with the system is

produced which drives the process into the analysis phase. After having
analysed the issue, solutions have to be found to overcome it. The
redesign step tries several modification such as adding observations or
eliminating non-determinism in the system. And those changes have to
be modelled so that a new iteration of the process can start.

The generation algorithms presented in the previous chapter can be
used during the validation phase to ensure that no potential automation
surprises can occur during an interaction between an operator and the
system. Running a generation algorithm on the system can either succeed,
in which case a minimal full-control conceptual model is produced, or it
can fail, which results in the production of a counterexample illustrating
why the generation failed and where there is a potential issue in the
design of the system.

6.3.1 Non-fc-deterministic System Model

If the generation algorithm fails, it is due to the fact that the system is
not fc-deterministic. As a reminder, it is impossible to build a full-control
conceptual model for a system if there exist situations where the operator
can not know what commands are possible on the system after a given
execution. More formally, there exists a trace σ such that $s_0 \overset{\sigma}{\Longrightarrow} s$ and
$s_0 \overset{\sigma}{\Longrightarrow} s'$ with $A^c(s) \neq A^c(s')$.

In term of interactions, it means that after the execution of a given
trace, there is an uncertainty about the possibility to execute some
commands. The operator cannot firmly believe that a command is
possible, and if he tries to execute it, expecting some reaction from
the system, he can be surprised since it is possible that nothing will
happen. Of course, if the operator is well-trained and made aware of
that uncertainty, he can be driving the system safely, by avoiding to
use a command for which there is uncertainty. That possibility is not
considered in this work and left for future work.

The following example, presented on Figure 6.6, illustrates that
situation. The example represents part of the behaviour of a TV decoder.
The decoder is initially shut down (state **A**) and pressing the on button
turns the decoder on. If the decoder manages to get connected to the
internet (state **C**), the user is granted the possibility to select the first
channel by pressing the ch_1 button to play it (state **D**). If the internet
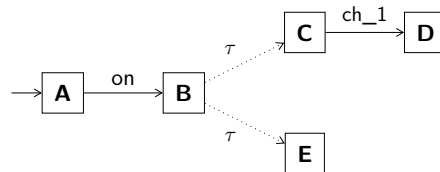connection is not available, the user cannot do anything (state **E**).

Figure 6.6. The system model of a small example representing a part of the behaviour of a TV decoder.

Running a generation algorithm on the TV decoder example produces the following trace as a counterexample: ⟨on⟩. This trace leads to several states having different sets of possible commands, which is a violation of the fc-determinism criterion. The issue is that after performing an execution corresponding to the ⟨on⟩ trace, the operator must be able to distinguish states **B**, **C** and **D**. A direct solution to redesign the system is thus to provide a way for the operator to distinguish both states, with an observation.

Figure 6.7 shows a possible redesign of the system. After pressing the on button, if the decoder manages to access the internet, an online signal is produced with a sound or with a led starting to blink, for example. That observation can be used by the operator to know if he can launch the first channel or not.
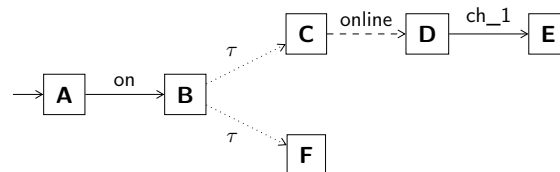
Figure 6.7. A possible adaptation of the system model of the TV decoder of Figure 6.6 which solves the non-fc-determinism issue.

Solving a non-fc-determinism issue can always be done simply by adding new observations to the model, that can be used by the operator to distinguish the different states that can be reached with the same trace, and having different sets of possible commands. But the additional cost of doing so is an increase of the complexity of the system and the number of observations that the operator must manage.

6.3.2 Minimal Full-control Conceptual Model

If the system model is indeed fc-deterministic, the generation algorithm does produce a minimal full-control conceptual model. In terms of interaction, it means that in theory, that is, if the operator follows exactly the produced conceptual model while using the system, and never misses any observation, it is possible for him to never get confused during the interaction, in the sense of the full-control property.

In this case, it is still possible to get information from the produced conceptual model, for the validation phase. The number of states of the produced conceptual model gives a direct feedback about the complexity of controlling the system. Indeed, the larger the conceptual model, the harder it is for a human operator to memorise and use. The number of states of the produced conceptual model can be used as a metric for the system model. Such a metric can serve to choose between several different designs for the same system.

Generating Training Manual

Moreover, the generated conceptual model can serve the production of training material, following for example approaches such as [TL96, ML03]. In order to generate a training manual that is as concise as possible, it is required to get the smallest possible conceptual model for the system. To illustrate that, let us take again the microwave oven example from Section 6.2.1. The system can be modelled as an HVS described with two state-variables: mode whose value is either *disabled* or *operational* and opstate whose value is one of *settime*, *idle*, *program* or *cook*. Without any visible state-variable, the generation algorithms produce a conceptual model with eight states. The four states corresponding to the operational states are all duplicated, so as to have one corresponding to the disabled mode and one to the operational mode, as shown on Figure 6.8. It indeed corresponds to the system model since nothing can be reduced.

But, the mode state-variable is by nature visible, since it corresponds to the door of the microwave oven being opened or not. Making only that state-variable visible reduces the size of the produced minimal full-control conceptual model, by dividing its number of states by two. The close command is only possible if the door is open, and all the other commands are only possible if the door is closed. Figure 6.9 shows the reduced full-control conceptual model.
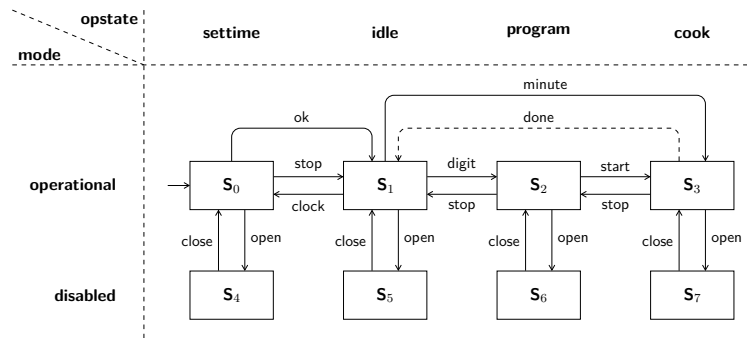
Figure 6.8. Generated conceptual model for the microwave oven example, considering that no state-variable are visible.
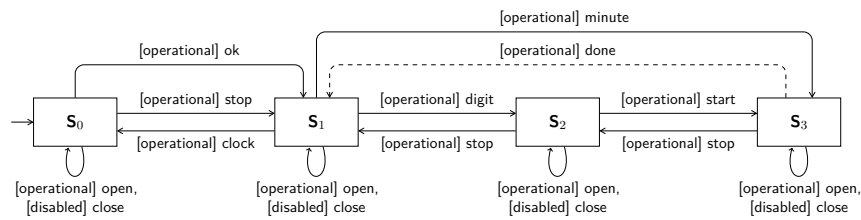

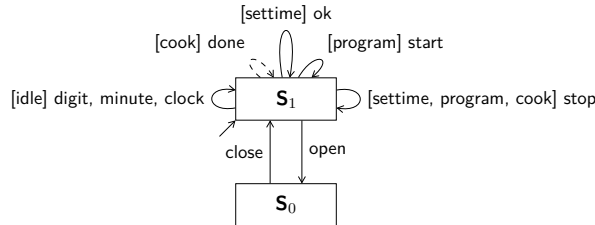**Figure 6.9.** Generated full-control conceptual model for the microwave oven example, considering the mode state-variable as visible.

By making the opstate state-variable visible, it is also possible to reduce the system model to an even smaller full-control conceptual model, shown on Figure 6.10. The system model is reduced to a two-state model, one for each value of the mode state-variable. The state S_0 corresponds to the door being opened, the only possible command being to close it. The state S_1 corresponds to the operational mode, and all the commands are guarded by the values of the opstate state-variable corresponding to the states of the system where they are possible.

By choosing carefully which state-variables are visible it is possible to reduce the number of states of the generated conceptual model. However, finding the subset of state-variables to make visible in order to minimise the number of states of the generated conceptual model is not immediate.

Figure 6.10. Generated full-control conceptual model for the microwave oven example, considering the opstate state-variable as visible.

6.4 Mode Confusion Analysis

As previously stated, mode confusion potential during human-machine interaction can be harmful, as testified by many accidents that occurred due to a mode confusion. This work is focused on the more general automation surprises potential and on controllability issues, but it is straightforward to handle mode confusion. Information about modes can be added to HMI-LTSs, as well as in the enriched models, by defining a set of distinct modes and by partitioning the states of the system so that each state is assigned a mode. A *mode assignment function* can be associated to any of the proposed models.

Definition 6.1 (Mode assignment function)**.** *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and a finite set of modes $\mathcal{L}^m$, the modes of the states of the system are represented by the* mode-assignment function $\mu_S : S_S \mapsto \mathcal{L}^m$.

For an operator not to be confused with respect to modes, he must be able to track them. At any time during the interaction, the user must know the current mode of the system and he must be able to predict the mode the system will transition into for any action that takes place on the system. By having the mode information on the states of the system, and by also labelling the states of the mental model with mode information, it is possible to identify potential mode confusion with the interaction model. Indeed, any composite state where the modes according to the system model and the mental model do not coincide exhibits a potential mode confusion.

Information about modes may have been added as a state-variable for the mode. Of course, such a state-variable should not be visible to the operator, since if it was, the operator will be able to track the mode and there will not be any potential mode confusion situations. Mode confusion is only relevant if information about the modes is not directly available to the operator. Moreover, with the enriched models, state-variables can not be used on the mental models since there is no state-value on HVM. That latter fact would make it impossible to check, with the composite states of the interaction model, whether there are any discrepancy between the modes of the system and the mental models. For all those reasons, and to highlight the fact that information about modes is distinct from visible state-values, they have been modelled separately.

6.4.1 Generating Minimal Mode-preserving Conceptual Model

Given a system model, with a mode assignment function, the techniques presented in this work can be slightly adapted in order to automatically generate conceptual models that guarantees that, if followed exactly by an operator, avoid potential mode confusion situations. The *mode-preserving* property captures that notion by requiring that for any state of the interaction model, the mode the system is into is the same that the mode expected by the operator.

Definition 6.2 (Mode-Preserving Property for HMI-LTS)**.** *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, a deterministic and non-divergent HMI-LTS $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, a set of modes $\mathcal{L}^m$ and the two mode assignment functions $\mu_S : S_S \mapsto \mathcal{L}^m$ and $\mu_H : S_H \mapsto \mathcal{L}^m$, $\mathcal{H}$ is said to be* mode-preserving *with respect to $\mathcal{S}$, which is denoted $\mathcal{H} \, \mathbf{mp} \, \mathcal{S}$, if and only if for all $\sigma \in \mathcal{L}^*$ such that $s_M \in (s_{0_M} \, \mathbf{after} \, \sigma)$ and $s_H \in (s_{0_H} \, \mathbf{after} \, \sigma)$:*

$$\mu(s_M) = \mu(s_H)$$

Analysing mode confusion potential is quite straightforward. The direct way to do that is to modify the system model by adding one new command for each different mode, and by adding to each state of the system a loop command labelled with its corresponding mode. By doing so, since the modes are considered as commands, it will be ensured that

two behaviours that differ according to mode will be clearly separated in
the generated conceptual model.

Definition 6.3 (Mode completion)**.** *Given an HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o,$
$s_{0_S}, \rightarrow_S \rangle$, *a finite set of modes* $\mathcal{L}^m$ *and a mode-assignment function*
$\mu_S : S_S \mapsto \mathcal{L}^m$, *the* mode-completion *of* $\mathcal{S}$, *denoted* $\mathcal{S}_{\circlearrowleft_m}$, *is the HMI-*
LTS $\langle S_S, \mathcal{L}^c \cup \mathcal{L}^m, \mathcal{L}^o, s_{0_S}, \rightarrow_S \cup \{(s, \mu_S(s), s) \mid s \in S_S\} \rangle$.

Figure 6.11 recalls a system model in statechart for the Therac-25
medical device that has been introduced in Chapter 2 (page 45). The
accident that occurred with that device was precisely due to a mode
confusion. Patients were administered lethal doses of radiations because
operators thought that the machine was in the electron beam mode
whereas it was in the X-ray one.
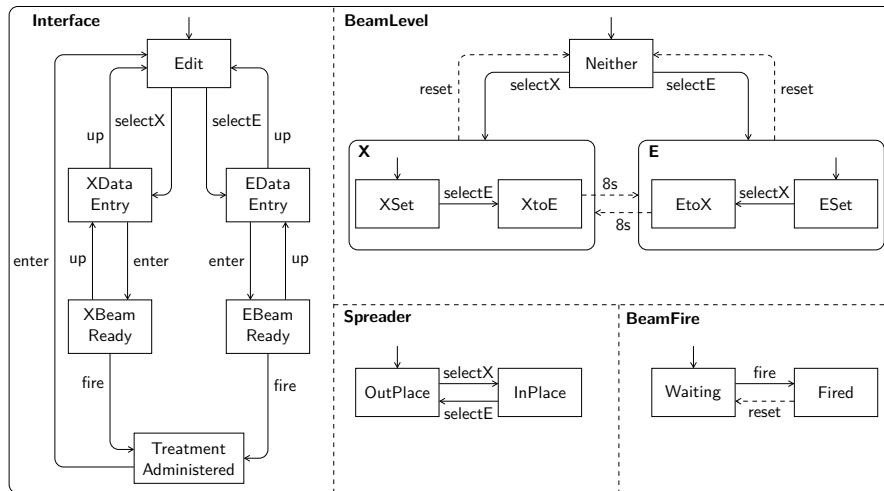
Figure 6.11. A model for the Therac-25 medical device (taken from [BBS08]).

In order to analyse the potential mode confusions for the Therac-25,
the device can be viewed as a three-mode system defined by the level of
the beam. Three commands are thus added to the model: neither, X-ray
and E-beam, corresponding to the three substates of the BeamLevel state
as shown on Figure 6.11, and the system is mode-completed according
to those modes. The system has a total of 68 states and 244 transitions

among which 180 are commands, 28 are observations and 36 are τ-transitions.

Executing a generation algorithm on this mode-completed model fails because the fc-determinism property is not satisfied. The following error trace is produced:

⟨selectX, enter, fire⟩

Figure 6.12 shows the error trace, with the situation that causes the fc-determinism issue. After executing the counterexample, both states D and E can be reached, and since they do not have the same sets of possible commands because of differing modes X-ray and neither, it is a violation of the fc-determinism property. The difference of possible commands is related to commands indicating the modes of the system, which means that the issue is related to a potential mode confusion.
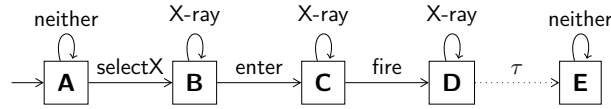
Figure 6.12. The counterexample witnessing the fc-determinism issue ends by an action indicating a mode which means that the counterexample highlights a potential mode confusion.

The issue is caused by the τ-transition that corresponds to the reset transition that occurs automatically once the beam has been fired. The operator may get confused since he cannot know for sure whether the beam level is set to X-ray or is unset. An easy and immediate solution to solve the issue is to make reset visible by turning it into an observation. Doing so, the system becomes fc-deterministic and the generated conceptual model has 12 states and 47 transitions.

The proposed approach can also be used to discover the potential mode confusion, in the same model, that caused several accidents and described in [LT93, BBS08]. The issue was caused by the fact that the eight seconds delay was not tracked by the operator for who it acts as an internal transition. Turning the 8s observation into a τ-transition makes the generation algorithm fail with the following counterexample:

⟨selectX, up, selectE, up, selectX⟩

This time, as illustrated by Figure 6.13, the issue is related to the 8s transition that will occur after the first correction and selection of electron beam. If the operator asks too quickly for the X-ray after the second correction, the beam level will go from state XtoE to ESet after he pressed selectX. So, after executing the counterexample, the system can both reach a state where the command X-ray is possible, and one where the command E-beam is possible, which is a potential mode confusion situation.
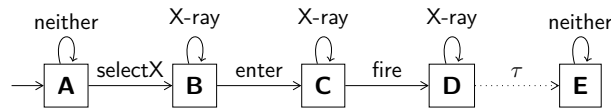

Figure 6.13. The counterexample witnessing the the well-known mode confusion that caused several accidents with the Therac-25.

The mode-completion is not necessary for the reduction-based generation algorithm. In fact, it suffices to take into account the modes when computing the initial partition that is used in the initial step of the algorithm. The algorithm is initialised with the coarsest partition such that all the states belonging to one block belongs to the same mode.

Identifying potential mode confusion situations is straightforward with the proposed algorithms. They do not need any adaptation, only the system has to be modified to integrate information about modes. The only element that has to be changed is the diagnostic information provided if the generation algorithm fails, so as to let the designer know if the issue is about potential mode confusion or a more general issue related to full-controllability. But as illustrated by the example, it can be easily systematised.

One limitation, that is not due to the methodology itself, is that each state of the system has only one mode. It could be useful to allow one state to have more than one mode, or to have a notion of "don't care". Such a possibility could have been useful for the first potential mode confusion highlighted for the Therac-25 device. If the state **E** of Figure 6.12 would have been categorised as a "don't care" or as in both modes X-ray and E-beam, the situation would not have been a potential mode confusion. That possible extension have not been considered in this work and left for future work.

6.4.2 Discovering Fc-modes

Another byproduct which is only provided by the reduction-based generation algorithm is that there is a direct relation between the states of the system model and those of the produced conceptual model. The states that are put together are fc-similar states, which mean that they share the same behaviour and that the operator need not to be able to distinguish among them. This is precisely one definition of mode.

In this case, the produced conceptual model may bring even more information for the validation phase. If the designer of the system thought about a set of modes when designing the system, they must in some way coincide with the sets of fc-similar states. Figure 6.14 illustrates that principle with the previously presented vehicle transmission system example. The system model has been designed with three modes in the mind of the designers: LOW, MEDIUM and HIGH. For that example, the reduction-based generation algorithm produces the conceptual model of Figure 6.14(b). There are two observations that can be done:

- There are no mode conflict, that is, two states of the system model with different modes have not been gathered in the same block of the partition of the produced conceptual model.

- The mode partition induced by the states of the conceptual model refines the mode partition defined by the system designer.

6.5 User Task Model

In the two previous sections, the elements of analysis are the system model and training materials. In this section, yet another point of view about the interaction is taken by introducing the user's tasks into the analysis. The idea behind the notion of user's tasks is that a given operator is not always interested to know all the behaviour of the system to be used. A given operator may only be interested in performing some *tasks*, which only covers a part of the behaviour of the system.

Given a model of the tasks that the user want to be able to accomplish on a given system, an question that can be answered with the approach developed in this work is whether the system model does support all
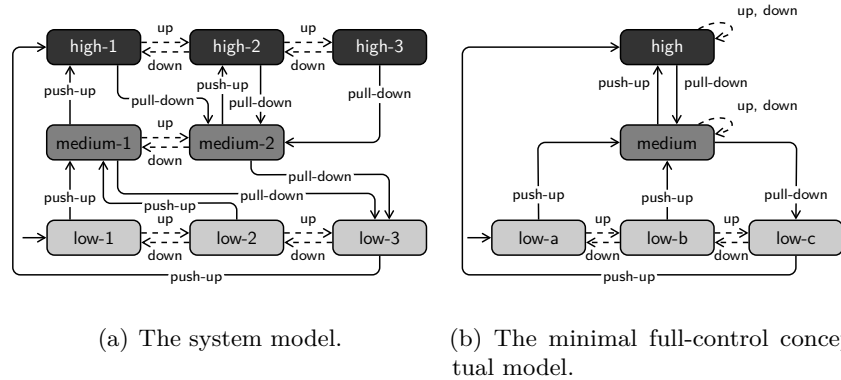
(a) The system model.   (b) The minimal full-control conceptual model.

Figure 6.14. The Vehicle Transmission System (VTS) designed with three modes (the three levels of grey backgrounds for LOW, MEDIUM et HIGH modes) has indeed five modes, according to the full-control property (Example from [HD07]).

the user's tasks. The idea is that a system model must have all the behaviour necessary to support the user's tasks, but of course may have more behaviour. That potential additional behaviour will just not be used by the user, but it must not disturb or confuse the operator.

A user task can also be modelled using an HMI-LTS, representing the behaviour related to the specific task. A *user task model* is simply a set of user task.

Definition 6.4 (User task model). *A* user task model $\mathcal{T}$ *is a deterministic HMI-LTS* $\mathcal{T} = \langle S_T, \mathcal{L}^c, \mathcal{L}^o, s_{0_T}, \rightarrow_T \rangle$.

Figure 6.15 illustrates the concept of user's task. The main element is the user's tasks model of Figure 6.15(a). The user just need to be able to perform the sequence of three actions $\sigma = \langle c_1 o_1 c_2 \rangle$. Looking at the system on Figure 6.15(b) reveals that there are two possible states that can be reached after the execution of the c_1 command: **B** or **E**. If the system does transition to state **E**, the user will be confused since it will be impossible for him to complete the task. In such a situation, the system model is said not to support the user's tasks model.
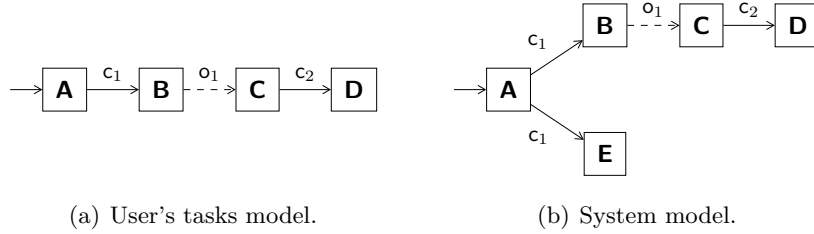
(a) User's tasks model.   (b) System model.

Figure 6.15. A user's tasks model that is not supported by a given system model due to an observation that may not occur in some interactions.

6.5.1 Task-supporting property

To capture the notion of a system model *supporting* a user's tasks model, it is possible to use a variant of the full-control property, where the roles of the commands and observations are reversed.

Definition 6.5 (Task-supporting Property for HMI-LTSs). *Given a deterministic HMI-LTS* $\mathcal{T} = \langle S_T, \mathcal{L}^c, \mathcal{L}^o, s_{0_T}, \rightarrow_T \rangle$ *and an HMI-LTS* $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, $\mathcal{S}$ *is said to* support *the user's task* $\mathcal{T}$, *which is denoted* $\mathcal{S} \, \mathbf{ts} \, \mathcal{T}$, *if and only if for all* $\sigma \in \mathcal{L}^*$ *such that* $s_{0_T} \xrightarrow{\sigma} s_T$ *and* $s_{0_S} \xRightarrow{\sigma} s_S$:

$$A^o(s_T) = A^o(s_S) \text{ and } A^c(s_T) \subseteq A^c(s_S)$$

For a system model to support a user's tasks model, the following must hold at any point during the interaction:

- The observations that are possible must be exactly the same on the system model and the user's tasks model.

- All the commands that the user may perform according to the user's tasks model must be possible on the system model.

The task-supporting property is thus exactly the same as the full-control property except that the requirements on commands and observations is just reversed. A direct consequence is that the full-control property check algorithm (Algorithm 5 on page 249) can be used to test whether a system model supports a user's tasks model.

Figure 6.16 shows yet another example to illustrate the task-supporting property. The system model of Figure 6.16(a) represents a simple

lamp. When the lamp has been turned on the on command, it can be turned off either directly with the off command or gradually with the fadeOut command. In that latter situation, the intensity of the lamp is gradually decreasing (fades state) until being completely turned off, which is represented by the endFading observation. Finally, when the lamp is turned on, it may burn out in which case the lamp becomes unusable (dead state). Since the user has no control about that situation, it is modelled as an internal transition leading to the dies state from which the user can observe that the lamp has died with the burnOut observation.
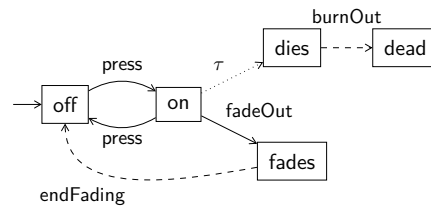
(a) System model.
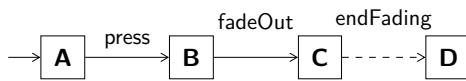
(b) User's tasks model $\mathcal{T}_1$.   (c) User's tasks model $\mathcal{T}_2$.
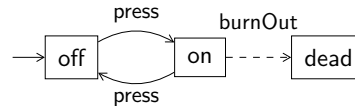
Figure 6.16. An example of a system model for a simple lamp and two different user's tasks model that are to be performed on the system.

Figure 6.16 also shows two user's task model. The first one $(\mathcal{T}_1)$ represents a linear task that consists in turning the lamp on, then turning it off using the fading out option. The second user's task $(\mathcal{T}_2)$ represents that the user must be able to switch between the lamp turned on and off, and is aware that it can burn out, in which case the user does not do anything. Regarding task-supporting property, the following holds:

- $\neg(\mathcal{S} \, \mathsf{ts} \, \mathcal{T}_1)$ since during the interaction, the state $(\mathsf{dies}, \mathbf{B})$ can be reached and its two states do not have the same set of observations. Indeed, if a burnOut observation occurs, it is not foreseen by the user's task which can surprise him.

- $\mathcal{S}\,\textbf{ts}\,\mathcal{T}_2$ since the task-supporting property is satisfied in any state of the interaction. Even if all the behaviour related to the fading out option is not covered by the user's task model. It is not an issue since that behaviour is activated by a command, but the task-supporting property allows the system model to have more commands than those present in the user's task model.

6.5.2 Symmetric Full-control Property

The full-control property requires that, at any time during the interaction, the set of commands that are possible on the system is exactly the same as the set of commands that the operator thinks that are possible on the system according to his mental model. However, as described in Chapter 4, one of the four potentially bad situations captured by the full-control property, namely when a command available on the system is not present in the mental model, is not a potential automation surprise.

The *symmetric full-control property* only requires that, at any point during the interaction, the operator expects at least all the observations that can arise on the system, and that the system supports at least all the commands that the operator is likely to execute.

Definition 6.6 (Symmetric full-control property for HMI-LTS). *Given two HMI-LTSs $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$ and $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$, $\mathcal{H}$ is a* symmetric-full-control *mental model for $\mathcal{S}$, which is denoted $\mathcal{H}\,\textbf{sfc}\,\mathcal{S}$, if and only if for all $\sigma \in \mathcal{L}^*$ such that $s_{0_S} \stackrel{\sigma}{\Longrightarrow} s_S$ and $s_{0_H} \stackrel{\sigma}{\longrightarrow} s_H$:*

$$A^c(s_S) \supseteq A^c(s_H) \ and \ A^o(s_S) \subseteq A^o(s_H)$$

Figure 6.17 illustrates the symmetric full-control property with a simple FM radio example. Once the radio has been turned on, the operator can set it to search the next available channel by pressing scan. Either a channel is found and gets playing; the user can then search for the next available channel. Or the end of the searchable range is reached in which case the operator can restart searching from the beginning of the range by pressing reset. At any time, the operator can also turn the radio off.

The proposed mental model is not a symmetric full-control mental model for the FM radio system, since it contains some behaviour that will
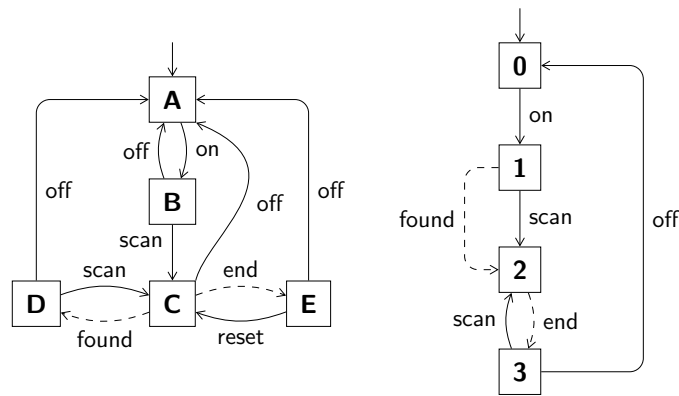
(a) System model $\mathcal{S}$.

(b) Mental model $\mathcal{H}$.
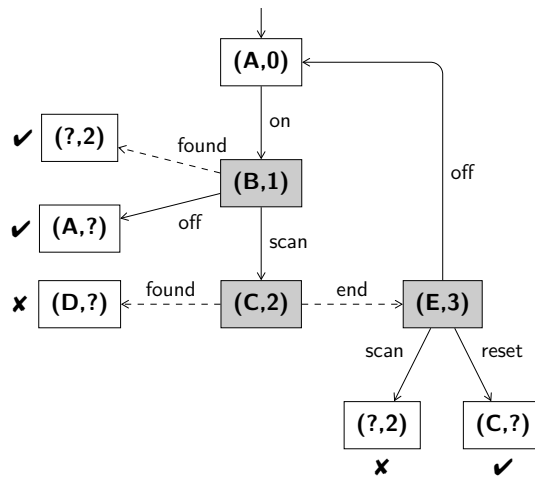
(c) Interaction model $\mathcal{S} \parallel \mathcal{H}$.

Figure 6.17. Example of an FM radio system modelled with an HMI-LTS $\mathcal{S}$. The proposed mental model $\mathcal{H}$ is not a symmetric full-control mental model for the system since there are states of the interaction model $\mathcal{S} \parallel_I \mathcal{H}$ where the symmetric full-control criterion is not satisfied (highlighted with a grey background color).

lead to states where the symmetric full-control property is not satisfied (the greyed states).

- In composite state ($\mathbf{C}, \mathbf{2}$) the system may produce a found observation that is not foreseen by the operator, which will surprise him if it occurs. On that state $A^o(\mathbf{C}) \nsubseteq A^o(\mathbf{2})$.

- In composite state ($\mathbf{E}, \mathbf{3}$) the operator may execute a scan command which is not supported by the system. If the operator executes it, the system will not behave as he would have expected, which will surprise him. On that state, $A^c(\mathbf{E}) \nsupseteq A^c(\mathbf{3})$.

In addition to those two situations which are potentially harmful, there are also other situations that are not problematic according to the symmetric full-control property.

- In composite state ($\mathbf{B}, \mathbf{1}$), the operator can react to a found observation that will in fact never occur on the system in that state. That situation was already the same for full-control property, the user can expect more observations.

- In composite state ($\mathbf{E}, \mathbf{3}$), the system provides a reset command that the user will in fact never use according to his mental model. That situation is different from full-control property in the sense that the symmetric variant allows the system to have more commands than those used by the operator.

Checking Symmetric Full-control

Checking whether a given mental model is a symmetric full-control mental model for a given system can be done exactly in the same way as it is done for the full-control property (see Section 4.2.1), except that the condition that is checked in every composite state is the one of the symmetric full-control property. The interaction model is built and the symmetric full-control property is checked in all its composite states. The interaction model is explored with a BFS so that the counterexample that may be produced if the property is not satisfied are the shortest one.

Minimal Symmetric Full-control Mental Model

The symmetric full-control property does not require the operator to perform all the commands offered by the system. That flexibility induces that a minimal symmetric full-control mental model can ignore all the commands. Only the observations provided by the system do require the symmetric full-control mental model to support them.

A direct consequence is that the minimal symmetric full-control mental model is an HMI-LTS that only has observations on its transitions. Moreover, since the mental model can expect observations that will not actually occur on the system, there is only one unique minimal symmetric full-control mental model for all the possible system models: the one-state HMI-LTS with one loop for each observation of the alphabet, shown on Figure 6.18.

$$\rightarrow \boxed{\mathbf{A}} \ \circlearrowleft \ \mathcal{L}^o$$
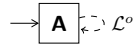
Figure 6.18. The unique minimal symmetric full-control mental model.

Property 6.7. *Given an HMI-LTS $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, the HMI-LTS $\mathcal{H} = \langle \{s_{0_H}\}, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H \rangle$ with $\rightarrow_H = \{(s_{0_H}, o, s_{0_H}) \mid o \in \mathcal{L}^o\}$ is the minimal symmetric full-control mental model for $\mathcal{S}$.*

Proof. By construction, for the mental model $\mathcal{H}$, $A^c(s_{0_H}) = \emptyset$ and $A^o(s_{0_H}) = \mathcal{L}^o$. Let the trace $\sigma \in \mathcal{L}^*$ be a common trace of both models. By construction, the trace will only be composed of observations and, given that $s_{0_S} \overset{\sigma}{\Longrightarrow} s_S$, it is always the case that $A^c(s_S) \supseteq \emptyset$ and $A^o(s_S) \subseteq \mathcal{L}^o$. Consequently, $\mathcal{H}$ is a symmetric full-control mental model for $\mathcal{S}$. Moreover, since it has only one state, it is also minimal. $\square$

The notion of minimal symmetric full-control mental model is not useful for an HMI perspective. Nevertheless, the proposed symmetric variant of the full-control property has some interesting applications as it is described in the next section.

6.5.3 Task Model Completion

The symmetric full-control property exactly captures the situations where the operator may get surprised during an interaction. Combined with the

ideas coming from the task-supporting property presented in Section 6.5.1, it can be used to analyse whether a given user's task, when executed on a given system, will never lead the operator to states where potential automation surprises may occur.

Figure 6.19(a) shows a user task model for the task which consists in turning the radio on and getting it play some music by scanning the frequencies. That user task model is not a symmetric full-control mental model for the system since it does not contain all the observations that may occur during the interactions. However, it can be completed, that is, enriched with additional behaviour, so that to satisfy the symmetric full-control property as shown on Figure 6.19(b).

(a) A user task model $\mathcal{T}$.

(b) Completion of the user task model that allows symmetric full-control of the system.
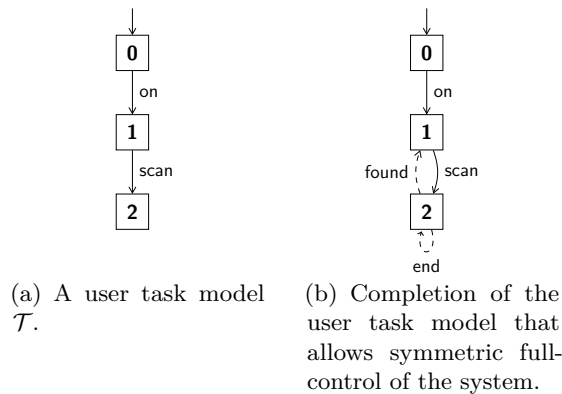
Figure 6.19. A user's task model for the FM radio example (on the left) that has been completed so that it is a symmetric full-control mental model for the system (on the right).

Completing a user task model so that it allows symmetric full-control of the system model, and so that it has the minimal number of states is not a trivial problem. Having such a minimal completion can help designers to test whether the design of a system makes it easy for the operator to user. Indeed, it a lot of states has to be added to the user task model, it would mean that the additional knowledge that the user has to know about the system may be too large. This problem and solutions for it has not been tackled in this thesis but are left for future work directions.

Chapter 7

The Autopilot Case Study

The concepts presented so far in the previous chapters have only been demonstrated on small-sized examples, even if some were realistic. This chapter presents a larger case study which is the model of the autopilot of a Boeing 777 aircraft, that has been modelled in the ADEPT toolset developed by researchers from NASA Ames. Section 7.1 presents the ADEPT toolset and Section 7.2 proposes a formal semantics for its models. Section 7.3 then presents how ADEPT models can be translated into HMI-LTSs provided the proposed formal semantics. Then, Section 7.4 presents completely the autopilot model and its analysis with the techniques presented in this thesis are presented in Section 7.5.

7.1 The ADEPT Toolset and Model

ADEPT, which stands for *Automatic Design and Evaluation Prototyping Toolset* [Fea10], is a Java-based tool that supports designers in the early prototyping phases of the design of automation interfaces. The tool also offers a set of basic analyses that can be performed on the model under development.

7.1.1 General Presentation

An ADEPT model is composed of two elements: a set of logic tables, coupled with an interactive user interface (UI). The *logic tables* are used to describe the dynamics of the system. They describe precisely how the state of the system evolves in reaction to user actions or due to events occurring in the environment. The logic tables also describe how the *user interface* is updated and what information is shown to the operator.

Figure 7.1 shows a screenshot of the autopilot model opened in ADEPT. The left part of the window shows one of the logic tables and the right part shows the user interface.
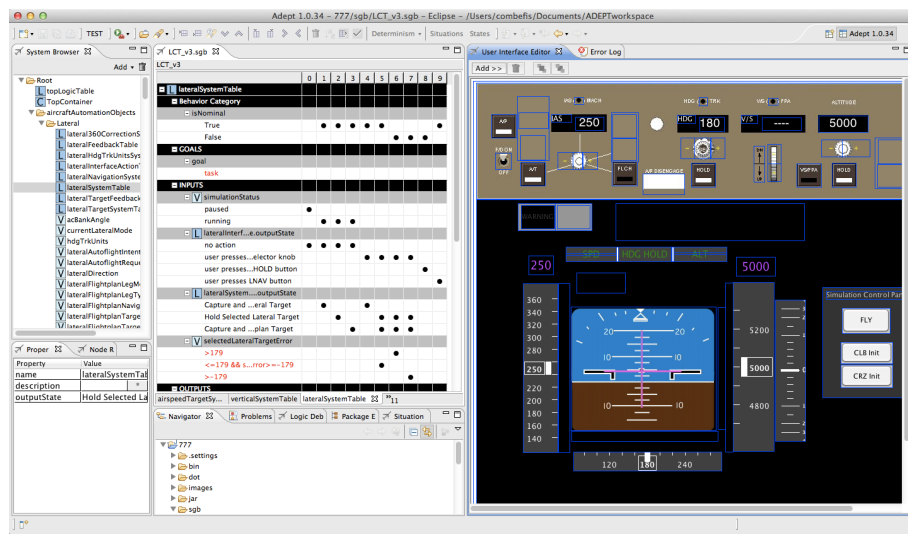

Figure 7.1. The autopilot model opened in ADEPT, with one logic table in the left part of the window and the user interface on the right.

Behind the scene, an ADEPT model is compiled into a Java program that can be executed in order to directly try the encoded behaviour with the user interface. That tool is meant to be used as a rapid prototyping tool. The models can then be tested and simulated by the designers, but can also be analysed by systematic and rigorous techniques. Possible analyses include validity checks on the structure of logic tables, for example.

The UI is composed of a set of components that are encoded as Java objects representing graphical widgets going from simple labels and buttons to more complex elements such as those used in avionics and present on the *primary flight display* (PFD), including the airspeed tape, for example. In addition to those widgets, an ADEPT model also comprises other kinds of elements such as timers, system variables and functions, which are each related to a specific and defined Java construction, namely java.util.Timer, instance variables and methods.

The logic tables can refer to the elements of the UI and to the other components through their Java instance variables, and interact with them through their methods, using Java syntax. The events that can occur in the program come from the components. They can be seen as boolean variables indicating whether they occurred and that can be used in the logic tables.

Figure 7.2 shows one of the logic tables of the autopilot model. The precise structure of those tables is detailed later in this section and a first example is provided just hereafter in Section 7.1.2. That logic table example illustrates the way it can interact with elements of the UI. For example, the last two lines of the logic table mean that the selectedSpeedTarget field of the pfdAirspeedTargetTape component of the UI is updated with the value of the system variable selectedSpeedTarget.

		0	1
L airspeedFeedbackTable			
INPUTS			
L airspeedSystemTable.outputState			
	Maintain Airspeed Target	•	
	Capture Airspeed Target	•	
	Hold Current Airspeed	•	
	Protect Airspeed Target		•
OUTPUTS			
C pfdAirspeedTape.currentValue			
	V indicatedAirspeed	•	•
C cautionLabel.background			
	255, 204, 0		•
C autothrottleModeFailureBar.opaque			
	False	•	
	True		
C pitchModeFailureBar.opaque			
	False	•	
	True		
C pfdAirspeedTape.preSelectedTarget			
	V selectedSpeedTarget		•
C pfdAirspeedTape.selectedTarget			
	V selectedSpeedTarget		•

Figure 7.2. An example of a logic table: the airspeed feedback table of the autopilot model contains the logic related to the update of the UI for the airspeed part.

7.1.2 A Simple Model Example

Figure 7.3 shows the logic table corresponding to a simple counter system. The value of the counter can be increased between 0 and 9 with a press on a button, and it can be reset to 0 at any time. Logic tables are divided into two parts: the input part is used to describe conditions and the output part is used to describe how the state of the system is updated.

		0	1
L **simpleCounter**			
INPUTS			
V value			
< 9		•	
ACTIONS			
press		•	
reset			•
OUTPUTS			
V value			
= value + 1		•	
= 0			•

Figure 7.3. An ADEPT model of a simple counter system, whose value ranges between 0 and 9.

Basically, an ADEPT model describes a system as a set of system variables. The rows of logic tables are divided into blocks of rows, each being related to one variable of the system, with their possible values. The global state of the model is defined by the values of those system variables. A special variable named ACTIONS is used to represent actions performed by the user on the system. Each numbered column corresponds to one possible scenario of the system behaviour.

The input part of the first scenario (the column numbered 0) represents the following condition:

$$\text{value} < 9 \wedge \text{ACTIONS} = \text{press}$$

If this condition is met in the current state of the system, then the next state is computed from the current state by executing the following assignment instruction:

$$\text{value} \leftarrow \text{value} + 1$$

ADEPT models have a similar expressivity than HVMs and they can thus be translated so that the analyses proposed in this thesis can be applied to them. Figure 7.4 shows how the behaviour encoded in the ADEPT model can be translated into a graph. Each state of the graph represents a state of the ADEPT model, that is, a valuation of the set of system variables. Transitions between states correspond either to the user having executed an action, or to the execution of a scenario of the logic table.
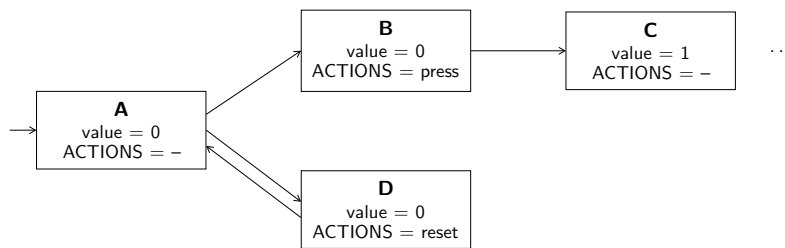

Figure 7.4. Partial translation of the simple counter ADEPT model into a graph and representing the same behaviour.

This proposed translation from an ADEPT model to a simple graph is not the only way to proceed. In particular, for this work, the important aspect are the labels on transitions that indicates the action performed by the user on the system. An HVS such as the one proposed on Figure 7.5 corresponds to a model that can be analysed with the techniques proposed in this work. The value of the special ACTIONS variable has been used to define command labels on the transitions. In order to be able to translate systematically and automatically ADEPT models into HVSs, a formal semantics is needed. The latter is the subject of the next section.
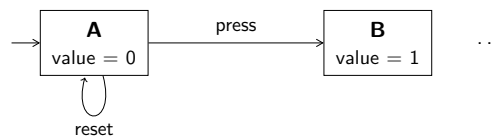

Figure 7.5. Partial translation of the simple counter ADEPT model into an HVS and representing the same behaviour.

7.2 A Formal Semantics of ADEPT

The meaning of an ADEPT model can be decomposed into two parts: the *logic part* is defined by the logic tables and the *program part* is defined by the UI components and the other elements. The behaviour of an ADEPT model is as that of a Java program where the logic tables are executed in reaction to events occurring in the program part. The logic tables can be seen as listeners reacting to Java events produced by the elements of the program part. Figure 7.6 illustrates the behaviour of an ADEPT model. All the elements are first initialised or assigned their initial values, which defines the initial state of the ADEPT model. The behaviour then loops forever, alternating between the program and logic parts. The execution leaves the program part whenever an event occurs, and the execution is transferred to the logic part, just after having set the boolean variable corresponding to the event that occurred. Once all the logic tables have been executed, the flow goes back to the program part.
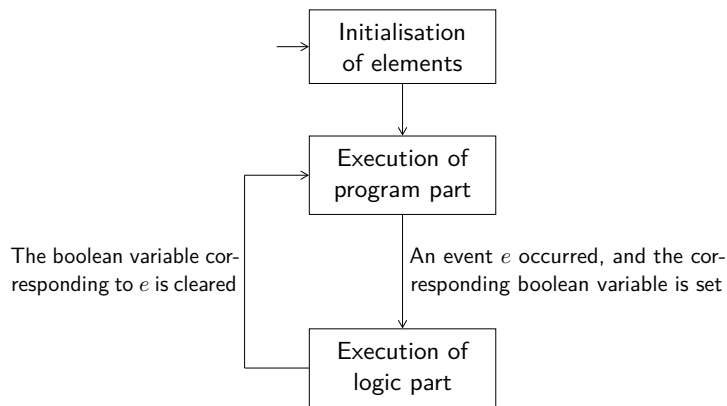

Figure 7.6. Behaviour of an ADEPT model. Once all the elements of the model have been initialised, the behaviour loops forever alternating between the program and the logic parts.

7.2.1 ADEPT Logic Tables

Logic tables are two-dimensional tables split in two parts: INPUTS and OUTPUTS. Each of those parts is then structured into a two-level hierarchy of elements and values, so that the values are always associated to an element. As previously mentioned, the element-value pairs are representing conditions for the inputs whereas they represent statements for the outputs. Figure 7.7 shows an example illustrating the structure of logic tables.

INPUTS			
element 1			
condition 1.1	•	•	
condition 1.2		•	•
condition 1.3	•		•
element 2			
condition 2.1	•		•
condition 2.2		•	•
OUTPUTS			
element 3			
statement 3.1	•	•	
statement 3.2		•	

Figure 7.7. Structure of the logic tables of ADEPT models.

In addition to the separation between the input and the output parts, a logic table is also structured horizontally. The left part of the table describes the element-value pairs and the right part of the table consists in a sequence of columns.

Row Headers

The *row headers* are representing the left part of the logic tables. A row header is a sequence of switches. Each switch is the association between an element and a sequence of values. A distinction has to be made between input and output headers, but their structure is identical. The table header is a pair made of an input header and an output header.

The following sets define formally those concepts:

$$
\begin{aligned}
Header &= IHeader \times OHeader \\
IHeader &= ISwitch^* \\
ISwitch &= IElem \times IVal^* \\
OHeader &= OSwitch^* \\
OSwitch &= OElem \times OVal^*
\end{aligned}
$$

The sets $IElem, IVal, OElem$ and $OVal$ are detailed further in Section 7.2.2, with the description of the program part of ADEPT models.

The input header of the example of Figure 7.7 contains two input switches and the output header one output switch. Formally, the table header of the example is defined as:

$$
\begin{aligned}
H &= (H^I, H^O) \\
H^I &= (S_1^I, S_2^I) \\
S_1^I &= (\text{element 1}, (\text{condition 1.1}, \text{condition 1.2}, \text{condition 1.3})) \\
S_2^I &= (\text{element 2}, (\text{condition 2.1}, \text{condition 2.2})) \\
H^O &= (S_1^O) \\
S_1^O &= (\text{element 3}, (\text{statement 3.1}, \text{statement 3.2}))
\end{aligned}
$$

Columns

The *columns* of a logic table define elementary fragments of dynamic behaviour, referred to as *situation-automation behaviour pairs* in [Fea07], expressed as conditions on the current state (the input part) and changes to be applied to the state to get the next state (output part).

A *row index* is a pair $y = (i, j) \in \mathbb{N} \times \mathbb{N}$, denoted $i.j$ and that represents the jth value of the ith element.

$$
Index = \mathbb{N} \times \mathbb{N}
$$

An *index set* is a set $Y \subset \mathbb{N} \times \mathbb{N}$ of row indices. An index set is an *index range* if its first indices cover a continuous range $1, \cdots, n$ and its second indices cover a continuous range $1, \cdots, n_i$ for each first index i. The subset $Y|_i = \{i.j \in Y\}$ contains row indices whose first index is i. For example, the set $Y = \{1.1, 1.2, 2.1, 2.2, 2.3, 3.1\}$ is an index range, but $\{1.1, 3.1\}$ and $\{1.1, 1.3\}$ are not, and $Y|_2 = \{2.1, 2.2, 2.3\}$.

An (input or output) header H is seen as a mapping from row indices to (E, V) pairs according to the following rule. Given that $H = (H_1, \cdots, H_n)$ with $H_i = (E_i, (V_{i,1}, \cdots, V_{i,n_i}))$, then $H(i.j) = (E_i, V_{i,j})$, provided that $1 \leq i \leq n$ and $1 \leq j \leq n_i$. The domain of H is thus an index range, by definition.

A column gives a binary value for every row of the logic table that corresponds to a value (condition or statement). That assignment is represented by the bullets in the graphical representations of the logic tables. Formally, it can be represented as a mapping from row indices to boolean values. Another equivalent way to describe a column is to define two sets of row indices C^I and C^O, respectively included in the domains of the input and output headers of the logic table, so that $i.j \in C^I$ if and only if the row $i.j$ is marked in that column of the logic table, and similarly for C^O.

$$Col \quad = \quad 2^{Index} \times 2^{Index}$$

The three columns of the example of Figure 7.7 can be formally defined as:

$$
\begin{aligned}
C_1 &= (\{1.1, 1.3, 2.1, 3.1\}, \{3.1\}) \\
C_2 &= (\{1.2, 2.2\}, \{3.1, 3.2\}) \\
C_3 &= (\{1.2, 1.3, 2.1, 2.2\}, \emptyset)
\end{aligned}
$$

Tables and Models

Given the definitions of the row headers and of the columns, it is now possible to define the logic tables. A *logic table* is a structure $T = (H^I, H^O, CC)$, where:

- $H^I \in IHeader$ is an input header;

- $H^O \in OHeader$ is an output header;

- and $CC \in Col^*$ is the table body consisting of a list of columns $C = (C^I, C^O)$ where $C^I \subseteq \mathbf{dom}(H^I)$ and $C^O \subseteq \mathbf{dom}(H^O)$.

All the structures that make a table are totally ordered, either explicitly because they are defined as sequences, or implicitly by the ordering on indices, in the case of row index sets.

$$Table \quad = \quad IHeader \times OHeader \times Col^*$$

Finally, an ADEPT (logic part of) model M is a collection of named logic tables (on a set of names $Name$), such that all the names are different and with a distinguished logic table named *top*.

$$Model = 2^{Name \times Table}$$

A model M is seen as a mapping from names to tables, defined so that $M(N) = T$ if $(N, T) \in M$.

7.2.2 ADEPT Programs

The *program part* of an ADEPT model captures the behaviour related to the interaction of the user with the user interface. The elements of the program part belong to a Java program, and all the elements, conditions and statements that lie in the tables result, directly or indirectly, into the generation or execution of Java code. In particular, all system variables are implemented as Java variables.

Entities types

There is a total of five different types of entities, designated with capital boxed letters, that can be used to compose an ADEPT model. Those entities can appear in row headers as elements and as values. Of course, the meaning of those entities varies if they are used in the input or output part. The following types are supported in ADEPT:

$\boxed{\text{V}}$ *System variables* correspond to Java variables and can appear both as elements and as values, in input and output switches.

$\boxed{\text{C}}$ *UI components* correspond to Java GUI widgets from the user interface. Each widget can be used in several ways in the row headers:

– *GUI events* can be used in input switches for the distinguished element `ACTIONS` as input conditions. The event parameters are not accessible. Examples include mouse pressed/released/clicked.

– *GUI component attributes* (foreground/background colour, currentValue, opaque...) can be used in output switches as elements whose value can be changed with output statements.

- Arbitrary *GUI component methods* can be invoked in output switches for the distinguished element PRIMITIVES as output statements corresponding to their execution.

$\boxed{\text{O}}$ *Timers* are used to schedule repetitively some events. They can be used in two ways:

- *Timer events* can be used in input switches for the distinguished element ACTIONS as input conditions. It corresponds to the actionPerformed Java method and the event parameter is not accessible.
- *Timer methods* can be used in output switches for the distinguished element PRIMITIVES as output statements corresponding to their execution. The Java methods that can be invoked are start and stop.

$\boxed{\text{F}}$ *Functions* correspond to Java methods which return a value. They can be used in output switches as values.

$\boxed{\text{L}}$ *Logic tables* can appear by themselves in output switches for the distinguished element LOGIC as output values. Each logic table T also has an associated variable $T.\text{outputState}$ that can appear both in input and output switches as element and whose values are of enumerated type.

Conditions

Input switches of logic tables define *conditions* that correspond to Java boolean expressions. Conditions are derived from input element-value pairs, that is, $B = cond(E^I, V^I)$ with $E^I \in IElem$ and $V^I \in IVal$.

$$cond : IElem \times IVal \mapsto JavaExpr$$

The definition of *cond* depends on the type of the E^I element. The result of *cond* is a text string that has to be interpreted as Java code representing a Java condition that can be evaluated.

- $cond(\boxed{\text{V}}\ \text{var}, \text{expr}) = \text{var} == \text{expr}$

  expr is a single expression. For example, $cond(\boxed{\text{V}}\ \text{x}, 8) = \text{x} == 8$.

- $cond(\boxed{\text{V}}\ \text{var}, \text{str}) = \text{var str}$

  str is typically of the form op expr where op is a comparison operator, but it can be any string that is used to form non-atomic boolean expressions. For example, $cond(\boxed{\text{V}}\ \text{x}, >= 0\ \&\&\ \text{x} < 5) = \text{x} >= 0\ \&\&\ \text{x} < 5$.

- $cond(\text{ACTIONS}, \boxed{\text{C}}\ \text{component.event}) = \text{component.event}$
  $cond(\text{ACTIONS}, \boxed{\text{O}}\ \text{timer.event}) = \text{timer.event}$

  event is an event name of an UI component (component or timer), or more precisely a boolean variable associated with the event.

- $cond(\boxed{\text{L}}\ \text{table.outputState}, \text{value}) = \text{table.outputState} == \text{value}$

  value is a constant value in the range of table.outputState since the outputState variable that can be associated to a table is an enumerated type.

Statements

Output switches of logic tables define *statements* that correspond to Java statements. Statements are derived from output element-value pairs, that is, $S = stmt(E^o, V^o)$ with $E^o \in OElem$ and $V^o \in OVal$.

$$stmt : OElem \times OVal \mapsto String$$

The definition of *stmt* also depends on the type of the E^o element. The result of *stmt* is also a text string that has to be interpreted as Java code representing a Java statement that can be executed.

- $stmt(\text{LOGIC}, table) = \text{call (table)}$

  table is the name of a logic table that has to be executed. The call Java method is defined so that $exec(\text{call(table)}, q) = [\![M(\text{table})]\!](q)$.

- $stmt(\text{PRIMITIVES}, \boxed{\text{O}}\ \text{timer.method}) = \text{timer.method()}$
  $stmt(\text{PRIMITIVES}, \text{statement}) = \text{statement}$

  method is typically start or stop and statement is any Java statement, typically a single method call or an assignment.

- $stmt(\boxed{\mathbf{V}}\ \mathsf{var}, \mathsf{expr}) = \mathsf{var} = \mathsf{expr}$
  $stmt(\boxed{\mathbf{V}}\ \mathsf{var}, \boxed{\mathbf{V}}\ \mathsf{var'}) = \mathsf{var} = \mathsf{var'}$

  expr is a single expression.

- $stmt(\boxed{\mathbf{V}}\ \mathsf{var}, \boxed{\mathbf{F}}\ \mathsf{fun}) = \mathsf{var} = \mathsf{fun}()$

  fun is a function implemented as a Java method which returns a value and that can possibly change the state of the model (that is, modify the value of the system variables).

- $stmt(\boxed{\mathbf{V}}\ \mathsf{var}, \mathsf{str}) = \mathsf{var}\ \mathsf{str}$

  str is typically of the form = expr or op = expr. For example, $stmt(\boxed{\mathbf{V}}\ \mathsf{x}, += 3) = \mathsf{x}\ += 3$.

- $stmt(\boxed{\mathbf{L}}\ \mathsf{table.outputState}, \mathsf{value}) = \mathsf{table.outputState} = \mathsf{value}$

  value is a constant value in the range of table.outputState. Moreover, table must be the current logic table the outputState variable can only be changed by the table it belongs to.

- $stmt(\boxed{\mathbf{C}}\ \mathsf{component.field}, \mathsf{value}) = \mathsf{component.field} = \mathsf{value}$

  component.field designates the field attribute of the component UI component. It can for example be used to change the appearance of a GUI widget.

7.2.3 Execution Semantics

The state of an ADEPT model is defined by the state of its program part. The state is composed of the value of the system variables and of the individual states of all the other elements of the program part, such as the UI components and timers. The program part therefore defines the set of possible states $q \in Q$ of the ADEPT model. In particular, all the elements do have an initial state, that is pre-defined (timers are for example initially inactive) or user-defined (for example for system variables), which defines the global initial state of the ADEPT model.

Well-formed Logic Tables

Logic tables must satisfy two requirements so that the ADEPT model composed by those tables is valid:

1. within a logic table, the input part of the columns should be *complete* and *non-overlapping*;

2. and within an input switch, the conditions should be *complete* and *non-overlapping*, except for the input switches having the distinguished ACTIONS element.

Those two requirements together guarantee that exactly one column (at most one for ACTIONS) will be applied for any invocation of a table. The second requirement ensures that only one condition is true for any input switch in any state, and the first requirement ensures that any combination of such conditions is covered in any table.

The ACTIONS elements need a special treatment. The values of those elements correspond to events produced either by interactions with UI components or timer triggers. By construction, at most one event variable is set on any invocation of the logic tables. However, ACTIONS input switches need not to cover all the possible actions and thus, may be incomplete.

The two requirements required for a logic table to be well-formed can be formally defined. Given an index set Y, the *choices* over Y are defined as the set:

$$choices(Y) = Y|_1 \times \cdots \times Y|_n$$

where $n = \max\{i \mid i.j \in Y\}$. One choice over Y picks one $i.j$ index for each first-level index i appearing in Y. The *choices* function applied to an input header H^I, that is, applied to $Y = \mathbf{dom}(H^I)$, returns all possible combinations of single values for each input switch of H^I.

It may be the case that for some input switches, there are some columns for which no conditions are marked. The expansion adds all the conditions for input switches where no condition is marked. The corresponding interpretation is that blank input switches correspond to "don't-cares" and must therefore be interpreted as always true. Given an index set C (from an input column) and an index range Y (from the

input header), the *don't care expansion* of C with respect to Y is defined as:

$$expand(C, Y) = C \cup \bigcup \{Y|_i \mid C \cap Y|_i = \emptyset\}$$

Property 7.1 (Well-formed logic tables). *Two requirements have to be satisfied for a logic table (H^I, H^O, CC) to be well-formed.*

- **(*Requirement 1*)** *Let $CC = (C_1, \cdots, C_m)$ and $C_i = (C_i^I, C_i^O)$. For any choice $C^* \in choices(\mathbf{dom}(H^I))$, there is a unique $1 \leq i \leq m$ such that $C^* \subseteq expand(C_i^I, \mathbf{dom}(H^I))$.*

- **(*Requirement 2*)** *For any input switch $(E^I, (V_1^I, \cdots, V_k^I))$, where $E^I \neq \mathsf{ACTIONS}$ (resp. $E^I = \mathsf{ACTIONS}$), for any state q, there is a unique (resp. at most one) i such that $1 \leq i \leq k$ and $eval(cond(E^I, V_i^I), q) = T$.*

Figure 7.8 shows several input part examples that illustrate the two requirements necessary for a logic table to be well-formed.

- Example of Figure 7.8(a) fails to satisfy requirement 1. Indeed, there is, for example, no $1 \leq i \leq 3$ that supports the choice $C^* = (1.3, 1.1)$. However, requirement 2 is satisfied.

- Example of Figure 7.8(b) fails to satisfy requirement 2. Indeed, for a state q where the value of x is 4, the second and third conditions on the element are both satisfied. However, requirement 1 is satisfied.

- Example of Figure 7.8(c) does satisfy both requirements and is thus a well-formed logic table.

Whereas the first requirement can be automatically checked by the ADEPT toolset, it is not possible for the second requirement. If the first requirement fails to be satisfied, the ADEPT toolset generates a warning, but the model is still executable. Jointly enabled columns are seen as non-determinism. The column that will be executed is undetermined. The second requirement involves arbitrary Java code, which makes it undecidable in general.

Another constraint, enforced by the first requirement, is that if a behaviour is provided for an event, it must be provided for any combination of conditions. If the event should be ignored under some conditions,

INPUTS

x			
<= 0	•		
> 0 && x < 5		•	•
>= 5			
y			
== 100	•		
!= 100		•	•

INPUTS

x			
<= 0	•		
> 0 && x < 5		•	
=> 4		•	
y			
== 100	•		
!= 100			•

INPUTS

x			
<= 0	•		
> 0 && x < 5		•	•
>= 5		•	•
y			
== 100		•	
!= 100			•

(a) Requirement 1 fails. (b) Requirement 2 fails. (c) Well-formed logic table.

Figure 7.8. Illustration of the two requirements necessary for a logic table to be well-formed. The two system variables x and y are supposed to be int that can take any value in the domain of Java int variables.

the corresponding column must have an empty output part. The event is said to be *inhibited*. This makes it possible to distinguish intentionally inhibited events from mistakenly unspecified behaviour in some set of conditions.

Execution Semantics of ADEPT Tables

The elementary computations that are performed during the execution of an ADEPT model are captured in the *eval* and *exec* functions:

- A condition B evaluated in a program state q results in a boolean value defined by $eval(B, q) \in \{T, F\}$.

- A statement S executed in a program state q results in a new state defined by $exec(S, q) \in Q$.

The *execution semantics* defines semantic mappings $[\![\alpha]\!](q)$, denoting the semantics of the syntactic constructs α in a state q of the model. For input constructs, the result of the semantic mappings is a boolean value and for output constructs, the results is a new state.

The tables are first converted by *projection of headers on columns*. The idea is that a column C can be seen as a filter on the table column headers H^I and H^O. That idea is formalised by the projection operation $(H^I, H^O)/C$ that produces a reduced table header containing only rows selected by the column C. The projection operation also integrates the expansion of don't-cares.

An (input or output) header $H = (H_1, \cdots, H_n)$ is projected on an index set $Y \subseteq \mathbf{dom}(H)$ as follows:

$$(H_1, \cdots, H_n)/Y = (H_1/Y, \cdots, H_n/Y)$$

where, for each $H_i = (E_i, (V_{i,1}, \cdots, V_{i,n_i}))$,

$$(E_i, (V_{i,1}, \cdots, V_{i,n_i}))/Y = (E_i, (V_{i,j} \mid i.j \in Y, 1 \le j \le n_i))$$

All the elements E_i are preserved, with an empty list of values if Y contains no index $i.j$. The projection is extended to a column $C = (C^I, C^O)$ as:

$$(H^I, H^O)/(C^I, C^O) = (H^I/expand(C^I, H^I), H^O/C^O)$$

and then to a list of columns $CC = (C_1, \cdots, C_m)$ as:

$$(H^I, H^O)/(C_1, \cdots, C_m) = ((H^I, H^O)/C_1, \cdots, (H^I, H^O)/C_m)$$

Figure 7.9 shows the execution semantics of ADEPT models. The semantics mapping is presented in a top-down denotational style and is based on common functional programming constructs. The two requirements for well-formed logic tables ensure that there is at most one i such that $[\![H_i^I]\!](q)$ for any projected input column $(H_1^I, \cdots, H_{n^i}^I)$. Moreover, the second requirement also guarantees that there is at most one i such that $[\![E^I]\!](V_i^I)$, and exactly one except if $E^I = \mathsf{ACTIONS}$.

7.3 ADEPT to HMI-LTS Translation

Based on the formal semantics proposed in the previous section, it is possible to translate an ADEPT model into any other formalism that is able to represent the same kind of behaviour. More precisely, only the logic part of ADEPT models is covered by the proposed semantics since Java semantics is not captured at all. Indeed, in the proposed semantics, elements coming from the Java language including conditions and statements are directly evaluated according to the Java semantics. Moreover, the translation proposed in this work does not cover all the models, but only some of them following a precise structure, defined in this section.

$$
\begin{array}{rcl}
[\![M]\!] & : & Q \mapsto Q \\
[\![M]\!](q) & = & [\![M(top)]\!](q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![T]\!] & : & Q \mapsto Q \\
[\![(H^I, H^O, CC)]\!](q) & = & [\![(H^I, H^O)/CC]\!](q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![((H_1^I, H_1^O), \cdots, (H_n^I, H_n^O))]\!] & : & Q \mapsto Q \\
[\![((H_1^I, H_1^O), \cdots, (H_n^I, H_n^O))]\!](q) & = & \left\{ \begin{array}{ll} [\![H_i^O]\!](q) & \text{if } \exists i \cdot [\![H_i^I]\!](q) \\ q & \text{otherwise} \end{array} \right.
\end{array}
$$

$$
\begin{array}{rcl}
[\![H^I]\!] & : & Q \mapsto \{T, F\} \\
[\![(S_1^I, \cdots, S_n^I)]\!](q) & = & [\![S_1^I]\!](q) \wedge \cdots \wedge [\![S_n^I]\!](q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![S^I]\!] & : & Q \mapsto \{T, F\} \\
[\![(E^I, (V_1^I, \cdots, V_k^I))]\!](q) & = & [\![(E^I, V_1^I)]\!](q) \vee \cdots \vee [\![(E^I, V_k^I)]\!](q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![(E^I, V^I)]\!] & : & Q \mapsto \{T, F\} \\
[\![E^I, V^I]\!](q) & = & eval(cond(E^I, V^I), q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![H^O]\!] & : & Q \mapsto Q \\
[\![(S_1^O, \cdots, S_n^O)]\!](q) & = & ([\![S_n^O]\!] \circ \cdots \circ [\![S_1^O]\!])(q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![S^O]\!] & : & Q \mapsto Q \\
[\![(E^O, (V_1^O, \cdots, V_k^O))]\!](q) & = & ([\![(E^O, V_k^O)]\!] \circ \cdots \circ [\![(E^O, V_1^O)]\!])(q)
\end{array}
$$

$$
\begin{array}{rcl}
[\![(E^O, V^O)]\!] & : & Q \mapsto Q \\
[\![E^O, V^O]\!](q) & = & exec(stmt(E^O, V^O), q)
\end{array}
$$

Figure 7.9. Execution semantics of ADEPT models.

7.3.1 ASF structure

ADEPT models considered in this thesis, and in particular the autopilot model, are structured in a particular way which allows a clear distinction between:

- the actions performed by the operator on the user interface;
- the internal decision logic of the system;
- and the feedback provided by the system to the user through the user interface.

That particular structure, referred to as the *action-system-feedback structure* (ASF), is reflected in the way the logic tables of the model

are particularised and organised. The logic tables are partitioned into three groups: the action tables, the system tables and the feedback tables. The execution of the logic part of an ADEPT model is always happening following the same order: action tables, then system tables and finally feedback tables, as illustrated by Figure 7.10. Commands executed by the user are dealt with by the actions tables and observations produced by the system and sent back to the user interface are managed by the feedback tables. The operator can then examine the outputs to choose the next action to be performed on the system, which makes the execution loop of the system.
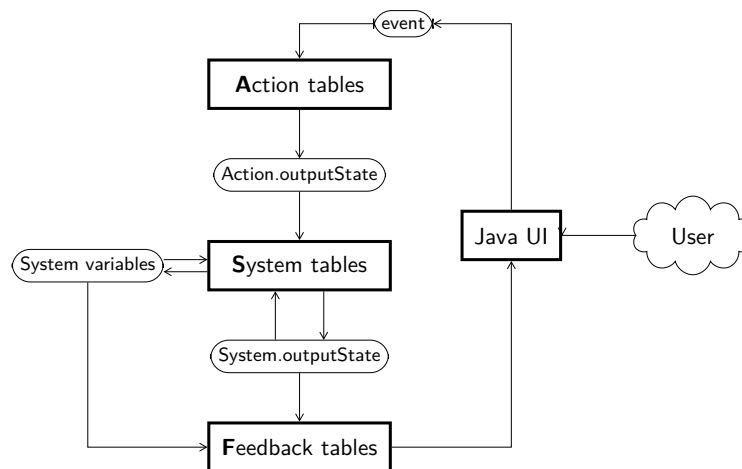
Figure 7.10. Structure of the logic tables of ADEPT models used in this thesis, and execution loop of the model.

1. The *action tables* manage the interaction between the operator and the system through the action components on the interface. Those tables essentially take as inputs actions by the operator on the interface action components or automatic events triggered by timers. The different possible results from those tables are summarised with the outputState variable of the table.

2. The *system tables* contain the decision logic of the system. Those tables use information from the operator's actions that are in the

outputState variables of the action tables and, combined with the
values of the variables, update the state of the system. The be-
haviour related to the different system tables is also summarised
with the outputState variable of each table. These different output-
State variables can be regarded as defining the operating mode of
the system.

3. Finally, the *feedback table* are used to characterise what information
is shown to the user. Information about what the system has done
and is doing is provided to the user through display components
on the interface. The outputState variables of the system tables
are used to display the operating mode and some system variables'
values to the user.

The partitioning imposed by the ASF structure also conditions where
the different elements will be used, both in the input and in the output
parts of the logic tables. For example, events will only occur in the inputs
of action tables and widgets properties will only occur in the outputs
of feedback tables. Moreover, given that the events are generated by
interaction with Java GUI widgets or by timer triggers, and that those
events are managed by a unique *event-dispatching thread* (EDT) in
Java, it can be assumed that each loop through the logic tables will be
associated with one single event. This important assumption plays a
crucial role in the translation from an ASF ADEPT model to an HVS.

The AP-FD-AT Example

Figures 7.11 to 7.13 illustrate how the ASF structure is implemented
for a particular part of the ADEPT model of the autopilot used in this
chapter. The three presented tables are used to manage three elements:
A/P (Autopilot), F/D (Flight Director) and A/T (Auto Throttle).

Figure 7.11 shows the action table. The input part only considers
the distinguished element ACTIONS and manages click actions on several
UI components. It also considers the actionPerformed action of a warning
timer. The output part summarises the command performed by the
operator using the outputState variable associated to the logic table. In
this particular situation, the number of different values for the outputState
variable is the same as the number of actions, but this is not always

the case. The three rows related to the functionWarningTimer are not considered in this work and can be ignored.

	0	1	2	3	4	5	6	7
L apFdAtInterfaceActionTable								
INPUTS								
ACTIONS								
C mcpCaptainsFdSwitch.mouseClicked	•							
C mcpFoFdSwitch.mouseClicked		•						
C mcpLeftApButton.mouseClicked			•					
C mcpRightApButton.mouseClicked				•				
C mcpAtArmSwitches.mouseClicked					•			
C mcpLocButton.mouseClicked						•		
C mcpAppButton.mouseClicked							•	
O functionWarningTimer.actionPerformed								•
OUTPUTS								
L apFdAtInterfaceActionTable.outputState								
user toggles captains fd switch	•							
user toggles first officers fd switch		•						
user presses left AP button			•					
user presses right AP button				•				
user toggles autothrottle arm switches					•			
user presses LOC button						•		
user presses APP button							•	
no action								•
PRIMITIVES								
O functionWarningTimer.stop	•	•	•	•	•	•	•	•
O functionWarningTimer.start	•	•	•	•	•	•	•	

Figure 7.11. The autopilot flight director attitude interface action table manages the events related to the UI components concerning the management of the autopilot.

Figure 7.12 shows the system table which contains the decision logic. The input part is looking at the outputState of the corresponding action table and, based on that, sets the outputState of its own logic table. Generally, the input part can also have conditions on the state variables of the system, as well as conditions on its own outputState. The output part can also update the state variables. The outputState variable of system tables can be considered as describing modes, since they summarise part of the behaviour of the global system. For the current example, the captains FD can either be activated or not.

		0	1	2
L captainsFdSystemTable				
INPUTS				
L apFdAtInterfaceActionTable.outputState				
no action				•
user toggles captains fd switch		•	•	
L captainsFdSystemTable.outputState				
captains FD on		•		
captains FD off			•	
OUTPUTS				
L captainsFdSystemTable.outputState				
captains FD off		•		
captains FD on			•	

Figure 7.12. The captains flight director system table contains the logic related to the management of the flight director by the captain.

Finally, Figure 7.13 shows the feedback table. The input part of the logic table refers to the outputState of the corresponding system table. In this particular example, it does refer to the outputState of two other tables. The output parts is concerned with the update of GUI components. Feedback tables encode in some way the observation that are made available to the operator.

7.3.2 ASF ADEPT Model Translation

As a reminder, a system model $\mathcal{S}$ is defined with an HVS which is a tuple $\mathcal{S} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow, \mathcal{L}^v, \mathcal{O} \rangle$ (Definition 3.13 on page 76). When translating an ADEPT model with the ASF structure, only the logic tables belonging to the system tables category are considered. The reason motivating that choice is that the decision logic of the system is precisely described in that part of the model. The other tables indirectly play a role in the translation. Actions tables are used to identify the alphabet of the HVS and feedback tables are used to define the visible system variable used to define the set of state-values.

- Each state $s \in S$ of the HVS corresponds to a unique assignment of values to all the system variables of the ADEPT model (including outputState variables).

	0	1	2	3
L apFdAtInterfaceFeedbackTable				
INPUTS				
L captainsFdSystemTable.outputState				
captains FD on	•	•		
captains FD off			•	•
L firstOfficersFdSystemTable.outputState				
first officers FD on	•		•	
first officers FD off		•		•
OUTPUTS				
C mcpCaptainsFdSwitch.source				
B777_fd_switch_on.jpg	•	•		
B777_fd_switch_off.jpg			•	•
C mcpFoFdSwitch.source				
B777_fd_switch_on.jpg	•		•	
B777_fd_switch_off.jpg		•		•

Figure 7.13. The autopilot flight director attitude interface feedback table updates the UI components concerning the management of the autopilot.

- The set of commands $\mathcal{L}^c$ corresponds to all the actions that the operator can perform on the system. That set is composed of the union of the domains of all the outputState variables of the action tables, except the distinguished no action value that has a special role explained below.

- The set of observations $\mathcal{L}^o$ is empty.

- The initial state s_0 is based on the initial values of the system variables and is the same as the initial state of the ADEPT model.

- The transition relation $\rightarrow$ is defined according to the execution semantics defined in Section 7.2.

- The set of state-values $\mathcal{L}^v$ is defined by the visible system variables and is defined with the state-value mapping function $\mathcal{O}$ exactly as described in Section 3.3.1.

Hypotheses

An assumption that has to be satisfied is that the operator cannot perform more than one action at a time. For that to be true, it means that for each execution cycle of the ADEPT model, there is only one

table at most which is executed and outputs a table outputState variable. If the particular value "no action" is true in all the tables of the model, it corresponds to an internal τ-translation in HVS.

7.3.3 The Video Cassette Recorder example

The *video cassette recorder* example (VCR) is a model coming from ADEPT which consists of a single logic table shown on Figure 7.14. The machine has a total of six main operating modes: play, stopped, fast forward, rewind, pause and record. In addition to a command to activate each of those modes, the machine has one button to turn the machine on or off. Moreover, the fast forward and the rewind modes can operate at different speeds. The speed is automatically adjusted according to the remaining tape length, so that it slows down when the remaining tape length is becoming smaller. Finally the machine automatically switches to the rewind mode whenever the tape reaches its end, that is, the remaining tape length is zero.

The VCR ADEPT model is an example that is not following the ASF structure. However, it can be translated into an HVS, following the proposed semantics. The translation of the VCR model into an HVS results in a system with 1088 states and 3740 transitions. There are seven commands (one to activate each mode and a power button) and two observations (tape moving forward and backward). The two observations correspond to timer events that are produced whenever the remaining tape length is changing, because of some of the operating modes. The states of the HVS are characterised by three variables: the status of the VCR (ON or OFF), its mode among the six possible and finally the value of the tapeRemaning variable.

The analysis of the system, trying to generate a full-control conceptual model for it, raised some issues. More precisely, the system is not fc-deterministic and it can for example be observed at the [ON, STOP, 0.01] state. That state corresponds to the VCR being turned on, and with the tape being stopped with a remaining tape length of 0.01. From that state, if the user presses the play button, the system can transition into one of the two following states: [ON, PLAY, 0.01] or [ON, REWIND_FULL, 0.00]. The sets of possible commands in those two states are not the same. Indeed, the pause command is not possible in both states. In the first

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
L airspeedFeedbackTable																							
INPUTS																							
ACTIONS																							
C fastForwardButton.mouseClicked	•	•	•																				•
C playButton.mouseClicked	•			•	•																		•
C rewindButton.mouseClicked	•					•	•																•
C stopButton.mouseClicked								•															•
C pauseButton.mouseClicked	•								•	•													•
C recordButton.mouseClicked											•	•											•
C tapePosition.actionPerformed													•	•	•	•	•	•	•	•			•
C powerButton.mouseClicked																					•	•	
L topLogicTable.outputState																							
Stop Tape			•	•	•	•	•			•			•										
Play Tape			•	•	•	•	•		•						•	•							
Fast Forward Tape			•	•	•	•	•			•							•	•					
Rewind Tape			•	•	•	•	•			•									•	•			
Pause Tape			•	•	•	•	•			•				•									
Record Tape	•														•	•							
V tapeRemaining																							
>= 1			•		•	•						•				•		•	•				
< 1 && tapeRemaining > 0		•		•		•				•					•		•		•				
<= 0		•		•			•			•					•		•			•			
V vcrPowerStatus																							
off																					•		•
on	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	
OUTPUTS																							
L topLogicTable.outputState																							
Stop Tape			•		•		•	•			•	•						•		•			
Play Tape				•																			
Fast Forward Tape		•																					
Rewind Tape						•																	
Pause Tape									•					•									
Record Tape											•												
V tapeRemaining																							
+= .00390625				•							•				•								
+= .015625		•															•						
-= .015625						•													•				
C functionDisplay.text																							
Stop			•		•		•	•			•	•											
Play				•																			
F. Forward		•																					
Rewind						•																	
Pause									•					•									
Record											•												
PRIMITIVES																							
O tapePositionTimer.start		•		•		•					•												
O tapePositionTimer.stop																							
V vcrPowerStatus																							
off																						•	
on																					•		
C powerStatus.background																							
0,0,0																						•	
2551,0,0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		

Figure 7.14. The unique table of the ADEPT model of the VCR example.

case, the system just switches in the play mode. In the second situation, the system just transitioned into the play mode but then automatically moved to the rewind mode. Such kind of silent transition is precisely one cause of surprise by the operators of interactive systems. For that precise example, it is maybe not so harmful but it clearly shows that something can happen inside the system, without warning the user. One solution to avoid that potential surprise is to add an observation whenever the VCR is moving from the play mode to the rewind one.

7.4 The Autopilot Model

Experiments using the techniques proposed in this thesis have been performed on a realistic model of a Boeing 777 autopilot. The autopilot has been modelled using ADEPT and partially represents the behaviour of a large subset of the real autopilot. The full autopilot ADEPT model has a total of 38 logic tables shown on Figure 7.15. Two of the 38 logic tables are used to represent user's tasks, which is an experimental feature of ADEPT; they are not considered in this work. The arrows linking the logic tables represent the call relation. Three major parts can be identified in the model, namely one for the lateral aspect, one for the vertical aspect and finally one for the airspeed aspect. The logic tables belonging to those three categories are identified with different grey-level background colours.

Lateral aspects is related to the heading of the aircraft, that is, the direction that the nose of the aircraft is pointing to. Vertical aspects is related to the altitude of the aircraft, and to the vertical navigation mode, which includes how the aircraft is instructed to climb or descend. Finally, airspeed aspects is related to the speed of the aircraft, that can be controlled either in knots or in machs.

7.4.1 Autopilot Model Characteristics

As presented in the previous section, only the system tables are considered when transforming an ADEPT model with the ASF structure to an HVS. The ADEPT autopilot model has a total of 15 logic tables that are classified as system tables. Among those logic tables, the deadAp-
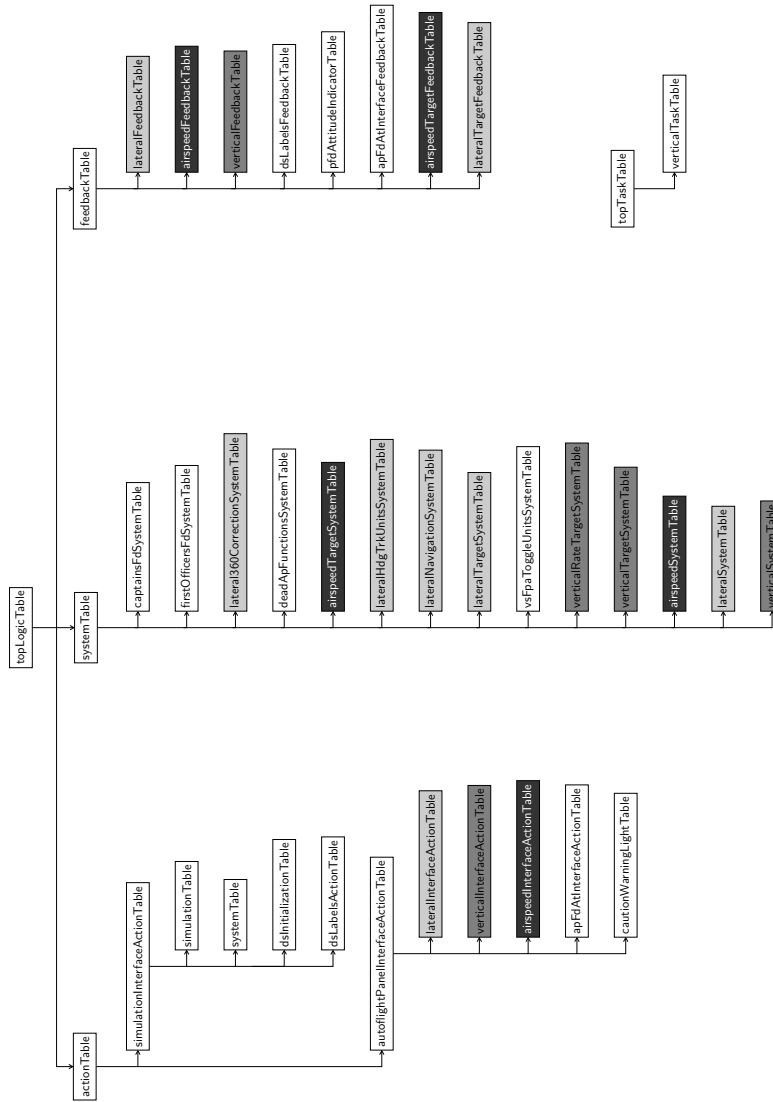
Figure 7.15. Logic tables of the ADEPT autopilot model. Arrows indicate call relation between logic tables, respecting the calls order. The grey-level background colours highlight the three main parts: lateral, vertical and airspeed aspects.

FunctionsSystemTable logic table has been ignored since it only contains functions that have not been implemented in the actual ADEPT model. Moreover, the lateral360CorrectionSystemTable logic table has also been ignored since its only role is to compute a modulo to keep the value of the lateralDirection variable between 1 and 360, which is already ensured by the domain limitations that have been performed on the values of some system variables. Given those two additional restrictions, in addition to the main systemTable, only 12 system tables are taken into account for building the HVS model:

1. captainsFdSystemTable
2. firstOfficersFdSystemTable
3. airSpeedTargetSystemTable
4. lateralHdgTrkUnitsSystemTable
5. lateralNavigationSystemTable
6. lateralTargetSystemTable
7. vsFpaToggleUnitsSystemTable
8. verticalRateTargetSystemTable
9. verticalTargetSystemTable
10. airspeedSystemTable
11. lateralSystemTable
12. verticalSystemTable

The commands of the system are defined according to the outputState of the referred action table. The reduced autopilot model features 20 different possible commands, corresponding to the manipulation of knobs, thumbwheels, buttons and switches. Knobs can be pressed and rotated clockwise and counterclockwise. Thumbwheels can be rotated up and down. Buttons can be pressed and finally switches can be toggled.

- airspeed selector knob   (airspd, airspd↻, airspd↺)
- lateral target selector knob   (lattgt, lattgt↻, lattgt↺)
- altitude selector knob   (altsel↻, altsel↺)
- vertical rate thumbwheel nose   (vertrthb↗, vertrthb↘)
- Lateral HOLD button   (latHOLD)
- LNAV button   (LNAV)
- altitude HOLD button   (altHOLD)

- FLCH button   (FLCH)
- VSFPA button   (VSFPA)
- IAS Mach Units button   (MACHun)
- HdgTrk Units button   (TRKun)
- VSFPA units button   (VSFPAun)
- captains fd switch   (cptFD)
- first officers fd switch   (foFD)

The reduced autopilot model has a total of 25 system variables among which 12 variables correspond to the outputState variables attached to the system logic tables. The other system variables are mainly integer or floating point numbers.

7.4.2 Independent Subsystems

An *independent subsystem* is a set of tables that are the only tables concerned with a subset of the system variables and that do not depend on other system variables. Such independent subsystems can be analysed separately since they do not interfere with the other logic tables. In particular, these logic tables are independent of the rest.

Such an independent subsystem is the one composed of the two logic tables captainsFdSystemTable and firstOfficersFdSystemTable, subsequently referred to as *the F/D subsystem*. That limited subsystem corresponds to an HVS with four states, two state-values (cpt and fo, one for each outputState variable) and two commands (cptFD and foFD). Figure 7.16 shows the HVS of the F/D subsystem, with the corresponding minimal full-control conceptual model.

The conceptual model is reduced to a single-state model, which explains that no matter in which state the system is, both commands are always possible and executing them lead to states with the same behaviour. Moreover, the state-values are not necessary since they are not used in the conceptual model, that is, cpt and fo have not to be visible to the operator, for him to drive the system safely according to the full-control property.

Since the F/D subsystem is not used by any of the other logic tables, the only contribution that it can bring to the whole system model would be a multiplication of the states of the system, those states being merged

(a) System model of the subsys-
tem.

(b)   Minimal   full-
control   conceptual
model.
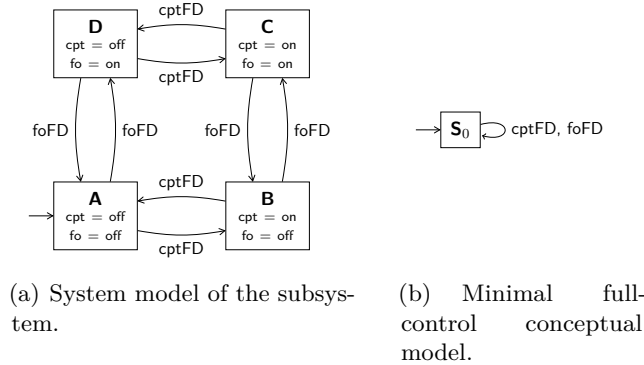
Figure 7.16. Independent subsystem of the autopilot model which manages the F/D (Flight Director) mode setting by the captain and first officer.

together anyway, in the conceptual model. For that reason, the F/D subsystem is also ignored in the analysis of the autopilot case study, reducing the number of considered logic tables to 11.

7.4.3 Reducing the Model

The full autopilot model, and also the version reduced to 11 logic tables, are too large to be analysed by the techniques proposed in this thesis. The main reason is related to the state explosion problem caused by the large number of system variables with large domains. The explicit approach used in this thesis, which requires the system to be completely expanded, does not scale well. Either the ADEPT model is too large for the HVS to be generated, or the obtained HVS is too large to be processed.

To be able to analyse that ADEPT autopilot model with the techniques proposed in this work, several solutions are possible. The experiences presented in this section have been done by only considering fragments of the whole system, as already introduced just here above. Reduced models are obtained by only considering some logic tables, by reducing the domains of some system variables and by omitting some features that do not bring new behaviour but duplicate it for two different units for the mach/knots switch, for example. The last choice will lead to a suppression of input and output switches and columns of logic tables.

In addition to reducing the size of the model, reducing the domain of a variable can also eliminate some behaviour depending on the new domain and the condition the variable is used in. For example, let suppose an integer variable x that appears in one input switch and in one output switch, as shown on Figure 7.17. Let also suppose that the initial value of the variable is 0 and that its original domain is the range $[-180; 180]$. If the domain is, for example, reduced to the range $[0; 25]$, any state change that could have been triggered by the condition < 0 is removed from the reduced model.

INPUTS
x
< 0
$== 0$
> 0
OUTPUTS
x
$++$
$--$

Figure 7.17. Header of a logic table showing an integer variable x that appears in an input and in an output switches.

Abstraction

Another common way that is used to reduce the size of a domain is to eliminate non relevant behaviour as done in *predicate abstraction* [Gd97] of the model. In such abstraction, the set of concrete values that a variable can take is replaced by an approximate abstraction. For example, the domain of the x variable presented in Figure 7.17 can be reduced to three abstract values, following the following three predicates:

- $neg(\text{x}) \iff \text{x} < 0$
- $zero(\text{x}) \iff \text{x} = 0$
- $pos(\text{x}) \iff \text{x} > 0$

When doing predicate abstraction, the states and transitions of the original model are replaced by abstracted versions. Figure 7.18 shows the abstract model corresponding to values of the x variable. Predicate abstraction is an over-abstraction, meaning that additional behaviour

is introduced, that is, all the traces that belong to the abstraction do not correspond to a trace in the original model. Moreover, spurious non-determinism can also be introduced by the abstract transitions, such as visible on Figure 7.18. A direct consequence is that spurious fc-determinism issues can be introduced in the model, making the analyses more difficult, since it has to be checked whether the alerts for non-fc-determinism are effectively present in the original model or not. That is the major reason motivating the abstraction choice made in this work.
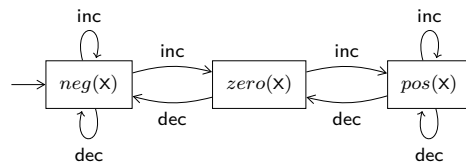
Figure 7.18. Abstract model contains spurious traces that introduce fc-determinism issues.

7.5 Analysis

This section presents the analyses that have been performed on the autopilot case study. The first reduced model shows the role that inhibited commands can play to reduce the size of the minimal full-control conceptual model. The second reduced model illustrates a situation where the generation algorithms succeed to build a minimal full-control conceptual model. This example also shows the role of visible system variables. Finally, the last reduced model exhibits a potential mode confusion, and thanks to the techniques proposed in this thesis, manages to identify why it may happen.

7.5.1 Inhibited Command

The first considered fragment of the full model only covers a very small fragment of behaviour. It consists in only one logic table, the airSpeedTargetSystemTable table, and is restricted to the knots mode, that is, only three columns are taken into account. The logic table is used to manage

the selection of the airspeed by the pilot. Three variables are involved in this model fragment:

- selectedSpeedTarget for the selected speed target   (selSpdTgt)
- airspeedTarget for the airspeed target   (airSpdTgt)
- airspeedTargetOutput for the outputState of the table   (airSpdTgtOut)

The two first system variables are integer numbers and have 250 as initial value, and are ranging in the interval $]245, 246, \cdots, 255[$. They have been limited between 248 and 252 knots in the reduced model, so as to reduce the size of their domain to five different values, and to stay around the initial value. Doing so does not reduce the behaviour covered by the logic table since the two cases related to the selectedSpeedTarget system variable are still covered with the reduced range. Moreover, the alphabet of the system is composed of two commands (airspd↻ and airspd↺). Figure 7.19(a) shows the reduced airSpeedTargetSystemTable logic table that has been used in this first experiment.

The HVS generated from the ADEPT table has nine states and sixteen transitions, all being commands (meaning that there is no internal transitions). Figure 7.20(a) shows the obtained HVS. Only the selectedSpeedTarget and airspeedTargetOutput system variables are shown to keep the figure readable. Indeed, both selectedSpeedTarget and airspeedTarget always have the same value since, the last output switch is the only one concerned with the update of airspeedTarget and makes both system variables equal.

Considering that there is no state-values, that is, the value of the three system variables is invisible to the user, a minimal full-control conceptual model can be generated. Figure 7.20(b) shows the obtained conceptual model which has five states and eight transitions. The system cannot be further reduced since the two extreme states (S_2 and S_4) can both only perform one of the two commands of the system, and they must therefore be distinguishable. By propagation, all the other states must also be separated and thus cannot be merged.

Analysing more carefully the logic table of the system shows that it is in fact not well-formed. The situation where the operator executes the airspd↻ command, while the value of the selectedSpeedTarget variable is greater than 255, is not covered by the logic table. One solution to solve that problem in order to get a smaller conceptual model is to

airspeedTargetSystemTable	0	1	2	3
L **airspeedTargetSystemTable**				
INPUTS				
L airspeedInterfaceActionTable.outputState				
user rotates Airspeed selector knob clockwise	•			•
user rotates Airspeed selector knob counterclockwise		•		•
no action			•	
V selectedSpeedTarget				
< 255	•			
> 245		•		
OUTPUTS				
L airspeedTargetSystemTable.outputState				
increase IAS airspeed target	•			
decrease IAS airspeed target		•		
V selectedSpeedTarget				
++	•			
−−		•		
V airspeedTarget				
V selectedSpeedTarget	•	•		

(a) Logic table of the reduced version.   (b)

Figure 7.19. A reduced version of the airspeed target system table which manages the selection of the airspeed by the pilot. Only the behaviour related to knots has been kept.

add one column with the two commands airspd↻ and airspd↺, with the selectedSpeedTarget switch empty and with an empty output part, so as to define an inhibited event such as defined in Section 7.2.3. That additional column is shown on Figure 7.19(b) on page 230.

Another solution to get a simpler conceptual model without changing the structure of the system model is to make visible the value of the selectedSpeedTarget variable. By doing so, the minimal full-control conceptual model obtained is a single-state model as shown on Figure 7.21.

7.5.2 A First Conceptual Model

The first consequent fragment of the model that has been considered covers four logic tables. In addition to the one used for the experiment
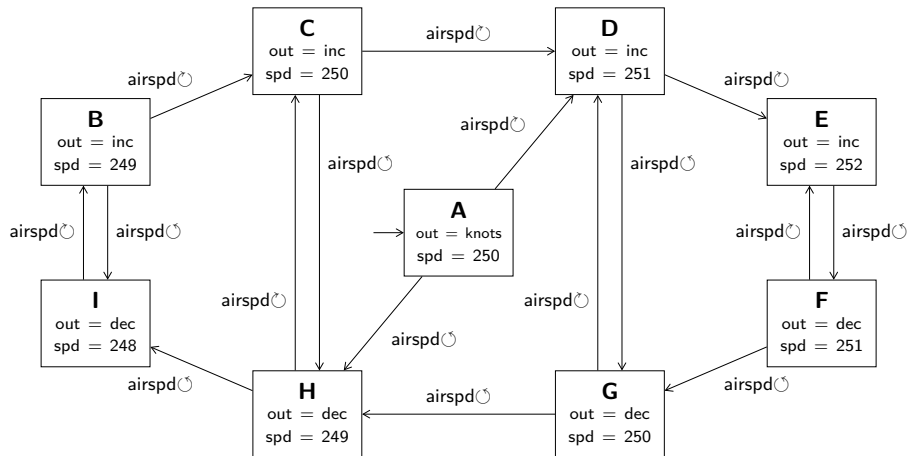
(a) HVS of the system (out corresponds to airspeedTargetOutput and spd to selectedSpeedTarget).
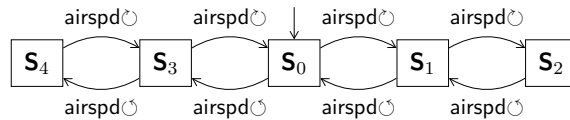
(b) HVM of the minimal full-control conceptual model.

Figure 7.20. Reduced model of the autopilot only composed of the reduced airspeedTargetSystemTable logic table of Figure 7.19, with the corresponding minimal full-control conceptual model.
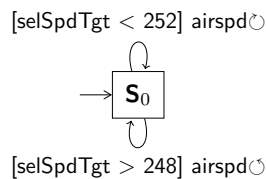
Figure 7.21. Making the value of the selectedSpeedTarget visible to the operator makes it possible to get a smaller full-control system abstraction.

just described in the previous section, the three following logic tables have been added, with some restrictions for the third one:

- lateralTargetSystemTable

- airspeedSystemTable

- lateralSystemTable   (*except scenarios where simulation is running*)

Whereas the previous experiment is only concerned by behaviour related to airspeed aspects, this experiment mixes behaviour related to airspeed and lateral aspects.

Ten system variables are implicated in this fragment model, in addition to the three that were already present in the previous experiment. Some of those variables have been additionally bounded, that is, their values have been forced to stay in a fixed arbitrary interval.

- lateralDirection (latDir) and indicatedAirspeed (indAirspd) always have the same value in that fragment of the model (respectively 180 and 250), since they are not updated by the selected logic tables

- lateralTarget (latTgt), selectedLateralTarget (selLatTgt) and preselected-LateralTarget (preselLatTgt) are bounded between 178 and 182

- lateralTargetError (latTgtErr) is not bounded and selectedLateralTargetError (selLatTgtErr) is bounded between -3 and 3

- The outputState variables for the three tables: lateralOutput (latOut), lateralTargetOutput (latTgtOut) and airspeedOutput (airSpdOut)

The generated HVS has 7680 states and 66242 transitions and the alphabet is composed of 9 commands. Among the transitions, 57545 are labelled with commands and 8697 are internal τ-transitions. For the first experiment, all the states of the HVS are labelled with the same state-value, that is, none of the system variables are visible. The obtained full-control conceptual model HVM has 25 states and 180 transitions.

The generation algorithm succeeded to generate a minimal full-control conceptual model. It means that the system model is fc-deterministic, and that the τ-transitions are not harmful for full-controllability. Looking more carefully at the model reveals that there are in fact two kinds of τ-transitions in this model: 3672 of them are τ-loops and the remaining are all related to one particular value for the outputState variable of the

lateralTargetSystemTable. Either the value of the lateralTargetOutput variable is automatically changing to Lateral Target is Static, while the other system variables are unchanged, or the value of lateralTargetOutput is and remains Lateral Target is Static while some other system variables have changed. Figure 7.22 illustrates those two last situations.

<center>(a)   (b)</center>
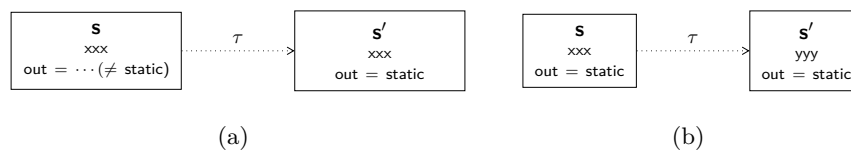
Figure 7.22. Two kinds of internal transitions in the first experiment.

Since the lateralTargetOutput variable does not appear in any input switch of the considered logic tables, states s and s′ corresponding to the situation from Figure 7.22(a) can both execute exactly the same actions, with the same resulting state. The consequence is that s and s′ are fc-compatible. The same observation can be done for the second kind of τ-transition, but the fc-compatibility is this time a consequence of the reduced range for the system variables, that makes the same actions possible in s and s′.

Varying the Visible System Variables

Table 7.1 summarises experiments where different sets of visible system variables have been chosen for the HVS, and the number of states and transitions of the obtained reduced HVM.

- The two first examples show that making visible the latDir or indAirspd system variables does not change the size of the generated HVM. It is just a consequence of the fact that they always have the same value, in this fragment of the model.

- Examples 3 to 5 make visible the airSpdTgt, selSpdTgt or both system variables. Making so does reduce the number of states of the generated HVM, and the reason is exactly the same as for the fragment model analysed in the previous section. The five different state-values correspond to the five possible values of the domain of the system variables $(248, 249, \cdots, 252)$.

The size of $\mathcal{L}^v$ is also 5 for the example 5 because both system variables airSpdTgt and selSpdTgt always have exactly the same value.

- Examples 6 to 8 make visible the system variables related to the lateral target. For the three examples, the visible system variable induces five different state-values corresponding to the five possible values of their domain $(178, 179, \cdots, 182)$.

- The example 9 makes visible all the system variables. The generated HVM has only one state and 10270 transitions. The fact that the generated mental model has only one state means that if the operator can observe all the system variables, he can always know whether a command can be performed, only by checking the state-value of the current state, that is, the assignment of the visible system variables. The situation is exactly the same with example 10 where all the system variables, except latDir and indAirspd, are made visible. This is also a direct consequence of the fact that the latDir or indAirspd system variables always have the same value.

	Visible system variables	$|\mathcal{L}^v|$	$|S_H|$	$| \rightarrow_H |$
1	latDir	1	25	180
2	indAirspd	1	25	180
3	airSpdTgt	5	21	180
4	selSpdTgt	5	21	180
5	airSpdTgt, selSpdTgt	5	21	180
6	latTgt	5	46	360
7	selLatTgt	5	146	1080
8	preselLatTgt	5	146	1080
9	All, except outputState variable	1425	1	10270
10	All, except outputState, latDir and indAirspd variables	1425	1	10270

Table 7.1. Number of different state-values and number of states and transitions of the reduced HVM, for several situations where the sets of visible system variables are different.

7.5.3 Analysing Mode Confusion

ADEPT models do not contain explicit information about modes, but the outputState variables associated with the system tables represent in

fact a mode information. It is therefore possible to check whether the operator can always track the mode related to a system table, by using the construction presented in Section 6.4. The mode associated with the airspeedSystemTable and lateralTargetSystemTable both provoke a potential mode confusion issue. Let us look more carefully at the potential mode confusion related to the airspeedSystemTable. The outputState variable corresponding to the system table can take four different values:

- Hold Current Airspeed   (hold)
- Capture Airspeed Target   (capture)
- Maintain Airspeed Target   (maintain)
- Protect Airspeed Target   (protect)

Executing a generation algorithm on the reduced model that has been mode-completed fails with a situation that exhibits an fc-determinism issue illustrated on Figure 7.23. The issue is that the state $\mathbf{S'}_{2480}$ can lead, with the same action airspd, to two states that are not fc-compatible since they do not belong to the same block in that step of the reduction-based generation algorithm.
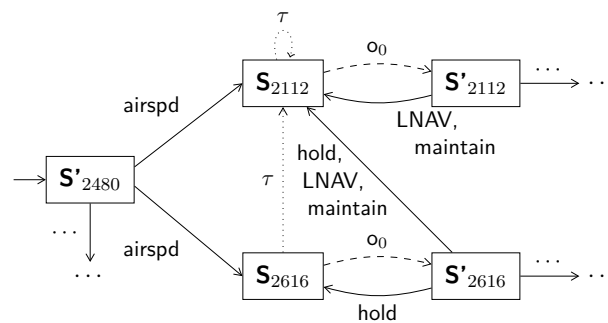
Figure 7.23. Situation exhibiting the fc-determinism issue indicating a potential mode confusion situation. States $\mathbf{S'}_{2112}$ and $\mathbf{S'}_{2616}$ both also have the following commands, leading to the same blocks: lattgt, lattgt↻, lattgt↺, latHOLD, airspd, airspd↻ and airspd↺.

Figure 7.24 shows the reduced airspeedSystemTable that has been used in this experiment. Analysing the situation of Figure 7.23 shows that the issue is that the $\mathbf{S'}_{2112}$ state is in the maintain mode whereas the $\mathbf{S'}_{2616}$

state is in the hold mode. But, given the τ-transition going from $\mathbf{S'}_{2616}$ to $\mathbf{S'}_{2112}$, there is a potential mode confusion between maintain and hold when the system is in the $\mathbf{S'}_{2616}$ state. That automatic mode change is described by the column 1.

In order to solve the potential mode confusion, a solution is to make some system variable visible so that the state-values of the two states $\mathbf{S'}_{2112}$ and $\mathbf{S'}_{2616}$ are different which will imply that the $\langle \mathsf{airspd}, \mathsf{o}_1 \rangle$ trace does not lead anymore from $\mathbf{S'}_{2480}$ to two states with different sets of commands.

	0	1	2	3	4
L airspeedSystemTable					
INPUTS					
V airspeedTarget					
== airspeedTarget && (airspeedTarget - indicatedAirspeed) <= 5 && (airspeedTarget - indicatedAirspeed) >= -5)		•			
> indicatedAirspeed && (airspeedTarget - indicatedAirspeed) > 5			•		
< indicatedAirspeed && (indicatedAirspeed - airSpeedTarget) > 5			•		
V indicatedAirspeed					
< Vmin				•	
<= Vmax && indicatedAirspeed >= Vmin	•	•	•		
> Vmax					•
L airspeedInterfaceActionTable.outputState					
user presses Airspeed selector knob	•				
no action		•	•		
OUTPUTS					
L airspeedSystemTable.outputState					
Hold current Airspeed	•				
Capture Airspeed Target			•		
Maintain Airspeed Target		•			
Protect Airspeed Target				•	•
V selectedSpeedTarget					
V indicatedAirspeed	•				
F minProtectSpeedTarget				•	
F maxProtectSpeedTarget					•
V airspeedTarget					
V indicatedAirspeed	•				
F minProtectSpeedTarget				•	
F maxProtectSpeedTarget					•

Figure 7.24. A reduced version of the airspeed system table which manages the selection of the airspeed by the pilot. Behaviour related to machs has been removed.

Analysing the autopilot case study illustrates several elements. First of all, it shows that the technique proposed in this thesis can be applied to ADEPT models and therefore could be integrated in a tool targeted to system designers. Of course, the semantics and translation algorithm proposed in this work is only a first step towards such an integration. Whereas the top-down direction from ADEPT models to HVSs has been worked on, nothing yet has been performed on the bottom-up direction that would allow to directly related results obtained by the generation algorithms to the logic tables of the ADEPT model. Future work in this direction also includes a more thorough analysis of how to define modes and perform mode confusion analysis.

Chapter 8

Conclusion

The research goal of this thesis is to develop a formal framework that can be used to check whether a given system can be controlled by an operator while avoiding potential automation surprises. Section 8.1 summarises the contributions of this work and assesses how they meet the research goal. Section 8.2 draws up perspectives that may be explored and studied to extend this work and to open new research directions. Finally, Section 8.3 states a final word summarising this whole work.

8.1 Contributions of this Thesis

The starting point which triggered this work is the one by Degani et al. [Deg04, HD07] that proposes a systematic approach to detect potential mode confusion for a given system. Degani et al. themselves based their work on previous work by Rushby et al. [Rus00, Rus02] but brought a more general methodology supported by an algorithm that can be applied to any system for which a model exists. Even though the approach of Degani et al. is systematic and automatic, the focus is on potential mode confusions and the framework is only concerned by a user, the machine being operated and its user interface.

To meet the research goal of this thesis, several aspects necessary to provide a formal analysis framework have been considered. Firstly, the modelling of the elements playing a role in the interaction are considered in Chapter 3. The chosen mathematical formalism is based on an enriched version of labelled transition systems (LTS). The motivation for that choice is to keep the formalism simple enough to decrease the complexity of the analysis algorithms; while keeping it rich enough to be able to model all the aspects relevant allows the analysts to identify the potential wrong interactions to be captured. Since the focus is put on the behavioural

aspects of the interaction in this work, LTS or any equivalent formalism is a good choice since it focuses on the executed actions. Finally, enriched LTS also makes it possible to have observable information on states. This possibility makes it possible to have models that are closer to what the designers are usually working with. It also eases the integration with other formalisms such as statecharts and interactors, or with design languages such as ADEPT tables.

A key aspect of this thesis, and around which all the contributions are revolving, is the full-control property which characterises interactions that are free of potential automation surprises. The definition of the property and its precise characterisation are covered by Chapter 4. The full-control property guarantees that there is a way for the operator to know enough about the system to use it safely, that is, without being surprised ever when using it. Having defined formally a property serves the research goal in the sense that the property can be analysed rigorously. A comparison of the full-control property with other existing properties, mainly coming from the model-based testing of reactive systems field had as outcome a trace characterisation of the full-control property and a proposition of a variant of the full-control property.

Based on the full-control property, three algorithms used to analyse system models are proposed and presented in Chapter 5. Those algorithms automatically generate, from a given system model, a conceptual model that allows full-control of the system. They are the corner stones of this thesis, in the sense that they concretise the research goal. Thanks to those algorithms, a formal methodology has been proposed to check whether potential automation surprises can occur when interacting with a system. If it is the case, the algorithms output feedback information that can be used to redesign the system and otherwise, the algorithms generate a minimal full-control conceptual model that can be used to better understand the system and to produce training material.

The proposed formal framework has been put in context and showed to be practically usable by system designers, based on small realistic examples. Chapter 6 presents how the proposed techniques could be used in the design process. Other examples that have already been studied in the literature have also been analysed to validate the approach. Those experiences are used to illustrate the fact that the full-control property captures a good intuition of *"controllability without surprises"*

of a system. Finally, Chapter 7 presents how the proposed techniques can be used in relation with ADEPT, an existing tool dedicated to the analysis of human-machine interactions. The goal was in fact double: besides the integration of the proposed analysis techniques into ADEPT, the ADEPT model of the autopilot is also used to assess how the proposed techniques work with a realistic real-size problem.

8.2 Perspectives

There are many possible extensions to the work presented in this thesis. A first is the integration of user task models into the analysis. A user is interacting with a system with as main purpose to perform some tasks. In addition to what is presented in Section 6.5, ideas for future work around this thematic have also been proposed in [Com09]. The idea is to have, in addition to the system and mental models, an operational model that represents user tasks. Those user tasks may not be described with the same abstraction level as the system and mental models. Therefore a correct mapping between the actions must be performed, exploiting *action refinement used in model-based testing* [vdBRT05], for example.

Another possible perspective is the study of the robustness of a conceptual model to human errors. Analysing the impact on the interaction when the operator deviates from a nominal behaviour helps to assess whether a given system model is robust enough. For example, a relevant analysis is to check whether the operator can recover from an error. Research has already been done in the domain of analysing the effect of a deviation from a task model [PS02, BB06]. A direction to be worth investigation is to exploit *mutation analysis* [JH11, FDMM94, LDL09]. Given a system and a conceptual model allowing full-control of it, the idea is to mutate the conceptual model and to examine what is the impact on the controllability of the system. Existing mutation operators have to be selected to be relevant to model human errors, and new operators could also have to be defined.

There is also work to be done in the tool support for the techniques proposed in this thesis. First of all, the translation from ADEPT models to HVS is currently performed manually. However the translation is systematic and could be automated. Such automation is necessary to

consider the integration of the proposed framework with ADEPT. The algorithms can also be improved for some of their parts. Indeed, the Paige-Tarjan algorithms or the DFA-minimisation algorithms are examples where the current implementation relies on naive versions which are not necessarily the most efficient ones. The DFA-minimisation algorithm could be improved with the approach described in [RHSJ94] or with the L^* algorithm as proposed in [PO99].

In order to make the proposed techniques scalable, another direction of work could be to explore how conceptual models can be generated compositionally. The idea would be to split the system model into more or less independent parts, and then to find a way to recompose the generated conceptual models to have one for the whole system. One way to explore is to use measures such as the modularity [New06] or other centrality indices [BE05] to identify independent parts of a system. Another possible direction of investigation is compositional model checking [CLM89].

Finally, another promising extension of this work is to consider the integration of the proposed techniques in a real development process. In a similar way as what has been proposed by Campos et al. [CHL04], the generation algorithms could be used in the design process of a new system. The generation algorithms can be run on the system candidate of each design loop, to check whether there are potential automation surprises that could occur. The output of the generation algorithm can then be used in the next iteration of the design process. One another way to use the generated minimal full-control conceptual model is to compare different proposed system models for the same system. The conceptual model can indeed be integrated in a metric, taking for example the number of states, that aims at providing a score to the systems to compare them.

8.3 Final Word

The work presented in this thesis is a new branch of a tree initiated in the mid-1980s. At that time, researchers investigated the use of formal methods to analyse behavioural aspects of human-machine interactions. In the beginning, researchers were focused on the formal and rigorous

analysis of existing accidents. Then, with Rushby et al., a first step towards automation of the analyses was made. Systems implied in accidents were modelled formally and automatically analysed with model-checking techniques. Whereas the approach of Rushby et al.[Rus02] was specific to every analysed accident, but systematic, Degani et al.[DH02] pioneered the domain by bringing a generic approach. They proposed an analysis framework, based on statecharts, that is able to automatically generate user interface for given systems. The work of this thesis started a new branch based on the one of those two groups of researchers. The new branch grew by getting even more general and by providing a solid and rigorous formal base on top of which a formal methodology to analyse potential automation surprises in human-machine interactions, and in particular mode confusion, has been developed and is proposed in this thesis.

Appendix A

Abbreviations and Acronyms

ACM	Association for Computing Machinery
ASF	Action-State-Feedback structure
ADEPT	Automation Design and Evaluation Prototyping Toolset
BFS	Breadth-First Search
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
CTT	ConcurTaskTress
DFS	Depth-First Search
EDT	Event-Dispatching Thread
EOFM	Enhanced Operator Function Model
FSM	Finite State Machine
GUI	Graphical User Interface
HAI	Human-Automation Interaction
HCI	Human-Computer Interaction
HMI	Human-Machine Interaction
HMI-LTS	Human-Machine Interaction Labelled Transition System
HVM	HMI state-Valued Mental model
HVS	HMI state-Valued System model
ICO	Interactive Cooperating Object
ISP	Internet Service Provider
JPF	Java Pathfinder

LTL   Linear Temporal Logic

LTS   Labelled Transition System

MAL   Modal Action Logic

MMI   Man-Machine Interaction

MSC   Message Sequence Chart

MTS   Modal Transition System

NASA   National Aeronautics and Space Administration

OFM   Operator Function Model

PN   Petri Net

SMV   Symbolic Model Verifier

UAN   User Action Notation

UML   Unified Modeling Language

Appendix B

List of System Examples

Appendix C

Algorithms

C.1 Full-control Property Check

In order to be able to compare the sets of possible actions efficiently, the system model should be preprocessed with a τ^*-completion, as described in Section 5.4.1. That preprocessing implies that the interaction model may have more than one initial state and so the set S of the algorithm must be initialized with $\{(s, s_{0_H}) \mid s \in (s_{0_S} \textbf{ after } \varepsilon)\}$.

Algorithm 5: Full-control property check algorithm.

Input: $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S, \mathcal{L}^s, \mathcal{O} \rangle$ a system model
Input: $\mathcal{H} = \langle S_H, \mathcal{L}^c, \mathcal{L}^o, s_{0_H}, \rightarrow_H, \mathcal{L}^s \rangle$ a mental model
Output: **true** if $(\mathcal{H} \textbf{ fc } \mathcal{S})$ and **false** otherwise

$S \leftarrow [(s_{0_S}, s_{0_H})]$
while not $isEmpty\ (S)$ **do**
    $(s, h) \leftarrow removeFirst\ (S)$

    if not $A^c(s) = A^c(h) \wedge A^o(s) \subseteq A^o(h)$ **then**
        ∟ **return false**

    mark (s, h) **as** *visited*
    foreach $(s, \tau, s') \in \rightarrow_S$ **do**
        if not (s', h) **is** *visited* **then**
          ∟ $addLast\ (S, (s', h))$

    foreach $(s, \alpha, s') \in \rightarrow_S$ *s.t.* $(h, \mathcal{O}(s), a, h') \in \rightarrow_H$ **do**
        if not (s', h') **is** *visited* **then**
          ∟ $addLast\ (S, (s', h'))$

return true

C.2 Identification of Pairs of Compatible State

The set of pairs of compatible states is necessary to be able to build a
consistent DFA for a given 3DFA. The algorithm starts by initialising
an implication table and then performs iterations until the table reaches
fixed point.

Algorithm 6: Identification of pairs of compatible state.

Input: $\mathcal{C} = \langle \Sigma, S, s_0, \delta, Acc, Rej, DC \rangle$, a 3DFA
Output: $Comp \subseteq S \times S$, the set of pairs of compatible states

$\mathcal{I} \leftarrow new\_implication\_table(S)$
foreach $X \in S$ **do**
    foreach $Y \in S$ **do**
        if $(X \in Acc$ **and** $Y \in Rej)$ **or** $(X \in Rej$ **and** $Y \in Acc)$ **then**
            mark $\mathcal{I}_{(X,Y)}$ **as** *incompatible*
        else
            mark $\mathcal{I}_{(X,Y)}$ **with** $\{(X',Y') \in S \times S \mid (X,Y) \xrightarrow{\alpha} (X',Y')$
            $\wedge (X',Y') \neq (X,Y) \wedge X' \neq Y'\}$

while not $\mathcal{I}$ **is** *stable* **do**
    foreach $\mathcal{I}_{(X,Y)}$ **do**
        if not $(\mathcal{I}_{(X,Y)}$ **is** *compatible* **or** $\mathcal{I}_{(X,Y)}$ **is** *incompatible*) **then**
            if $\exists (X',Y') \in \mathcal{I}_{(X,Y)}$ **with** $\mathcal{I}_{(X',Y')}$ **is** *incompatible* **then**
                mark $\mathcal{I}_{(X,Y)}$ **as** *incompatible*
            else if $\forall (X',Y') \in \mathcal{I}_{(X,Y)} : \mathcal{I}_{(X',Y')}$ **is** *compatible* **then**
                mark $\mathcal{I}_{(X,Y)}$ **as** *compatible*

return $\{(X,Y) \mid \mathcal{I}_{(X,Y)}$ **is** *compatible*$\}$

C.3 Identification of Compatibles

From a set of pairs of compatible states, that can be represented as an implication table, a set of compatibles must be computed in order to build a consistent DFA for a given 3DFA. The idea is to find the set whose elements are the largest sets of states with all the pairs of states being compatible.

Algorithm 7: Identification of compatibles.

Input: $\mathcal{I}$, an implication table over S
Output: $Comp \subseteq 2^S$, the set of compatibles

$L \leftarrow \{S\}$
foreach $X \in S$ **do**
    foreach $E \in L$ **do**
        if $X \in E$ **then**
            $L \leftarrow L \setminus \{E\}$
            $L \leftarrow L \cup \{E \setminus \{X\}\} \cup \left\{ E \setminus \{Y \in S \mid \mathcal{I}_{(Y,X)} \text{ is } incompatible\} \right\}$
    $eliminate\_redundant\_elements(L)$
return L

C.4 3NFA-completion Completion

An HMI-LTS can be completed for missing transitions so that every action of the alphabet is possible from any state. The result that is computed by the completion is a 3NFA.

Algorithm 8: HMI-LTS 3NFA-completion.

Input: $\mathcal{S} = \langle S_S, \mathcal{L}^c, \mathcal{L}^o, s_{0_S}, \rightarrow_S \rangle$, an HMI-LTS
Output: $\mathcal{N} = \langle \Sigma, S, s_{0_N}, \delta, Acc, Rej, DC \rangle$, a 3NFA

foreach $s \in S_S$ **do**
    foreach $a \in \mathcal{L} \setminus A(s)$ **do**
        if $a \in \mathcal{L}^c$ **then**
            $\rightarrow_S \leftarrow \{(s, a, \Pi)\} \cup \rightarrow_S$
        else
            $\rightarrow_S \leftarrow \{(s, a, \Delta)\} \cup \rightarrow_S$

return $\langle \mathcal{L}, S_S \cup \{\Pi, \Delta\}, s_{0_S}, \rightarrow_S, S_S, \{\Pi\}, \{\Delta\} \rangle$

C.5 3NFA Determinisation

A 3NFA can be determinised in order to obtain a 3DFA. The algorithm is very similar to the classical subset construction used for DFAs, except that rules have to be defined for classifying the states of the determinised model as Acc, Rej or DC states.

Algorithm 9: 3NFA determinisation.

Input: $\mathcal{N} = \langle \Sigma, S_N, s_{0_N}, \delta_N, Acc_N, Rej_N, DC_N \rangle$, a 3NFA
Output: $\mathcal{C} = \langle \Sigma, S_C, s_{0_C}, \delta_C, Acc_C, Rej_C, DC_C \rangle$, a 3DFA

$s_{0_C} \leftarrow s_{0_N}$ **after** ε
$L \leftarrow \{s_{0_C}\}$
while not $isEmpty$ (L) **do**
    $s_C \leftarrow removeElem$ (L)
    $S_C \leftarrow S_C \cup \{s_C\}$

    if $\Pi \in s_C$ **then** $Rej \leftarrow \{s_C\}$
    else if $\exists s'_C \in s_C : s'_C \in Acc$ **then** $Acc \leftarrow \{s_C\}$
    else $DC \leftarrow \{s_C\}$

    foreach $\alpha \in A(s_D)$ **do**
        $s'_C \leftarrow \bigcup_{s \in s_C} s\,\textbf{after}\,\alpha$
        $\delta_C \leftarrow \{(s_C, \alpha, s'_C)\} \cup \delta_C$
        if not $s'_C \in S_C$ **then** $addElem$ (L, s'_C)

return $\langle \Sigma, S_C, s_{0_C}, \delta_C, Acc, Rej, DC \rangle$

Glossary

action guards

An action guard is a condition that is on the transition of an HVM to indicate that an operator will only perform the transition if the state-value of the current state of the system satisfies the condition.

commands

A command is an action that is executed by the user on the system. It corresponds to an input from the system point of view. They are referred to as observed events by Degani et al. and as commanded state transitions by Javaux.

conceptual model

The conceptual model refers to a formal model that represents a model of the system that has to be communicated to the operator. In particular, it refers to the user's model as defined by Norman. In this thesis, it refers to the "perfect mental model" that is generated from the system model by the proposed generation algorithms.

enabled actions

Enabled actions are visible actions that are directly available, that is with a strong transition.

environment

The environment refers to any element that is external to the system, its interface and the operator and that can influence the interaction between the operator and the system.

HMI state-valued mental model

HVM are enriched version of HMI-LTS used to model mental models in this work. They are characterised by action guard on their transitions.

HMI state-valued system model

HVS are enriched version of HMI-LTS used to model system models in this work. They are characterised by state-values attached to their states.

human-machine interaction labelled transition systems

HMI-LTSs are enriched version of LTS used to model system and mental models in this work. They are characterised by their actions that can be commands, observations or internal actions. They are equivalent to LTS/IO and similar to IOTS.

interface

The interface refers to the communication channel that lies between the operator and the system. This thesis considers that it is part of the system model.

internal actions

An internal action is one that takes place in the system without being triggered not observed by the user. Since the user cannot distinguish them, they are all denoted with the same symbol τ. They are referred to as unobserved events by Degani et al.

mental model

The mental model refers to a formal model representing the behaviour of the system, as it lies in the mind of its operator. The mental model can evolve over time as the operator is gaining additional experience and information about the system. Then mental model is not to be confused with the conceptual model.

observations

An observation is an action that is executed autonomously by the system without any intervention of the user, but he can observe

it. It corresponds to an output from the system point of view. They are referred to as observed events by Degani et al. and as uncommanded state transitions by Javaux.

possible actions

Possible actions are visible actions that may be available, that is, are either directly available or become so after a sequence of internal transitions.

state-values

A state-value is an observation that can be done on the current state of the system.

system model

The system model refers to a formal model of the behaviour of the system. In this thesis, they are described with HMI-LTSs or HVSs.

training material

The training material refer to any artifact that is used by an operator to learn how to use a given system. It is a kind of abstraction of the behaviour of the system.

Bibliography

[Ang87]   Dana Angluin. Learning regular sets from queries and coun-
             terexamples. *Information and Computation*, 75(2):87–106,
             November 1987.

[AUU06]   Masakazu Adachi, Toshimitsu Ushio, and Yoshitaka Ukawa.
             Design of user-interface without automation surprises for
             discrete event systems. *Control Engineering Practice*,
             14(10):1249–1258, October 2006.

[BB06]   Rémi Bastide and Sandra Basnyat. Error patterns: System-
             atic investigation of deviations in task models. In Karin
             Coninx, Kris Luyten, and Kevin A. Schneider, editors, *Pro-
             ceedings of the 5th International Conference on Task Models
             and Diagrams for Users Interface Design (TAMODIA 2006)*,
             volume 4385 of *Lecture Notes in Computer Science*, pages
             109–121. Springer, October 2006.

[BBS08]   Matthew Bolton, Ellen Bass, and Radu Siminiceanu. Using
             formal methods to predict human error and system failures.
             In *Proceedings of the Second International Conference on
             Applied Human Factors and Ergonomics (AHFE 2008)*, 2008.

[BBS13]   Matthew Bolton, Ellen Bass, and Radu Siminiceanu. Using
             formal verification to evaluate human-automation interac-
             tion, a review. *IEEE Transactions on Systems, Man, and
             Cybernetics, Part A: Systems and Humans*, 43(3):488–503,
             May 2013.

[BCC+11]   Ann Blandford, Abigail Cauchi, Paul Curzon, Parisa Eslam-
            bolchilar, Dominic Furniss, Andy Gimblett, Huayi Huang,
            Paul Lee, Yunqiu Li, Paolo Masci, Patrick Oladimeji, Atish
            Rajkomar, Rimvydas Rukšėnas, and Harold Thimbleby.
            Comparing actual practice and user manuals: A case study
            based on programmable infusion pumps. In *Proceedings
            of the 1st International Workshop on Engineering Inter-
            active Computing Systems for Medecine and Health Care
            (EICS4Med 2011)*, June 2011.

[BCGS09]   Thomas Anung Basuki, Antonio Cerone, Andreas Gries-
            mayer, and Rudolf Schlatte. Model-checking user behaviour
            using interacting components. *Formal Aspects of Computing*,
            21(6):571–588, November 2009.

[BE05]   Ulrik Brandes and Thomas Erlebach, editors. *Network Anal-
            ysis: Methodological Foundations*, volume 3418. Springer,
            2005.

[BGL+00]   Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech,
            César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad
            Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish
            Tiwari. An overview of SAL. In C. Michael Holloway, edi-
            tor, *Proceedings of the 5th NASA Langley Formal Methods
            Workshop (LFM 2000)*, pages 187–196, June 2000.

[BNP03]   Rémi Bastide, David Navarre, and Philippe Palanque. A tool-
            supported design framework for safety critical interactive
            systems. *Interacting with Computers*, 15(3):309–328, June
            2003.

[Boy11]   Guy A. Boy, editor. *The Handbook of Human-Machine
            Interaction: A Human-Centered Design Approach*. Ashgate
            Publishing Limited, 2011.

[Bri88]   Ed Brinksma. A theory for the derivation of tests. In S. Ag-
            garwal and K. Sabnani, editors, *Proceedings of the IFIP
            WG6.1 8th International Symposium on Protocol Specifica-
            tion, Testing and Verification (PSTV 1988)*. North-Holland,
            1988.

[BSB11]   Matthew Bolton, Radu Siminiceanu, and Ellen Bass. A systematic approach to model checking human-automation interaction using task analytic models. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 41(5):961–976, September 2011.

[But04]   Bettina Buth. Analysing mode confusion: An approach using FDR2. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *Proceedings of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2004)*, volume 3219 of *Lecture Notes in Computer Science*, pages 101–114. Springer, September 2004.

[Cac04]   Pietro C. Cacciabue. Elements of human-machine systems. In *Guide to Applying Human Factors Methods: Human Error and Accident Management in Safety Critical Systems*, chapter 2, pages 9–47. Springer, 2004.

[Car97]   John M. Carroll. Human-computer interaction: Psychology as a science of design. *International Journal of Human-Computer Studies*, 46(4):501–522, April 1997.

[CCJ+10] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.5 User Manual*, 2010. Available under http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87, Berlin, DE, June 2003. Springer.

[CE81]   Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Proceedings of the*

Logics of Programs Workshop, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, May 1981.

[CFC+09]   Yu-Fang Cheng, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating dfa's for compositional verification. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithm for the Construction and Analysis of Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software (TACAS/ETAPS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45, Berlin, Heidelberg, March 2009. Springer.

[CGP99]   Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999.

[CGPF11a] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. A formal framework for design and analysis of human-machine interaction. In *Proceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2011)*, pages 1801–1808. IEEE, October 2011.

[CGPF11b] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. Learning system abstractions for human operators. In *Proceedings of the 2011 International Workshop on Machine Learning Technologies in Software Engineering (MALETS 2011)*, pages 3–10, New York, NY, USA, November 2011. ACM.

[CGPM11]   Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Peter Mehlitz. A JavaPathfinder extension to analyse human-machine interactions. In *Proceedings of the Java Pathfinder Workshop 2011*, November 2011.

[CH97]   José Creissac Campos and Michael D. Harrison. Formally verifying interactive systems: A review. In Michael D. Harrison and Juan Carlos Torres, editors, *Proceedings of the 4th*

International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 1997), pages 109–124. Springer, June 1997.

[CH08]   José Creissac Campos and Michael D. Harrison. Systematic analysis of control panel interfaces using formal tools. In T. C. Nicholas Graham and Philippe A. Palanque, editors, *Proceedings of the 15th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2008)*, volume 5136 of *Lecture Notes in Computer Science*, pages 72–85. Springer, July 2008.

[CH09]   José Creissac Campos and Michael D. Harrison. Interaction engineering using the IVY tool. In Gaëlle Calvary, T.C. Nicholas Graham, and Philip Gray, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2009)*, pages 35–44, New York, NY, USA, July 2009. ACM.

[CH11]   José Creissac Campos and Michael D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3):275–310, August 2011.

[CHL04]   José Creissac Campos, Michael D. Harrison, and Karsten Loer. Verifying user interfaces behaviour with model checking. In Juan Carlos Augusto and Ulrich Ultes-Nitsche, editors, *Proceedings of the 2nd International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS 2004)*, pages 87–96. INSTICC Press, April 2004.

[CJR00]   Judith Crow, Denis Javaux, and John Rushby. Models and mechanized methods that integrate human factors into automation design. In Kathy Abbott, Jean-Jacques Speyer, and Guy Boy, editors, *Proceedings of the 8th International Conference on Human-Computer Interaction in Aeronautics (HCI-Aero 2000)*, pages 163–168, September 2000.

[CLM89]   Edmund M. Clarke, David E. Long, and Kenneth L. MacMillan. Compositional model checking. In *Proceedings of the*

Fourth Annual Symposium on Logic in Computer Science (LICS 1989), pages 353–362. IEEE, June 1989.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, August 2009.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates Inc., May 1983.

[Com09] Sébastien Combéfis. Operational model: Integrating user tasks and environment information with system model. In *Proceedings of the 3rd International Workshop on Formal Methods for Interactive Systems*, pages 83–86, November 2009.

[CP09] Sébastien Combéfis and Charles Pecheur. A bisimulation-based approach to the analysis of human-computer interaction. In Gaëlle Calvary, T.C. Nicholas Graham, and Philip Gray, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2009)*, pages 101–110, New York, NY, USA, July 2009. ACM.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1987)*, pages 178–188. ACM, January 1987.

[CRC07a] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3: The Essentials of Interaction Design.* John Wiley & Sons, May 2007.

[CRC07b] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3: The Essentials of Interaction Design.* John Wiley & Sons, May 2007.

[CseB07] Paul Curzon, Rimvydas Rukšėnas, and Ann Blandford. An approach to formal verification of human-computer interac-

tion. *Formal Aspects of Computing*, 19(4):513–550, October 2007.

[CW96]   Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computer Survey*, 28(4):626–643, December 1996.

[dAH11]   Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference, Held Jointly with the 9th ACM SIG-SOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2011)*, pages 109–120, New York, NY, USA, September 2011. ACM.

[Dan07]   Danby Products Ltd. *Danby Silhouette®, DPAC120061 Model, User manual*, Version 1.12.07.

[DBMD95]   David J. Duke, Philip J. Barnard, Jon May, and David A. Duce. Systematic development of the human interface. In *Proceedings of the 2nd Asia-Pacific Software Engineering Conference (APSEC 1995)*, pages 313–, December 1995.

[Deg04]   Asaf Degani. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, January 2004.

[DFAB03]   Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice Hall, September 2003.

[DH84]   Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.

[DH93]   David J. Duke and Michael D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, August 1993.

[DH00]   Asaf Degani and Michael Heymann. Pilot-autopilot interaction: A formal perspective. In Kathy Abbott, Jean-Jacques Speyer, and Guy Boy, editors, *Proceedings of the 8th International Conference on Human-Computer Interaction in*

Aeronautics (HCI-Aero 2000), pages 157–168, September 2000.

[DH02]   Asaf Degani and Michael Heymann. Formal verification of human-automation interaction. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 44(1):28–43, Spring 2002.

[Dil96]   David L. Dill. The Murϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer, August 1996.

[Dix13]   Alan J. Dix. Formal methods. In Mads Soegaard and Rikke Friis Dam, editors, *The Encyclopedia of Human-Computer Interaction*, chapter 29. The Interaction Design Foundation, Aarhus, Denmark, 2nd edition, 2013.

[DLDvL05]   Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, December 2005.

[DoD84]   DoD. System safety program requirements. Technical Report MIL-STD-882B, Department of Defense, March 1984.

[DR85]   Alan J. Dix and Colin Runciman. Abstract models of interactive systems. In Peter Johnson and Stephen Cook, editors, *Proceedings of the Conference of the British Computer Society Human Computer Interaction Specialist Group – People and Computers I*, pages 13–22. Cambridge University Press, August 1985.

[DS03]   Dan Diaper and Neville Stanton, editors. *The Handbook of Task Analysis for Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., September 2003.

[FDMM94]   Sandra Camargo Pinto Ferraz Fabbri, Márcio Eduardo Delamaro, José Carlos Maldonado, and Paulo Cesar Masiero.

Mutation analysis testing for finite state machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 220–229. IEEE, November 1994.

[Fea07]   Michael Feary. Automatic detection of interaction vulnerabilities in an executable specification. In Don Harris, editor, *Proceedings of the 7th International Conference on Engineering Psychology and Cognitive Ergonomics (EPCE 2007)*, volume 4562 of *Lecture Notes in Computer Science*, pages 487–496. Springer, July 2007.

[Fea10]   Michael S. Feary. A toolset for supporting iterative human–automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, March 2010.

[FM91]   Jean-Claude Fernandez and Laurent Mounier. A tool set for deciding behavioral equivalences. In Jos C. M. Baeten and Jan Friso Groote, editors, *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR 1991)*, volume 527 of *Lecture Notes in Computer Science*, pages 26–29. Springer, August 1991.

[For10]   Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, October 2010. Available under http://www.fsel.com/documentation/fdr2/fdr2manual.pdf.

[For12]   For HOL Kananaskis-8. *The HOL System Description*, October 2012. Available under http://hol.sourceforge.net/documentation.html.

[FP90]   Giorgio P. Faconti and Fabio Patern'o. An approach to the formal specification of the components of an interaction. In C. E. Vandoni and D. A. Duce, editors, *Proceedings of the European Computer Graphics Conference and Exhibition (EUROGRAPHICS 1990)*, pages 481–494. North-Holland, September 1990.

[GC96]   Christian Gram and Gilbert Cockton, editors. *Design Principles for Interactive Software*. Chapman & Hall, June 1996.

[Gd97]   Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, June 1997.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[GM93]   M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, June 1993.

[GV90]   Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In Michael S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer, 1990.

[Har87]   David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HBC$^+$92] Thomas T. Hewett, Ronald Baecker, Stuart Card, Tom Carey, Jean Gasen, Marilyn Mantei, Gary Perlman, Gary Strong, and William Verplank. ACM SIGCHI curricula for human-computer interaction. Technical report, ACM SIGCHI, 1992.

[HD02]   Michael Heymann and Asaf Degani. On the construction of human-automation interfaces by formal abstraction. In Sven Koenig and Robert C. Holte, editors, *Proceedings of the 5th International Symposium on Abstraction, Reformulation and*

Approximation (SARA 2002), volume 2371 of *Lecture Notes in Computer Science*, pages 99–115, London, UK, August 2002. Springer.

[HD07]   Michael Heymann and Asaf Degani. Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 49(2):311–330, April 2007.

[Hen96]   Thomas A. Henzinger. Some myths about formal verification. *ACM Computer Surveys*, 28(4es), December 1996.

[HJS01]   Michael Huth, Radha Jagadeesan, and David A. Schmid. Modal transition systems: a foundation for three-valued program analysis. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP 2011)*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer, April 2001.

[HKB08]   Markus Herrmannsdōrfer, Sascha Konrad, and Brian Berenbach. Tabular notations for state machine-based specifications. *CrossTalk: The Journal of Defense Software Engineering*, 21(3):18–23, 2008.

[HLP97]   Martin G. Helander, Thomas K. Landauer, and Prasad V. Prabhu, editors. *Handbook of Human-Computer Interaction*. North-Holland, 1997.

[Hoa85]   Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, April 1985.

[Hol93]   Erik Hollnagel. The phenotype of erroneous actions. *International Journal of Man-Machine Studies*, 39(1):1–32, July 1993.

[Hol97]   C. Michael Holloway. Why engineers should consider formal methods. In *Proceedings of the 1997 AIAA/IEEE 16th Digital Avionics Systems Conference (DASC 1997)*, volume 1, pages 16–22, October 1997.

[HP85]   David Harel and Amir Pnueli. On the development of con-
             current systems. In Krzysztof R. Apt, editor, *Logics and
             Models of Concurrent Systems*, NATO Advanced Science
             Institute, pages 477–498. Springer, 1985.

[HSH90]   H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The
             UAN: A user-oriented representation for direct manipula-
             tion interface designs. *ACM Transactions on Information
             Systems*, 8(3):181–203, July 1990.

[HT90]   Michael Harrison and Harold Thimbleby. *Formal Methods in
             Human-Computer Interaction*. Cambridge University Press,
             February 1990.

[HV91]   Joseph Y. Halpern and Moshe Y. Vardi. Model checking
             vs. theorem proving: a manifesto. In Vladimir Lifschitz,
             editor, *Artificial Intelligence and Mathematical Theory of
             Computation*, pages 151–176. Academic Press Professional,
             Inc., 1991.

[HV06]   Henri Hansen and Antti Valmari. Operational determinism
             and fast algorithms. In Christel Baier and Holger Hermanns,
             editors, *Proceedings of the 17th International Conference
             on Concurrency Theory (CONCUR 2006)*, volume 4137 of
             Lecture Notes in Computer Science, pages 188–202. Springer,
             August 2006.

[Jac83]   Robert J.K. Jacobs. Using formal specifications in the design
             of a human-computer interface. *Communications of the
             ACM*, 26(4):259–264, April 1983.

[Jav02]   Denis Javaux. A method for predicting errors when interact-
             ing with finite state systems. How implicit learning shapes
             the user's knowledge of a system. *Reliability Engineering
             and System Safety*, 75:147–165, February 2002.

[JH11]   Yue Jia and Mark Harman. An analysis and survey of the
             development of mutation testing. *IEEE Transactions on
             Software Engineering*, 37(5):649–678, 2011.

[Kat05]   Joost-Pieter Katoen. Labelled transition systems. In Man-
            fred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin
            Leucker, and Alexander Pretschner, editors, *Model-Based
            Testing of Reactive Systems*, volume 3472 of *Lecture Notes
            in Computer Science*, chapter 22, pages 615–616. Springer,
            2005.

[KS83]   Paris C. Kanellakis and Scott A. Smolka. Ccs expressions,
            finite state processes, and three problems of equivalence. In
            Robert L. Probert, Nancy A. Lynch, and Nicola Santoro,
            editors, *Proceedings of the 2nd Annual ACM Symposium on
            Principles of Distributed Computing (PODC 1983)*, pages
            228–240, New York, NY, USA, August 1983. ACM.

[Lar90]   Kim G. Larsen. Modal specifications. In Joseph Sifakis,
            editor, *Proceedings of the International Workshop on Auto-
            matic Verification Methods for Finite State Systems*, volume
            407 of *Lecture Notes in Computer Science*, pages 232–246.
            Springer, 1990.

[LDL09]   Jin-Hua Li, Geng-Xin Dai, and Huan-Huan Li. Mutation
            analysis for testing finite state machines. In Ming Li, Fei
            Yu, Jian Shu, and Zhigang Chen, editors, *Proceedings of the
            2nd International Symposium on Electronic Commerce and
            Security (ISECS 2009)*, pages 620–624. IEEE, August 2009.

[Lic60]   J. C.R. Licklider. Man-computer symbiosis. *IRE Trans-
            actions on Human Factors in Electronics*, HFE-1(1):4–11,
            March 1960.

[LP97]   Nancy G. Leveson and Everett Palmer. Designing automa-
            tion to reduce operator errors. In *Proceedings of the 1997
            IEEE International Conference on Systems, Man, and Cy-
            bernetics (SMC 1997)*, volume 2, pages 1144–1150. IEEE,
            October 1997.

[LPS+97]   Nancy G. Leveson, L. Denise Pinnel, Sean David Sandys,
            Shuichi Koga, and Jon Damon Reese. Analyzing software

specifications for mode confusion potential. In C. W. Johnson, editor, *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, March 1997.

[LT87]   Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC 1987)*, pages 228–240, New York, NY, USA, August 1987. ACM.

[LT93]   Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[Luc93]   Paul Jay Lucas. An object-oriented language system for implementing concurrent, hierarchical, finite state machines. Master's thesis, University of Illinois, Urbana-Champaign, Illinois, 1993.

[LvdBC99] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. Statecharts via process algebra. In Jos C. M. Baeten and Sjouke Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR 1999)*, volume 1664 of *Lecture Notes in Computer Science*, pages 399–414. Springer, August 1999.

[McC03]   William McCune. *Otter 3.3 Reference Manual*. Mathematics and Computer Science Division, Argonne National laboratory, August 2003. Available under http://www.cs.unm.edu/~mccune/otter/Otter33.pdf.

[McM00]   Ken L. McMillan. *The SMV system*, 2000. Available under http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps.

[Meh08]   Peter C. Mehlitz. Trust your model – verifying aerospace system models with Java Pathfinder. In *Proceedings of the IEEE Aerospace Conference*, pages 1–11, March 2008.

[Mil80]   Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[ML03]   Mieke Massink and Diego Latella. Deriving manuals from
            formal specifications – extended version. Technical Report
            ISTI-2003-TR-01, C.N.R.-ISTI, March 2003.

[MLP79]   Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis.
            Social processes and proofs of theorems and programs. *Com-
            munications of the ACM*, 22(5):271–280, May 1979.

[MM86]   Christine M. Mitchell and Richard A. Miller. A discrete
            control model of operator function: A methodology for
            information display design. *IEEE Transactions on Systems,
            Man, and Cybernetics*, 16(3):343–357, May 1986.

[MR98]   Florence Maraninchi and Yann Rémond. Mode-automata:
            About modes and states for reactive systems. In Chris Han-
            kin, editor, *Proceedings of the 7th European Symposium on
            Programming, Held as Part of the European Joint Confer-
            ences on the Theory and Practice of Software (ESOP/E-
            TAPS 1998)*, volume 1381 of *Lecture Notes in Computer
            Science*, pages 185–199. Springer, March 1998.

[Mur89]   Tadao Murata. Petri nets: Properties, analysis and ap-
            plications. *Proceedings of the IEEE*, 77(4):541–580, April
            1989.

[New06]   Mark E. J. Newman. Modularity and community structure in
            networks. *Proceedings of the National Academy of Sciences*,
            103(23):8577–8582, April 2006.

[Nor86]   Donald A. Norman. Cognitive engineering. In *User Centered
            System Design: New Perspectives on Human-Computer In-
            teraction*, chapter 3, pages 31–61. L. Erlbaum Associates
            Inc., 1986.

[Nor88]   Donald A. Norman. *The Psychology of Everyday Things*.
            0465067093, June 1988.

[Nor02]   Donald A. Norman. *The Design of Everyday Things*. Basic
            Books, September 2002.

[NPB01]   David Navarre, Philippe Palanque, and Rémi Bastide. Engineering interactive systems through formal methods for both tasks and system models. In *Proceedings of the RTO Human Factors and Medicine Panel (HFM) Specialists' Meeting*, pages 20.1–20.17, June 2001.

[NPLB09]   David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction*, 16(4):18:1–18:56, November 2009.

[Pal95]   Everett Palmer. "oops, it didn't arm." – a case study of two automation surprises. In Richard S. Jensen and Lori A. Rakovan, editors, *Proceedings of the 8th International Symposium on Aviation Psychology (ISAP 1995)*, April 1995.

[Pal97]   Philippe Palanque, editor. *Formal Methods in Human-Computer Interaction.* Springer, 1997.

[PB95]   Philippe Palanque and Rémi Bastide. Verification of an interactive software by analysis of its formal specification. In Knut Nordby, Per H. Helmersen, David J. Gilmore, and Svein A. Arnesen, editors, *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT 1995)*, IFIP Conference Proceedings, pages 191–196. Chapman & Hall, June 1995.

[PB97]   Philippe Palanque and Rémi Bastide. Synergistic modelling of tasks, users and systems using formal specification techniques. *Interacting with Computers*, 9(2):129–153, November 1997.

[Per99]   Charles Perrow. *Normal Accidents: Living with High-Risk Technologies.* Princeton University Press, September 1999.

[Pet62]   Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Rheinisch-Westfäliches Institut für Intrumentelle Mathematik an der Universität Bonn, Schrift Nr 2, 1962. (Also,

English translation, Communication with automata, Griffiss Air Force Base, New York, Technical Report, RADC-TR-65-377, Vol. 1, Suppl. 1, 1966).

[PMM97]   Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In Steve Howard, Judy Hammond, and Gitte Lindgaard, editors, *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT 1997)*, volume 96 of *IFIP Conference Proceedings*, pages 362–369. Chapman & Hall, July 1997.

[Pnu77]   Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, November 1977.

[PO99]   Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(11):1619–1632, November 1999.

[PS02]   Fabio Paternò and Carmen Santoro. Preventing user errors by systematic analysis of deviations from the system task model. *International Journal of Human-Computer Studies*, 56(2):225–245, February 2002.

[PT87]   Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[PU59]   M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8(3):356–367, September 1959.

[QS82]   Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings*

of the 5th International Symposium on Programming, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, April 1982.

[RCP99]   John Rushby, Judith Crow, and Everett Palmer. An automated method to detect potential mode confusions. In *Proceedings of the 1999 AIAA/IEEE 18th Digital Avionics Systems Conference (DASC 1999)*. IEEE, October 1999.

[Rea90]   James Reason. *Human Error*. Cambridge University Press, October 1990.

[Rei81]   Phyllis Reisner. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, 7(2):229–240, March 1981.

[RFM91]   Mark Ryan, José Luiz Fiadeiro, and Thomas Stephen Edward Maibaum. Sharing actions and attributes in modal action logic. In Takayasu Ito and Albert R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS 1991)*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer, September 1991.

[RHSJ94]   June-Kyung Rho, Gary D. Hachtel, Fabio Somenzi, and Reily M. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):167–177, February 1994.

[RS59a]   Michael O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research Development*, 3(2):114–125, April 1959.

[RS59b]   Michael Oser Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[Rus00]   John Rushby. Analyzing cockpit interfaces using formal methods. In Howard Bowman, editor, *Proceedings of the*

Formal Methods Elsewhere Workshop, volume 43 of *Electronic Notes in Theoretical Computer Science*, pages 1–14. Elsevier, October 2000.

[Rus02] John Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering & System Safety*, 75(2):167–177, February 2002.

[Sch04] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[SHM10] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valérie Issarny, editors, *Proceedings of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM 2011)*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, June 2010.

[Shn80] Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, 1980.

[SN93] Nancy Staggers and Anthony F. Norcio. Mental models: Concepts for human-computer interaction research. *International Journal of Man-Machine Studies*, 38(4):587–605, 1993.

[SRP07] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. John Wiley & Sons, January 2007.

[Suc87] Lucy A. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press, November 1987.

[SWB97] Nadine B. Sarter, David D. Woods, and Charles E. Billings. Automation surprises. In G. Salvendy, editor, *Handbook of Human Factors & Ergonomics*, chapter 57, pages 1926–1943. Wiley, 1997.

[TG07]   Harold Thimbleby and Jeremy Gow. Applying graph theory
            to interaction design. In Jan Gulliksen, Morten Borup
            Harning, Philippe Palanque, Gerrit van der Veer, and Janet
            Wesson, editors, *Proceedings of the Engineering Interactive
            Systems Joint Working Conferences EHCI, DSV-IS, HCSE
            (EIS 2007)*, volume 4940 of *Lecture Notes in Computer
            Science*, pages 501–519. Springer, March 2007.

[The13]   The Coq Development Team. *The Coq Proof Assistant
            Reference Manual*, April 2013. Available under http://coq.
            inria.fr/distrib/current/files/Reference-Manual.pdf.

[Thi96]   Harold Thimbleby. Creating user manuals for using in col-
            laborative design. In *Conference Companion on Human
            Factors in Computing Systems*, pages 279–280. ACM, 1996.

[Thi10]   Harold Thimbleby. *Press On: Principles of Interaction
            Programming*. The MIT Press, January 2010.

[TL96]   Harold Thimbleby and Peter B. Ladkin. From logic to man-
            uals. *Software Engineering Journal*, 11(6):347–354, 1996.

[Tre08]   Jan Tretmans. Model based testing with labelled transition
            systems. In Robert Hierons, Jonathan Bowen, and Mark
            Harman, editors, *Formal Methods and Testing*, volume 4949
            of *Lecture Notes in Computer Science*, pages 1–38. Springer,
            2008.

[US94]   Andrew C. Uselton and Scott A. Smolka. A compositional
            semantics for statecharts using labeled transition systems.
            In Bengt Jonsson and Joachim Parrow, editors, *Proceedings
            of the 5th International Conference on Concurrency Theory
            (CONCUR 1994)*, volume 836 of *Lecture Notes in Computer
            Science*, pages 2–17. Springer, August 1994.

[vdBRT05]   Machiel van der Bijl, Arend Rensink, and Jan Tretmans.
            Action refinement in conformance testing. In Ferhat Khen-
            dek and Rachida Dssouli, editors, *Proceedings of the 17th
            IFIP TC6/WG 6.1 International Conference on Testing of
            Communicating Systems (TestComm 2005)*, volume 3502 of

Lecture Notes in Computer Science, pages 81–96. Springer, June 2005.

[vG01]   Rob J. van Glabbeek. The linear time – branching time spectrum i: The semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier, March 2001.

[VHBP00]   Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 3–12. IEEE, September 2000.

[Wes00]   Douglas B. West. *Introduction to Graph Theory.* Pearson, August 2000.

[Wie06]   Freek Wiedijk. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science.* Springer, 2006.

Index