Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Building Deductive Proofs of LTL Properties for Iteratively Refined Systems

Relatore: Prof. Carlo GHEZZI
Correlatori: Prof.ssa Paola SPOLETINI
Dott. Claudio MENGHI

Tesi di Laurea di:
Anna Bernasconi
Matricola 798921

Anno Accademico 2014-2015

To my models,
your claims,
and our intersections.

Abstract

Modern software development processes are evolving from sequential to increasingly agile and incremental paradigms. Verification, unavoidable step of a correct software production, cannot get left behind by this new quickly changing practice. Advances in verification techniques have been considerable in the past years, and feasibility has been achieved on always greater systems. Nevertheless, we believe that verification and modern development processes are still not going at the same pace in terms of incrementality.

Classical verification algorithms are applied when a complete specification of the model to verify is available, and several development costs and efforts have been already spent. Today more than ever, the description of a system changes continuously during the phase of analysis, asking for periodical adjustments in its specifications. Various parts are often only sketched waiting for further enrichment, which is sometimes delegated to third parties. The classical scenario is, therefore, not applicable anymore: it becomes essential coming up with light iterative methods formal verification methods, that can be applied also to incomplete models at each stage of the design and development phases, contributing more incisively to developers choices.

With particular focus on two main verification techniques, model checking and deductive verification, we study a way to integrate them into this incremental context. The idea is to supply each step of the design phase with a way to prove behaviors of incomplete systems or only single components. Stepwise model checking can be augmented by a simple incremental deductive system generator that justifies why the system actually meets some requested temporal specification (if this is the case). This kind of infrastructure can bring a useful contribution in cases and refinements where matters of safety, starvation or liveness are critical, and, in general, guide the choices of the developer that faces different designs.

The main idea is to combine two approaches presented in literature: on one side we would like to exploit a procedure of model checking that supports systems that are not completely specified, on the other side we study a mechanism to build deductive proofs using information gathered during model checking. This thesis deals with the construction of these incremental deductive proofs of linear temporal logic properties in incomplete systems that are completed progressively when the system gets refined.

Sommario

I processi di sviluppo del software, un tempo sequenziali, si stanno ora evolvendo verso paradigmi sempre più agili ed incrementali. La verifica formale, fase fondamentale del processo di produzione del software, deve dunque adattarsi a queste pratiche in continuo cambiamento. I progressi nelle tecniche di verifica sono stati considerevoli negli ultimi anni, soprattutto grazie alla loro attuabilità su sistemi di sempre maggiori dimensioni. Ciò nonostante, notiamo ancora un disallineamento tra le tecniche di verifica ed i processi di sviluppo moderni, in termini di metodologie incrementali.

Gli algoritmi di verifica classici vengono applicati in fasi già conclusive, quando le specifiche fornite sono pressoché complete e diversi costi di sviluppo sono già stati sostenuti. Oggigiorno, però, la descrizione di un sistema durante la fase di analisi è in continua evoluzione e richiede dunque frequenti modifiche alle specifiche. Infatti, diversi moduli vengono spesso solo abbozzati in attesa di un successivo arricchimento o direttamente affidati a terze parti. Lo scenario classico non è dunque più attuale: diventa essenziale elaborare metodi di verifica più leggeri ed iterativi, applicabili anche a modellizzazioni incomplete del software durante qualsiasi fase della pianificazione e dello sviluppo, in modo da contribuire in maniera più incisiva alle scelte degli sviluppatori.

Incentrando la nostra ricerca sulle tecniche di model checking e verifica deduttiva, valutiamo una soluzione per integrarle nel contesto incrementale. L'obbiettivo è quello di affiancare ad ogni fase del design una procedura che prova i comportamenti di interi sistemi incompleti e di loro singoli moduli, fornendo supporto per integrare le informazioni ottenute in modo frammentato. Mostriamo come il model checking incrementale può essere arricchito da un semplice generatore di prove deduttive che, quando il sistema in analisi rispetta un dato requisito, ne giustifica il perché. Questo tipo di supporto può fornire un contributo utile in ambiti in cui proprietà di safety, starvation e liveness sono critiche, e in generale guidare lo sviluppatore software che affronta qualsiasi tipo di design.

In questa tesi combiniamo insieme due diversi approcci provenienti dalla letteratura: da una parte utilizziamo una procedura di model checking che supporta sistemi specificati in modo incompleto [MSG15], dall'altra studiamo il meccanismo di costruzione di prove deduttive sfruttando direttamente le informazioni generate dalla procedura di model checking [PZ01, PPZ01].

Dopo aver introdotto il nostro lavoro nel panorama della verifica formale, presentiamo gli approcci che hanno ispirato il nostro lavoro; poi introduciamo

il nostro contributo, una tecnica di verifica ibrida applicabile in maniera flessibile a specifiche parziali confrontate con requisiti espressi in logica temporale, e risolviamo la problematica di comporre insieme prove deduttive ottenute a diversi livelli di dettaglio. Successivamente, presentiamo la validazione della nostra tecnica ottenuta tramite il tool ChIPS. Infine, chiariamo il nostro approccio tramite un case study che descrive un sistema di invio di messaggi e concludiamo descrivendo lo stato dell'arte ed i futuri sviluppi di questo lavoro.

Ringraziamenti

Questa tesi è nata un po' per caso, da un'idea messa al centro del tavolo e poi lasciata crescere, rotolare, fermare, ripartire, finalmente realizzare. Vederla acquisire un significato è la mia soddisfazione più grande.

Un sentito ringraziamento al Prof. Carlo Ghezzi per la sua disponibilità e alla Prof.ssa Lenore D. Zuck per le sue idee da oltreoceano. Per tutta la pazienza e il supporto, grazie alla Prof.ssa Paola Spoletini, che mi ha guidato per tutto il percorso, durante call interminabili dagli Stati Uniti alle ore più disparate. Grazie a Claudio Menghi, il vero eroe di questo lavoro, per l'infinita motivazione che mi ha trasmesso, fondamentale per chiudere i giochi.

Alla mia famiglia, i miei modelli, dedico tutto. Dall'inizio alla fine. Dalle prime fatiche alle ultime. Ai genitori, che mi sostengono sempre. A Chiari, esempio e punto di riferimento.

Grazie a Giorgia, perché sorride sempre. Grazie ai miei amici, di Bologna e di Milano, a tutti quelli che una volta o l'altra sono passati da Paracelso5 e da Goldoni57. Grazie a Madrid che mi ha dato la libertà e grazie a Chicago che mi ha dato la prospettiva.

Senza rendersene conto, questa tesi racconta del perché, passo dopo passo, possiamo costruire qualcosa che soddisfi tutte le nostre aspettative, e nessuno dei nostri rifiuti, rincorrendo i nostri modi di essere più positivi, senza mai trovarci in un vicolo cieco.

Contents

1 Introduction **1**
1.1 Motivation 4
1.2 Original contributions and structure of the thesis 5

2 Background and used formalisms **9**
2.1 Modeling the system 9
2.1.1 Complete models 9
2.1.2 Incomplete models 11
2.1.3 Refining incomplete models 12
2.2 Formalizing the specification 14
2.2.1 Syntax and semantics of LTL 15
2.2.2 LTL to automata 16
2.2.3 Labeled Generalized Büchi Automata 17
2.3 LTL model checking 20
2.3.1 Checking complete models 21
2.3.2 Checking incomplete models 22
2.3.3 Constraints and refinement checking 25
2.4 Proof of $\mathcal{M}$-validity of property ϕ 26

3 Contribution **33**
3.1 High level outline 33
3.2 Computing the master proof 36
3.2.1 Extending the intersection 40
3.2.2 Identification of strongly connected components 45
3.2.3 Rules writing 47
3.2.4 Rules conjunction 55
3.2.5 Dependency graph 57
3.2.6 Output of the proof 58
3.3 Computing the sub-proofs 59
3.3.1 Intersection for the sub-proof 60
3.3.2 Rules application 64
3.4 Plugging the sub-proofs into the master proof 66

4 Tool support: `ChIPS` **71**
4.1 The `CHIA` tool 71
4.2 A Checker Initializing Proof Systems: `ChIPS` 72
4.2.1 Modeling 72

4.2.2 Input and output 73
4.2.3 Building the proof 76
4.2.4 Initial framework 79
4.3 Interaction with the tool 81

5 Case study **83**
5.1 Master proof building 85
5.2 Computing the sub-proofs 89
5.3 Plugging the sub-proofs into the master proof 96

6 State of the Art **99**
6.1 Modeling incomplete systems 99
6.2 Model checking and incompleteness 100
6.3 Combining model checking and deductive verification 102

7 Conclusions **105**
7.1 Contributions and limits 105
7.2 Perspectives for future work 107

8 Bibliography **109**

List of Figures

2.1 LTL tableau for $\Diamond\Box\neg p$ 20
2.2 An example intersection automaton 30

3.1 Integration of the proof computation in an incomplete model checking framework 34
3.2 Model of railway crossing system 39
3.3 Negation of the property for the railway crossing system 39
3.4 LTL tableau for $\neg lowR\neg out$ 39
3.5 Example of nodes collapsing 41
3.6 Failed nodes generation: possible cases 43
3.7 Semantics of intersection automaton 45
3.8 Intersection automaton for the railway crossing example 45
3.9 Schematization of one-node-SCC cases 50
3.10 Schematization of one-mixed-node-SCC cases 51
3.11 Artificial initial node choice 55
3.12 Replacements for the railway crossing example 62
3.13 Sub-properties for the railway crossing example 63
3.14 Intersection automaton for $\mathcal{R}_{q_2}$ and its sub-property 63
3.15 Intersection automaton for $\mathcal{R}_{q_4}$ and its sub-property 64
3.16 Tree of dependencies between proofs 67

4.1 The class diagram of the modeling classes 72
4.2 The class diagram of the `chips.io.in` package 74
4.3 The class diagram of the `chips.io.out` package 76
4.4 The class diagram of the `chips.prover` package 77
4.5 The class diagram of the `chips.rule` package 78
4.6 The class diagram of the `chips.row` package 79
4.7 The class diagram of the `chips.framework` package 80

5.1 Model of the system in charge of sending a message 83
5.2 Automaton representing the negated claim 84
5.3 LTL tableau for $\neg\Box(send \rightarrow \Diamond success)$ 84
5.4 Intersection automaton for the sending message example 85
5.5 Graph analyzed for the rule generation 86
5.6 Sub-properties for the sending message example 90
5.7 Replacement for state $send_1$ 91
5.8 First extention step for automaton $\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1}$ 91

5.9 Second extention step for automaton $\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1}$ 92
5.10 Replacement for state $send_2$ 94
5.11 Extention of intersection automaton $\mathcal{R}_{send_2} \cap \bar{\mathcal{S}}_{send_2}$ 95

List of Algorithms

3.1 Model checking with deductive proof 36
3.2 Deductive proof construction 38
3.3 Extension of intersection automaton 40
3.4 Sorting of SCCs 46
3.5 Choice of rule for each SCC 49
3.6 Apply RuleFail to SCC 52
3.7 Apply RuleSucc to SCC 53
3.8 Apply RuleInd to SCC 54
3.9 Apply RuleConj to conclude the proof 56
3.10 Extension of intersection automaton for the sub-proof 60

Listings

3.1 Deductive proof of $\mathcal{M} \models \phi$ for the railway crossing system . . . 58

4.1 XML file corresponding to the claim BA presented in Figure 3.3 74
4.2 XML file for the model IBA presented in Figure 3.2 75

5.1 Deductive proof of $\mathcal{M} \models \phi$ for the sending message system . . . 89
5.2 Deductive sub-proof of $send_1 \models \phi$ 93
5.3 Deductive sub-proof of $send_2 \models \phi$ 96

List of Tables

2.1 Expansion rules of LTL tableau 18

3.1 Resolution of dependencies. Steps 1-3 68
3.2 Resolution of dependencies. Steps 4-5 68
3.3 Resolution of dependencies. Steps 6-8 69

5.1 Resolution of dependencies - step 1 97
5.2 Resolution of dependencies - step 2 98
5.3 Resolution of dependencies - step 3 98

List of Acronyms

AP	Atomic Propositions
BA	Büchi Automaton
CTL	Computational Tree Logic
DFS	Depth First Search
FSA	Finite State Automaton
FSM	Finite State Machine
HSM	Hierarchical State Machine
IBA	Incomplete Büchi Automaton
ILTS	Incomplete Labelled Transition System
KMTS	Kripke Modal Transition System
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MTS	Modal Transition System
SCC	Strongly Connected Component

1 Introduction

Software systems are usually produced through a sequence of development steps. These transform the initial, high level model of the system into the final artifact. Producing a correct software is becoming a cumbersome activity. Regardless of the used development technique, software is usually never correct at the first attempt: the final version is obtained by evaluating different design decisions and comparing the behaviors of components that can be used. Furthermore, software systems are rapidly growing in their functionality and scale, increasing this way the probability of errors, and the considerable damage caused by them. Because of the mentioned reasons, verification becomes essential to the software creation process.

Literature offers a wide range of approaches to supervise software behaviors, aimed at enhancing the quality and reliability of systems. Most of these approaches go under the name of *formal methods*. They are *formal* in the sense that they use a number of mathematical theories such as logic, automata and graph theory, to name some of them. They provide theories, techniques, and tools that support modeling and analysis of software systems under development. Modeling allows the developer to describe the system with respect to the properties he/she is interested in verifying. Analysis helps software engineers in checking the correctness, reliability, and robustness of their planned designs.

Formal methods can be used at different stages of the software development cycle: at the beginning, when an high level design of the system is considered, or at the end, when the final implementation of the system is available. Since their effectiveness tends to diminish with the size of the analyzed object, it is preferable to perform formal methods at early stages of the process, when the formalization of the system and the checked modules are still small, and errors cheaper to fix.

A wide area of formal methods is covered by *verification methods*. With *verification* we mean the process of applying a manual or automated technique that is supposed to establish whether software possesses properties of interest. Among others, model checking and theorem proving are two well-established techniques of verification. If effectively integrated at an early stage, they guide the design process towards a reliable system and allow to save considerable expenses before products are realized.

Model Checking was pioneered by Clarke and Emerson [CE82] and by Quielle and Sifakis [QS82] independently in the 1980s. It is an automatic

technique that verifies the model of a finite state system against its specification, expressed as a logic formula. The model describes which behaviors the system may exhibit, whereas the property dictates their peculiarities. The model checker exhaustively explores all possible behaviors of the model in a systematic manner, to verify if they match the property of interest. If a behavior that violates the property is found, it is returned as a counterexample. The system model can be automatically generated from the implementation of the system or outlined by hand. The property is often expressed as a temporal logic formula.

Theorem Proving, or *Deductive Verification*, is an alternative approach to model checking. It requires expressing both the system and the requested property as mathematical logic statements. The starting point is a *formal system*, where a set of axioms and inference rules are defined. The goal is to derive that the property is a theorem of the checked model, i.e., given the statements of the model, it is demonstrated that the property holds, by performing the steps of the proof. The main benefit of deductive verification is the possibility to actually explain *how* the system meets its specification. Theoretically, this method can be applied to any model and specification, only limited by the mathematical skills of the user. In practice, proving a program requires checking the validity of a great number of statements. This technique has therefore flourished thanks to automated theorem provers.

From falsification to verification Model checking and deductive verification are usually considered as very different methods, with different application contexts: the first one works to exclude some behavior we do not want a system to take on; the second works to justify why a system follows some positive behavior. On one side we are proceeding by "falsification", on the other by "verification" [PPZ01].

In the traditional version of model checking, a positive output appears to have a slightly "weaker" justification than a negative one. When the answer is negative, this is supported by a counterexample that shows how the system violates the requirement, but there is no additional explanation of a positive answer. What is missing is a way to show how the search for a counterexample has failed.

Model checking is usually used to find counterexamples and, therefore, identify faults in the design. A dual approach consists of generating a deductive proof of the fact that there are no counterexamples, i.e., the system does in fact satisfy the specification. In [PZ01] and [PPZ01], Peled et al. proposed a technique to do this: while performing model checking, by exhaustively searching the state space of the model, they collect the source material to feed an automated theorem rules generator, to prove the specification in that model.

From complete to partial specifications Software development processes are constantly evolving towards more flexible procedures, due to the need of accommodating changing user requirements and of reducing the time of the product release. With regard to this, there still seems to be a substantial mismatch between verification and development processes, that highly focus on agile and incremental, iterative methods to create software. Formal verification techniques still assume that the formalizations of systems should be completely available before they are applied. This is, unfortunately, not the case. Recent development of software engineering calls for agile methods of verification able to support development at each step. While sequential, waterfall models only allowed for checks performed at the end, new development cycles ask for techniques that can also deal with incomplete specifications. More precisely, the goal is to make verification techniques follow this mainstream and be more flexible, modular and incremental, going at the same pace as all other steps of the software life-cycle. The benefit of such techniques is that the unavailable modules can be developed and verified independently from the system.

In literature, several contributions to software verification have dealt with incompleteness in its various forms. We consider in particular [MSG15], that introduces an incremental model checking procedure, integrated with a way to initially specify models that supports incompleteness, a useful concept to deal with cases where some functionalities might be developed later or by third parties. They allow encapsulation of software sub-parts into unspecified components. At the time of replacing these with known descriptions, they only need to be checked with the constraints previously computed.

Thesis statement Verification techniques require a solid basis of logic and mathematical reasoning. Despite the utility of logic tools in computer science is unquestionable, it still deals with skepticism of practitioners for being too formal and leading to a theoretic approach worthless to the world of software development. Logic proceedings are not fully appreciated yet; the main stream goes nowadays in another direction. Recently, though, more promising signals are arriving from the communication and the hardware industries, that, where reliability is critical, are starting to use formal methods, or even develop their own. We believe that, in a field that is not strictly scientific but conditions almost every action of our everyday lives, some precise and formal methods will eventually be appreciated. Our work brings some insight on techniques and uses of logic verification that should start to ease off the practitioners hesitancy.

Considering the *pros* and *cons* of model checking and deductive verification, this thesis proposes an approach to integrate the two mentioned verification techniques in the context of incompletely specified systems, that are nowadays needed to allow an agile development process of software.

In Section 1.1 we deal with observations that inspired our research and in Section 1.2 we analyze in detail the original contributions of this thesis within the context of combined and modular verification techniques for software systems.

1.1 Motivation

Our research is mainly justified by the following statements:

- *Verification is based on a given model.* Models usually describe systems up to some level of abstraction. The model checking result "is only as good as the model of the system" [BK08]. It is effective in exposing at the developer attention potential design errors, but the failure to find counterexamples does not necessary imply the correctness of the real system, that could have been incorrectly described, or oversimplified.

Idea. Model checking techniques provide an answer based on the search for a counterexample; whenever a property is satisfied model checking tools do not provide any justification. On the contrary, deductive verification explains why a given model satisfies a property by providing a proof that the property of interest is a theorem for that model.

- *The model checker can contain software faults.* Despite its use, we need not to forget, it is a software itself.

Idea. Since the modeling process is itself subject to errors, it can easily happen that the model checking procedure is inaccurate. This asks for some method that supports model checking by detailing and describing which behaviors the procedure is taking into account and which not. The need for a proof is motivated by the fact that "intuition often fails to grasp the full intricacy of the algorithm" [PZ86].

- *By-hand proofs can be ambiguous and subjective.*

Idea. Algorithmic methods are generally preferred over the ones that require considerable human skills. Manually building proofs is time consuming and also error prone. Furthermore, at each refinement, or after any change, manually building the proof from scratch can be discouraging. On the other hand, the automated generation of the proof is convenient and fast. It also needs human supervision but offers a good starting point. We look for a method to collect information during the model checking procedure to feed the automated theorem rules generator. This way, not only the resolution of the proof will be automated, but also its setup.

- *Model checking does not support generalization and parametrization of systems.* In its classical definition, it does not treat the possibility to analyze incomplete models.

Idea. Model checking in its classical definition might return answers not as general as the user might hope. This speaks in favor of an *incremental refinements* approach supported by a modular model checking technique. This would allow more flexibility to the initial design and guarantee a checking procedure that follows the changes in the model step by step. We would like our method to support *evolutionary system development*, by allowing partial specification and analysis of selected aspects of a system.

- *Once-in-a-lifetime verification is not agile.* It is often difficult, even impossible, to apply formal methods to a complete system.

Idea. The study of a *compositional* technique to perform deductive verification inspired our work. We attempt to verify different parts of the system separately, and then make conclusions on the whole program. A modular way of doing verification could bring back the importance it deserves. Modularity can, on one side, be something less painful to those that were never able to appreciate it, and on the other, assist in a better way the ones that were always indulgent and purposeful with it, in spite of its slow and unfriendly functioning.

- *No formal method is likely to be suitable for analyzing every aspect of a complex system.*

Idea. Combining two different approaches can be a practical solution to this issue, ideally benefiting from the advantages of both. On one side the conciseness and effectiveness of the negative answer given by model checking, on the other the advantage of proofs to be more explanatory.

To conclude, the union of these motivations makes us see the utility of supplying a tool to fill that empty space left by the too concise answers of model checking with a complete and convincing argument of the validity of a property in a system that gets refined over time.

1.2 Original contributions and structure of the thesis

This work was conceived from the union of two research ideas. On one side, in [MSG15], Menghi et al. investigate different ways to making software verification techniques more agile, specifically through a modular procedure of model checking. On the other side, [PZ01, PPZ01] describe research on enhancing the linear temporal logic model checking process with additional features, by exploiting its information to justify model specification properties.

We believe that joining the purposes of the mentioned approaches in a common direction could lead to interesting and original content.

More precisely, the contributions of this work include:

- A deductive proof verification approach which supports incomplete models. In this framework the proof must justify why a model *satisfies* or *possibly-satisfies* (depending on later refinements) a specific property;

- A technique that is able to manage these proofs in an incremental way. Whenever an incomplete model is refined, the new proof is not built all over again but is obtained by computing a set of sub-proofs which are plugged into the initial master proof;

- A case study which exemplifies the applicability of the approach and a prototype tool that validates the contribution of this thesis.

The final goal is to support agile programming techniques with, not only incremental model checking, but also a simple deductive system generator that proves a property in a given system. One may consider this infrastructure to be only necessary in cases and refinements in which it is advisable to formally explain why the system satisfies a requested property (usually safety/starvation-critical) but we believe any developer would benefit from it.

Taking into consideration the limits of both model checking and deductive verification, an hybrid approach, intersecting the advantages of both, can also help using none as a black box giving out a result, and both as guides to understand the mechanisms of the modeled system, giving insight on the real one.

The thesis is organized as follows:

- Chapter 2 contains the concepts necessary to understand the basic ideas of our approach, outlined in the following chapter. It presents preliminary notions on the selected formalisms used to model systems and their required properties. In particular, we introduce automata and linear temporal logic syntax and semantics. Then, we provide a background on the two verification techniques we focus on: model checking and deductive verification, exploiting the deductive capability of logic of inferring new facts from given facts;

- Chapter 3 describes our proposed approach to incremental proofs of incomplete systems with particular attention to the steps of the procedure described in [MSG15] that we extended to create the proofs, and to the steps of [PZ01], that we modified to support incomplete systems. We show how to combine the strength of the two different approaches considered;

- Chapter 4 briefly describes ChIPS, a Java module implemented to validate our proposal of algorithm. ChIPS extends the tool CHIA developed by [MSG15][1];

[1]CHIA tool is available at http://home.deib.polimi.it/menghi/Tools/IncModChk.html

- Chapter 5 presents a well known case study from literature [AY01], used to analyze and better explain our approach. It represents a sending message system on which we prove a liveness property of interest for sequential systems;
- Chapter 6 contains an overview of the state of the art of the topics related to this work. It includes a general section on the problem of automatizing verification, a section on model checking and one on deductive verification. Then, it presents the literature that integrates these two approaches in various ways. Finally it shows how modularity and incompleteness have been dealt with in both model checking and theorem proving;
- Chapter 7 reports the conclusion of our work with a discussion of the method presented and suggests new directions of research.

2 Background and used formalisms

This chapter provides the reader with the background notions necessary to understand this thesis. First, Section 2.1 describes formalisms that allow to model the system under development, and explains how these models can be iteratively refined. Section 2.2 specifies how it is possible to express the requirements of the system under development, with a particular focus on functional properties. Section 2.3 explains how it is possible to check whether the system possesses the properties of interest. We consider the automata-based model checking procedure, with particular attention to the intersection automaton that is built. A distinction is made between two semantics for complete and incomplete systems. Moreover, the section describes a procedure of modular replacement checking that can be used after an incomplete model is refined. Finally, Section 2.4 introduces specific deductive rules that can be used to derive information from the model checking procedure and build a deductive proof.

2.1 Modeling the system

Model checking techniques are applied to different modeling formalisms depending on the system the developer is considering and on the properties of interest. In this work we consider *sequential* systems, that are developed using a top-down hierarchical development strategy. For this reason we choose to use *incomplete Büchi automata* ([MSG15]) which are an extension of the well known *Büchi automata* ([Büc90]).

Section 2.1.1 presents an overview of the two formalisms upon which incomplete Büchi automata are based: Finite State Automata and Büchi Automata. Section 2.1.2, instead, specifies the modeling formalisms used to support incomplete systems, i.e., incomplete Büchi automata. Finally, Section 2.1.3 describes the formalisms for their refinements.

2.1.1 Complete models

The formalism chosen to represent a system usually depends on the characteristics of this. Here follows the description of two widely used kind of automata, each of which runs on finite and infinite words, respectively.

Finite State Automata

A *finite-state automaton* (FSA) or *finite-state machine* (FSM) ([Mea55]) is a mathematical model of computation that represents an abstract machine as a set of states. Informally, *states* represent the possible system configurations; some of these can be defined as accepting, and when reached, mark the validation of the input. *Transitions* are the functions that determine the next configuration of the system. In the most common version of FSAs, they are labeled with a set of *propositions* (AP, atomic propositions) that show what condition is needed for the state of the system to change. Given a set of mathematical propositions AP, an FSA is mathematically defined as follows:

Definition 2.1 (Finite State Automaton [Mea55])**.** A *non-deterministic finite state automaton* (FSA) over finite words $\mathcal{M}$ is a tuple $\left\langle \Sigma, Q, \Delta, Q^0, F \right\rangle$, where $\Sigma = 2^{AP}$ is the finite alphabet, Q is the finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states.

If we consider a sequence or a string of characters v belonging to the alphabet Σ, with length $|v|$, we can define:

Definition 2.2 (FSA run [CGP99])**.** A *run* ρ of $\mathcal{M}$ is a mapping $\rho : \{0, 1, \ldots, |v|\} \mapsto Q$ such that:

1. The first state is an initial state, that is, $\rho(0) \in Q^0$;
2. Moving from the ith state $\rho(i)$ to the $i + 1$st state $\rho(i + 1)$ upon reading the ith input letter $v(i)$ is consistent with the transition relation. That is, for $0 \leq i < |v|$, $(\rho(i), v(i), \rho(i + 1)) \in \Delta$.

An *accepting run* of a FSA corresponds to a run in the automaton starting from an initial state $\rho(0)$ until a state $\rho(v) \in F$.

Büchi Automata

Since software systems are often designed to run on infinite inputs, we model executions as infinite sequences of states. The simplest automaton over infinite words is a Büchi automaton (BA), which basically has the same structure of a FSA, but is used to recognize infinite words, formed by a finite prefix, which is followed by a suffix repeated infinitely many times. FSAs are easily translatable to BAs using a stuttering rule, as explained in [PW97, PWW98].

Definition 2.3 (Büchi Automaton [Büc90])**.** A *non-deterministic Büchi automaton* (BA) $\mathcal{M}$ is a FSA $\left\langle \Sigma, Q, \Delta, Q^0, F \right\rangle$ where the set of final states F of the FSA is used to define the acceptance condition for infinite words (also called *ω-words*). Hence, for Büchi automata, F is usually called the set of *accepting states*.

A *run* of a Büchi automaton $\mathcal{M}$ over an infinite word $v \in \Sigma^{\omega}$ (where the superscript ω indicates an infinite number of repetitions) is defined in almost the same way as a run of a finite automaton over a finite word, except that now $|v| = \omega$. Thus, the domain of a run is the set of all natural numbers.

Definition 2.4 (BA accepting run [CGP99]). Let $inf(\rho)$ be the set of states that appear infinitely often in the run ρ. A run over an infinite word is *accepting* if and only if $inf(\rho) \cap F \neq \emptyset$, that is, when some accepting state appears in ρ infinitely often.

The language $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \Sigma^{\omega}$consists of all the $\omega - words$ accepted by $\mathcal{M}$.

2.1.2 Incomplete models

Software development techniques are nowadays incremental and iterative. Modeling incomplete systems has, therefore, become necessary in this context. To support the description of systems that are not completely specified, *incomplete finite state automata* consider the possibility to represent *transparent* states (opposed to *regular* ones), that are replaced in a second development phase with other automata.

Definition 2.5 (Incomplete Finite State Automaton [MSG15]). A non-deterministic incomplete finite state automaton (IFSA) $\mathcal{M}$ is a tuple $\left\langle \Sigma, R, T, Q, \Delta, Q^0, F \right\rangle$, where Σ is the finite alphabet, R is the finite set of *regular* states, T is the finite set of *transparent* states, Q is the finite set of states such that $Q = T \cup R$ and $T \cap R = \emptyset$, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation which allows the definition of transitions that connect states of Q irrespective of their type, $Q^0 \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

As specified in Definition 2.3, software systems are usually designed not to halt during their executions, therefore we need Incomplete Büchi Automata, that extend IFSAs by supporting infinite words. They are an extended version of BAs, designed to support incomplete specifications and their refinements. BAs can be formalized as follows:

Definition 2.6 (Incomplete Büchi Automaton [MSG15]). A non-deterministic incomplete Büchi automaton (IBA) is an FSA $\left\langle \Sigma, Q, \Delta, Q^0, F \right\rangle$, where the set of final states F of the FSA is used to define the acceptance condition for infinite words (ω-words). As in the case of BAs F identifies the set of *accepting states*.

Given an ω-word $v = v_0 v_1 v_2 ...$, a run represents an execution of an IBA.

Definition 2.7 (IBA run [MSG15]). A *run* $\rho : \{0, 1, 2, ...\} \mapsto Q$ over $v \in \Sigma^{\omega}$ is defined for an IBA as follows:

1. $\rho(0) \in Q^0$,

2. $\forall\ i \geq 0,\ (\rho(i), v_i, \rho(i+1)) \in \Delta \vee ((\rho(i) \in T) \wedge (\rho(i) = \rho(i+1)))$.

As it is explained in [MSG15], we can distinguish between three possible types of run:

- A run is *accepting* when some accepting state appears in ρ infinitely often and all states of the run are regular;
- A run is *possibly-accepting* when some accepting state appears in ρ infinitely often and there is at least one state in the run that is transparent;
- A run is *not accepting* in all other cases.

A Büchi automaton $\mathcal{M}$ *accepts* a word v if and only if there exists an accepting run of $\mathcal{M}$ on v. The language $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \Sigma^{\omega}$consists of all the words *accepted* by $\mathcal{M}$.

$\mathcal{M}$ *does not accept* v if and only if it does not contain any accepting or possibly accepting run for v.

Finally, $\mathcal{M}$ *possibly accepts* a word v if and only if it does not accept v and there exists at least a possibly accepting run of $\mathcal{M}$ on v. $\mathcal{L}^{\omega}_p(\mathcal{M}) \subseteq \Sigma^{\omega}$ is the language of all words *possibly-accepted* by $\mathcal{M}$. The language of accepted words can be obtained if we build an automata $\mathcal{M}_c$ from $\mathcal{M}$ by removing all its transparent states and their relative incoming and outgoing transitions.

2.1.3 Refining incomplete models

As previously mentioned, an incomplete model contains a set of transparent states that represent components whose behavior will be later described. The design of a system usually progresses through a set of development steps that concern the *refinement* of transparent states [MSG15]. Each substitution of a transparent state of the initial incomplete model $\mathcal{M}$ with a sub-automaton, that specifies the behavior of the system inside that state, is a *refinement round.* The specific sub-automaton that is substituted is called *replacement.*

From a formal point of view, the concept of refinement relation $\sqsubseteq$, that allows an iterative improvement of the model of the system, is defined as follows:

Definition 2.8 (Refinement [MSG15])**.** Let $\mathfrak{P}_{\mathcal{M}}$ be the set of all possible IBAs. An (I)BA $\mathcal{N}$ is a refinement of an IBA $\mathcal{M}$, i.e., $\mathcal{M} \sqsubseteq \mathcal{N}$[1] if and only if $\Sigma_{\mathcal{M}} \subseteq \Sigma_{\mathcal{N}}$ and there exists some refinement relation $\mathfrak{R} \in Q_{\mathcal{M}} \times Q_{\mathcal{N}}$, such that:

1. $\forall q^0_{\mathcal{M}} \in Q^0_{\mathcal{M}}$ there exists a $q^0_{\mathcal{N}} \in Q^0_{\mathcal{N}}$ such that $(q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \mathfrak{R}$. If $q^0_{\mathcal{M}} \in R_{\mathcal{M}}$, $q^0_{\mathcal{N}}$ must also be *unique*;
2. $\forall q^0_{\mathcal{M}} \in Q^0_{\mathcal{N}}$ there exists exactly one $q^0_{\mathcal{M}} \in Q^0_{\mathcal{M}}$ such that $(q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \mathfrak{R}$;

[1] In the original work [MSG15], the symbol $\preceq$ identifies the refinement relation.

3. $\forall (q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{N}} \in T_{\mathcal{N}}$ then $q_{\mathcal{M}} \in T_{\mathcal{M}}$;

4. $\forall (q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{N}} \in F_{\mathcal{N}}$ then $q_{\mathcal{M}} \in F_{\mathcal{M}}$;

5. $\forall (q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \Re$ and $\forall a \in \Sigma_{\mathcal{N}}$, the following holds:

   a) $(q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \rightarrow (\exists q'_{\mathcal{N}} | ((q_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}} \wedge (q'_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re) \vee ((q_{\mathcal{M}} \in T_{\mathcal{M}}) \wedge \exists q''_{\mathcal{N}} | (q_{\mathcal{M}}, q''_{\mathcal{N}}) \wedge (q''_{\mathcal{N}}, a, q'_{\mathcal{N}})))$;

   b) $(q_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}} \rightarrow ((\exists q'_{\mathcal{N}} | (q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \wedge (q'_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re) \vee (q_{\mathcal{M}} \in T_{\mathcal{M}}))$;

6. $\forall (q_{\mathcal{M}}, q_{\mathcal{N}}), (q'_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re$ such that $(q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}}$ and $(q_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}$, if $q_{\mathcal{M}} \in F_{\mathcal{M}} \cap T_{\mathcal{M}}$ then $q_{\mathcal{N}} \in F_{\mathcal{N}}$.

A refinement relation preserves every behavior of the original model $\mathcal{M}$ in the refined model $\mathcal{N}$ and states that, to every behavior of $\mathcal{N}$, corresponds a behavior of $\mathcal{M}$. In addition, the refinement relation preserves the language containment relation. This means that if a word was satisfied (not satisfied) in $\mathcal{M}$, it remains satisfied (not satisfied) in $\mathcal{N}$. On the contrary, if the word was possibly satisfied in $\mathcal{M}$, can now be satisfied, possibly satisfied or not satisfied, in the refinement $\mathcal{N}$.

While we refer to the new version of the model as *refinement*, we refer to the sub-automaton that substitutes a transparent state $s \in T_{\mathcal{M}}$ as *replacement*, that is formally defined as follows:

Definition 2.9 (Replacement [MSG15])**.** Given an IBA $\mathcal{M} = \langle \Sigma_{\mathcal{M}}, R_{\mathcal{M}}, T_{\mathcal{M}}, Q_{\mathcal{M}}, \Delta_{\mathcal{M}}, Q^0_{\mathcal{M}}, F_{\mathcal{M}} \rangle$, the replacement $\mathcal{R}_t$ of the transparent state $t \in T_{\mathcal{M}}$ is defined as a triple $\langle \mathcal{M}_t, \Delta^{inR}_t, \Delta^{outR}_t \rangle$. $\mathcal{M}_t = \langle \Sigma_t, R_t, T_t, Q_t, \Delta_t, Q^0_t, F_t \rangle$ is the (I)BA that encodes the automaton to be substituted to the state t, and $\Delta^{inR}_t \subseteq \{(s', a, q) | (s', a, t) \in \Delta_{\mathcal{M}}\ and\ q \in Q_t\}$, $\Delta^{outR}_t \subseteq \{(q, a, s') | (t, a, s') \in \Delta_{\mathcal{M}}\ and\ q \in Q_t\}$ are its incoming and outgoing transitions, respectively, which specify how the replacement is connected to the states of $\mathcal{M}$. $\mathcal{R}_t$ must satisfy the following conditions:

1. $t \notin Q^0_{\mathcal{M}} \rightarrow Q^0_t = \emptyset$;

2. $t \notin F_{\mathcal{M}} \rightarrow F_t = \emptyset$;

3. $\forall (s', a, t) \in \Delta_{\mathcal{M}}$, $\exists (s', a, q) \in \Delta^{inR}_t$;

4. $\forall (t, a, s') \in \Delta_{\mathcal{M}}$, $\exists (q, a, s') \in \Delta^{outR}_t$;

Depending of the morphology of the replacement considered, four different types of run can be executed. A *finite internal run* starts from an internal state of the replacement and reaches an outgoing transition of the replacement. An *infinite internal run* starts from an internal state of the replacement and

reaches an accepting state internal to the replacement without ever exiting it. A *finite external run* starts from an incoming transition of the replacement and reaches an outgoing transition of the replacement. Finally, an *infinite external run*, starts from an incoming transition of the replacement and reaches an accepting state inside the replacement without leaving the replacement.

Given these four kinds of runs defined over IBAs, it is also possible to distinguish between the three types described in Section 2.1.2: accepting, possibly accepting and not accepting.

Given a replacement $\mathcal{R}_t$ and a word $v \in \Sigma^*$, we say that:

- $\mathcal{R}_t$ *internally accepts (possibly accepts)* the *finite* word $v \Leftrightarrow \exists$ an internal finite accepting (possibly accepting) run of $\mathcal{R}_t$ on v;
- $\mathcal{R}_t$ *externally accepts (possibly accepts)* the *finite* word $v \Leftrightarrow \exists$ an external finite accepting (possibly accepting) run of $\mathcal{R}_t$ on v;
- $\mathcal{R}_t$ *internally accepts (possibly accepts)* the *infinite* word $v \Leftrightarrow \exists$ an internal infinite accepting (possibly accepting) run of $\mathcal{R}_t$ on v;
- $\mathcal{R}_t$ *externally accepts (possibly accepts)* the *infinite* word $v \Leftrightarrow \exists$ an external infinite accepting (possibly accepting) run of $\mathcal{R}_t$ on v.

A replacement $\mathcal{R}_t$ of a transparent state t of $\mathcal{M}$ can be plugged into the model $\mathcal{M}$ obtaining a refinement $\mathcal{N} = \mathcal{M} \bowtie \mathcal{R}_t$ of $\mathcal{M}$. The alphabet of the refined model $\mathcal{N}$ is the union of the alphabet of the incomplete model with the alphabet of the replaced sub-automaton. The set of regular states of the refined model is the union of the regular states of $\mathcal{R}_t$ and $\mathcal{M}$. The new set of transparent states corresponds to the one of $\mathcal{M}$ except for the transparent state that has just been replaced. The transitions of the refined model include all the transitions of the original model (with the exception of the transitions that reach and leave the transparent state replaced) and all the transitions of the replacement with its incoming and outgoing transitions (that link it to the rest of $\mathcal{M}$). The set of initial states of the refined model include all initial states of $\mathcal{M}$ (except for the transparent state that is substituted, if it was initial), and the ones of its replacement $\mathcal{R}_t$. The set of accepting states of $\mathcal{M} \bowtie \mathcal{R}_t$ include all the accepting states of $\mathcal{M}$ (except for the transparent state that is substituted, if present), and the ones of its replacement $\mathcal{R}_t$.

2.2 Formalizing the specification

Once the developer has proposed a (preliminary) design, he/she may want to check if it possesses certain properties, such as *liveness* or *safety* conditions. Safety properties are usually described as “nothing bad ever happens” and liveness properties as “something desirable will eventually happen”. Numerous combinations deriving from these two classes exist (see [MP89]).

In verification, we check the model against a formal representation of the property of interest. Notice that the specification formalism used to describe a property is strictly connected to the way the system is modeled and the granularity used to model it, as observed in [Wah08]. There are plenty of ways to specify requirements for a system; the formalization techniques are divided into informal, semi-formal and formal techniques. Focusing on the latter, different types of formalisms are available. They all mainly differ for *complexity* and *expressiveness*.

Linear Time Temporal Logic (LTL) is one of the most widely used specification languages for reactive systems [Pnu77]. The reasons of our interest in LTL are, first, the need of consistency with the two work that that inspired ours [PZ01, MSG15], second, its strong relation with Büchi automata that allows to use a model checking algorithm based on automata.

Temporal logic allows specifying behaviors in systems that evolve over time. These formalisms have played a crucial role in applications oriented to verification of programs, protocols and, more generally, abstracted automatic systems. The language of temporal logic defines its predicates over infinite *sequences* of states. In general, each formula is satisfied by a set of formulas and falsified by another set. When interpreted over an execution of a system, a formula expresses a property of that computation.

The most powerful kind of temporal logic is CTL*, from which CTL (Computational Tree Logic), describing properties of a computation tree, and LTL (Linear Temporal Logic), providing a description of events along a computation tree path, are derived.

In the following subsections we introduce syntax and semantics of LTL, and we present the procedure necessary to decorate the states of the claim automaton with the sub-formulae valid in it, according to the method used in [PZ01].

2.2.1 Syntax and semantics of LTL

LTL formulae can be obtained by combining atomic propositions with the boolean connectors $\neg$, $\wedge$, $\vee$ and $\rightarrow$. It includes three temporal unary operators: X ("next", or $\bigcirc$), F ("future", "eventually", or $\diamond$) and G ("globally", "always", or $\square$) and two temporal binary operators: U ("until") and R ("release" or $\mathcal{V}$).

The minimal set X, U, $\wedge$, $\neg$ can be used to derive all other operators.

Semantically, considering a ω-words $v = v_0 v_1 v_2 \ldots$ in Σ^{ω}defined over the alphabet $\Sigma = 2^{AP}$ and being $v^i = v_i v_{i+1} \ldots$ a suffix of v, the satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
v \models true & \\
v \models a & \Leftrightarrow a \in A_0 \\
v \models \phi_1 \wedge \phi_2 & \Leftrightarrow v \models \phi_1 \, and \, v \models \phi_2 \\
v \models \neg\phi & \Leftrightarrow v \not\models \phi \\
v \models X\phi & \Leftrightarrow v_1 \models \phi \\
v \models \phi_1 U \phi_2 & \Leftrightarrow \exists j \geq 0 \mid v_j \models \phi_2 \; and \; \forall 0 \leq i < j, v_i \models \phi_1
\end{array}
$$

2.2.2 LTL to automata

The BA needed by the model checking procedure to describe the specification is obtained with the classical translation algorithm from an LTL formula to an equivalent Büchi automaton ([GPVW95]). Note that this procedure also provides the LTL sub-formulae related to each state. The method is based on the fact that we can build an automaton which accepts all and only the infinite traces represented by an LTL formula.

The pattern used to provide the translation is made by the following steps: 1) *Formula rewriting*, 2) *Core translation*, 3) *Degeneralization*, 4) *Optimization.*

The first step concerns the translation of the LTL formula into its Negation Normal Form, which means that the negation operator $\neg$ is only applied to variables and the only other allowed Boolean operators are conjunction ($\wedge$) and disjunction ($\vee$). Every formula can be brought into this form by replacing implications and equivalences by their definitions, using De Morgan's laws to push negation inwards, and eliminating double negations.

The second step concerns the translation of the LTL formula into the corresponding Generalized Büchi Automaton, defined as follows:

Definition 2.10 (GBA [CGP99])**.** A *Generalized Büchi Automaton* is a tuple $\left\langle \Sigma, Q, \Delta, Q^0, F \right\rangle$ where Σ represents the finite alphabet, Q the finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^0 \subseteq Q$ is the set of initial states and $F \subseteq 2^Q$ the acceptance component.

In a GBA, a run ρ is accepting if for each set $P_i \in F$, $inf(\rho) \cap P_i \neq \emptyset$.

Note that the only thing that changes with respect to the formalism introduced in Definition 2.3 is the acceptance component. In some situations, it can be more convenient to work with automata that have several accepting sets (this is the meaning of F being a subset of the power-set of Q). More intuitively, an infinite word is accepted by a GBA if the execution passes an infinite number of times through at least one state in each element of F.

The third step of the translation refers to the degeneralization algorithm to switch from a GBA to its equivalent BA, expanding the size of the automaton by a factor of $n+1$ ([CGP99]). However, in general, the constructed automaton is small.

The fourth step employs various techniques to reduce the size of the obtained automaton.

2.2.3 Labeled Generalized Büchi Automata

Labeled Generalized Büchi Automata are a variant of GBAs that is worth mentioning because it allows moving labels from transitions to states, keeping the same semantic meaning ([GPVW95]). This convention results particularly useful to express exactly what temporal property holds in each state of the system.

Definition 2.11 (LGBA [PZ01])**.** A *Labeled Generalized Büchi Automaton* is a tuple $\mathcal{B} = \left\langle \Sigma, Q, \Delta, Q^0, \mathcal{F}, L \right\rangle$ where Σ, Q and Q^0 are defined as in its equivalent GBA, whilst this time $\Delta \subseteq Q \times Q$, $\mathcal{F} \subseteq 2^{Q \times Q}$ is the set of *acceptance sets* and $L : Q \to \Sigma$ is a *labeling function* of the states.

A *run* of the automaton $\mathcal{B}$ is an infinite sequence of Q-states $\alpha = q_0, q_{1,\ldots}$ such that $q_0 \in Q_0$, and for every $i \geq 0$, $(q_i, q_{i+1}) \in \Delta$. A run is *accepting* if for every $F \in \mathcal{F}$, $(q_i, q_{i+1}) \in F$ for infinitely many i's. The language accepted by an automaton $\mathcal{B}$, denoted by $\mathcal{L}(\mathcal{B})$, is the set of (labeled) sequences that are accepted by the automaton.

When an LTL formula is converted into a LGBA, a tableau construction is used to assign a sub-formula to each state. The LGBA $\bar{\Phi}$ associates to each state p a formula $\eta(p)$. The formula $\eta(p)$ is obtained through a tableau-like construction. This construction iteratively splits the formula $\eta(p)$ to be considered into two sub-formulae $\left(\bigwedge_{i=1}^{m_p} \nu_i^p\right)$ and $\left(\bigwedge_{j=1}^{n_p} \bigcirc\psi_j^p\right)$, the first of which must hold in the current state, and the other, prefixed by the $\bigcirc$ operator, must hold in the next state [PZ01].

$$\begin{aligned}
\Box A &\leftrightarrow A \wedge \bigcirc\Box A \\
\Diamond A &\leftrightarrow A \vee \bigcirc\Diamond A \\
AUB &\leftrightarrow B \vee (A \wedge \bigcirc(AUB)) \\
ARB &\leftrightarrow (A \wedge B) \vee (B \wedge \bigcirc(ARB))
\end{aligned} \tag{2.1}$$

Essentially, the rules presented in Equation 2.1 are iteratively applied until a fixed point is reached. At this point we know everything that must be valid in the examined state, and it is possible to proceed to the following one, generating a new state that will be labeled with those formulae that appeared prefixed with $\bigcirc$ in its parent node. At the end of the procedure, the formula contains only literals and formulae prefixed by the $\bigcirc$ operator.

The initial tableau, for a set S of formulae, corresponds to a unique node, labeled by S itself. The expansion of a tableau is triggered by applying a rule to one of the leaves nodes (a node without children).

To reduce the number of expansion rules, we will assume the initial set S to be formed by formulae in *negation normal form.* The application of whichever formula, will preserve this property. The expansion rules of LTL tableau are listed in Table 2.1, where symbols S and Λ indicate sets of formulae, A and B are formulae, and *comma* ",” represents the *union* of set theory. The classical

rules just serve as a re-writing in terms of nodes labeled with formulae sets. The temporal rules are written for the operators $\Box$, $\Diamond$, U, and R, and are based on the relative *fixed-point* equation for the same operator, that we have just listed in Equation 2.1. Finally, the rule for $\bigcirc$ is the one that allows to terminate the analysis of a state (the one where all literals in Λ are true, and cannot be further analyzed). The child of the expanded node represents a new state, the following one, where all formulae A such that $\bigcirc A$ is requested in the previous state, are valid. We can say that, whilst the classical rules and the ones for $\Box$, $\Diamond$, U and R are *static*, the rule for $\bigcirc$ is dynamic, concerning a state and all its consecutive. As in propositional logic, when a node contains both an atom and its negation, it must not be expanded, being *contradictory*.

Classical rules	
$\dfrac{A \wedge B, S}{A, B, S}(\wedge)$	$\dfrac{A \vee B, S}{A, S \quad B, S}(\vee)$
Temporal rules	
$\dfrac{\Box A, S}{A, \bigcirc \Box A, S}(\Box)$	$\dfrac{\Diamond A, S}{A, S \quad \bigcirc \Diamond A, S}(\Diamond)$
$\dfrac{AUB, S}{B, S \quad A, \bigcirc(AUB), S}(U)$	$\dfrac{ARB, S}{A, B, S \quad B, \bigcirc(ARB), S}(R)$
$\dfrac{\Lambda, \bigcirc A_1, \ldots, \bigcirc A_n}{A_1, \ldots, A_n}(\bigcirc)$, where Λ is a set of literals	

Table 2.1: Expansion rules of LTL tableau

Termination Branches can terminate when they contain two complementary literals (in this case the node is *closed*). Other branches can however keep expanding in an infinite cycle. Differently from classical propositional tableaux, that always expand towards simpler formulae, here we can keep getting the same formulae infinitely. To avoid cycles of this kind, a *loop checking* procedure stops expanding the formulae when a node with some already existing label is generated.

The procedure to extend a formula can be explained as follows:

Definition 2.12 (Node expansion [Wol85])**.** Let n be a non-contradictory node of a tableau T labeled with the set of formulae S. Let $\dfrac{S}{S_0}(\mathbf{R})$ be a unary expansion rule and $\dfrac{S}{S_0 \quad S_1}(\mathbf{R})$ a binary expansion rule, where $\mathbf{R}$ corresponds

to a an arbitrary logic operator. Then, the tableau T' resulting from T by expanding node n through rule (**R**) can be obtained from T by adding, for $i = 0$ (in case of unary rule) and $i = 0, 1$ (in case of binary rule):

1. a new node n_i as a child of T, if no node labeled with S_i appears in T;

2. an edge from n to m_i, if m_i is a node of T labeled by S_i.

In other words, if the child of some node n should be assigned the label S_i, but there already exists some other node m_i with label S_i, then, the new node is not created, but we add an edge from n to m_i. This way we guarantee that the construction of any tableau terminates.

Being $\eta(p)$ the formula associated to the state p of the automaton $\bar{\Phi}$, that is obtained through the tableau-like procedure just described, we introduce two other results specified in [PZ01]:

Proposition 2.1 ([PZ01])**.** *For every state p of $\bar{\Phi}$, we define $\mu(p) = \neg\eta(p)$, i.e., $\mu(p) = \left(\bigvee_{i=1}^{m_p} \neg\nu_i^p\right) \wedge \left(\bigvee_{j=1}^{n_p} \bigcirc\neg\psi_j^p\right)$.*

Roughly speaking, this means that every state p of the negated claim automaton can be decorated with a formula $\eta(p)$. The negation of each of this formula, $\mu(p)$, represents the actual temporal sub-formula that holds in the non-negated claim.

The following lemma follows immediately from the construction of the automaton $\bar{\Phi}$:

Lemma 2.1 ([PZ01])**.** *If a node p in the constructed LGBA has n immediate successors, $p_1, \ldots, p_n$, then $\eta(p) \rightarrow \left(\bigvee_{i=1}^{n} \bigcirc\eta(p_i)\right)$. Equivalently, $\bigwedge_{i=1}^{n} \bigcirc\mu(p_i) \rightarrow \mu(p)$.*

This basically means that, each formula applied to a state p of the negated claim automaton as a decoration, implies the disjunction of all formulae valid on the successors of p. Because of Proposition 2.1, this is equivalent to say that the conjuntion of the positive formulas associated to the successors of p, imply the formula in p.

Example 2.1. To exemplify the content of this section we here show how to obtain $\eta(p_i)$ and $\mu(p_i)$ for every state i of the requested claim, following the procedure described in Section 2.2.3. We consider the requested claim $\phi = \Box\Diamond p$: "always eventually p". According to the procedure of model checking that will be described in Section 2.3.1, we need to build the automaton equivalent to the negation of the needed property, therefore, $\neg\phi = \Diamond\Box\neg p$.

Figure 2.1 represents the tableau decomposition of the formula.

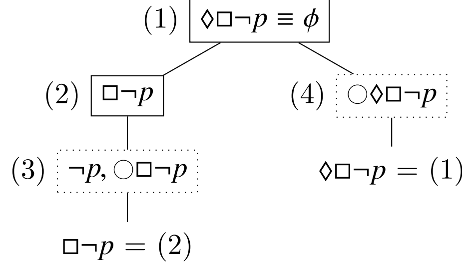


Figure 2.1: LTL tableau for $\Diamond\Box\neg p$

Node **(1)** represents the initial formula to be decomposed. We apply the rule of expansion ($\Diamond$) from Table 2.1 and obtain nodes **(2)** and **(4)**. By applying the rule of expansion ($\Box$) to **(2)** we obtain **(3)**, that is composed by a literal and a $\bigcirc$-formula, so we can apply the *dynamic rule* of Table 2.1, concluding $\Box\neg p$. We observe that such formula already exists in the expansion tree at node **(2)**, therefore we stop expanding and return the formula contained in the last numbered node of the branch: **(3)** $= \neg p, \bigcirc\Box\neg p$. On the right branch we apply the *dynamic rule* to node **(4)** obtaining the same formula as the one in node **(1)**. This means we can return the formula contained in the last numbered node of the branch: **(4)** $= \bigcirc\Diamond\Box\neg p$

The descripted procedure can be used to define a LGBA accepting the infinite words satisfying the formula. The set Q contains the nodes returned by the algorithm. The obtained LTL formulae are assigned to the negated claim nodes as described in [GPVW95]. Here it is even simpler: we observe that the node **(1)** corresponds to the root and has no incoming nodes. Therefore, the formula **(4)** $= \bigcirc\Diamond\Box\neg p$ derived from it corresponds to the initial condition of the automaton, therefore cannot but be on the initial node p_1. As a consequence **(3)** $= \neg p \wedge \bigcirc\Box\neg p$ is assigned to the second node p_2.

According to Proposition 2.1 we can calculate $\mu(p_1)$ as $\neg\eta(p_1) = p \vee \bigcirc\Diamond p$ and $\mu(p_2)$ as $\neg\eta(p_2) = \bigcirc\Box\Diamond p$. All $\mu(p_i)$ are the sub-formulae that used when computing the proof.

2.3 LTL model checking

Model checking can be framed in an *automata-based approach* ([VW86]). In this approach, the temporal formula ϕ is transformed into an equivalent automaton Φ whose language corresponds to the one of ϕ. Verifying if some model $\mathcal{M}$ satisfies the specification ϕ becomes then a question of verifying if the language of $\mathcal{M}$ is included in the language of Φ, which corresponds to $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\Phi)$. By exploiting the properties of BAs that are closed under intersection and complementation, it is derived that the mentioned condition is equivalent to $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\Phi}) = \emptyset$, i.e., the intersection between $\mathcal{M}$ and the

automaton representing the negation of the property $\bar{\Phi}$, is empty. Note that $\bar{\Phi}$ is usually directly obtained by the negation of the LTL formula ϕ, rather than from the complement of the automaton Φ.

In Section 2.3.1, we present the algorithm as it was introduced in [VW86, CVWY92], to check systems formalized as BAs, therefore *completely specified.* In Section 2.3.2, instead, we describe the algorithm modified by [MSG15] to support systems represented as IBAs, therefore *incompletely specified.*

2.3.1 Checking complete models

The classic algorithm can be summarized in three steps:

1) *Creating the automaton of the negated property*: the user might decide to design the model manually directly as a Büchi automaton, or to have it translated from a linear temporal formula with a time complexity that is exponential in the size of the formula, namely $O(2^{|\neg\phi|})$, being $|\neg\phi|$ the size of the negated translated formula.

2) *Building the intersection automaton*: Once the automata representing the model and the negated claim are available, the intersection automaton can be built.

Definition 2.13 (Intersection between two BAs [CGP99])**.** The intersection automaton $\mathcal{I}$ between a BA $\mathcal{M} = \left\langle \Sigma_{\mathcal{M}}, Q_{\mathcal{M}}, \Delta_{\mathcal{M}}, Q^0_{\mathcal{M}}, F_{\mathcal{M}} \right\rangle$ and a BA $\bar{\Phi} = \langle \Sigma_{\bar{\Phi}}, Q_{\bar{\Phi}}, \Delta_{\bar{\Phi}}, Q^0_{\bar{\Phi}}, F_{\bar{\Phi}} \rangle$ is another BA $\left\langle \Sigma_{\mathcal{I}}, Q_{\mathcal{I}}, \Delta_{\mathcal{I}}, Q^0_{\mathcal{I}}, F_{\mathcal{I}} \right\rangle$ defined as follows:

- $\Sigma_{\mathcal{I}}$ is the union of the two alphabets of the intersected automata;
- $Q_{\mathcal{I}} = Q_{\mathcal{M}} \times Q_{\bar{\Phi}} \times \{0, 1, 2\}$. The third component of states is affected by the accepting conditions of the two automata. Its role is to guarantee that accepting states of both $\mathcal{M}$ and $\bar{\Phi}$ occur infinitely often in an execution;
- $\Delta_{\mathcal{I}}$ is the set of transitions $\left(\left\langle q_i, q'_j, x \right\rangle, a, \langle q_m, q'_n, y \rangle\right)$ where $(q_i, a, q_m) \in \mathcal{M}$ and $\left(q'_j, a, q'_n\right) \in \bar{\Phi}$, i.e., the local components agree with the transitions of the two automata. The third component depends on the accepting conditions of both:
  - if $x = 0$ and $q_m \in F_{\mathcal{M}}$, then $y = 1$;
  - if $x = 1$ and $q'_n \in F_{\bar{\Phi}}$, then $y = 2$;
  - if $x = 2$, then $y = 1$;
  - otherwise, $y = x$;
- $Q^0_{\mathcal{I}} = Q^0_{\mathcal{M}} \times Q^0_{\bar{\Phi}} \times \{0\}$;
- $F_{\mathcal{I}} = Q_{\mathcal{M}} \times Q_{\bar{\Phi}} \times \{2\}$.

Notice that $\mathcal{I}$ accepts the intersection language $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\Phi})$. The set of final states does not correspond to the *Cartesian product* of the F sets of the two automata to be intersected because, while accepting states from both can appear individually infinitely often, they may appear together only finitely many times. The constructed automaton accepts its runs exactly when states from $F_{\mathcal{M}}$ and from $F_{\bar{\Phi}}$ happen infinitely many times.
Building the intersection automaton costs like $3 \cdot |Q_{\mathcal{M}}| \cdot |Q_{\bar{\Phi}}|$, where $|Q_{\mathcal{M}}|$ is the number of states of the model and $|Q_{\bar{\Phi}}|$ is the number of states of the automaton that translates the negated property ϕ.

3) Checking the emptiness of the intersection automaton: the model checking result depends on the emptiness of the intersection automaton. If the intersection is empty, the property is satisfied. If it is not, ϕ is violated. Checking the emptiness of a Büchi automaton consists in searching its entire state space for accepting runs, as stated in Definition 2.4. In particular, verifying the non-emptiness of our intersection automaton (i.e., showing that there exists a behavior of the model that does not satisfy the property) is equivalent to finding a strongly connected component (i.e., a maximal set of nodes where each node is reachable from all others), that contains at least an accepting state and that is reachable from any initial state of the model. This corresponds to an accepting run for the automaton. If $\mathcal{L}(\mathcal{I})$ is not empty, then there is a counterexample that can be expressed as a run starting from the initial state of the intersection automaton, and reaching, through a finite prefix, a periodic sequence of states.
The algorithm originally used to find strongly connected components is Tarjan's depth first search [Tar72], with linear complexity in the number of states and transitions $O(|Q_{\mathcal{I}}| + |\Delta_{\mathcal{I}}|)$. A double DFS was explained in [CVWY92] using a more efficient algorithm to solve this problem. The two searches are nested: the first one calls the second one; this one can either terminate the entire algorithm or resume the first search from where it had been interrupted. The basic principle on which this double search is founded is that the first search looks for accepting states. As soon as one is found, the second search starts, looking for a cycle through this state. If the second search fails, the algorithm resumes the first search, that backtracks from the accepting state found. If a cycle is found, the algorithm terminates with *true*, a counterexample exists. The first DFS stack will contain a run from the initial state to an accepting state. The accepted word is an example of a fair execution of the system that does not satisfy the property φ.

2.3.2 Checking incomplete models

The traditional approach just described has been modified in [MSG15] to deal with systems modeled through *incomplete* BAs, to support the modern development process that has become more agile. In this section we describe

the modifications needed to the original model checking algorithm. We remind that the possibility to check incomplete systems offers various benefits. For example, it allows to check a planned design at early stages of the development cycle of software, it lets the developer encapsulate complex parts into abstract modules to examine the design at different abstraction levels, and it gives the opportunity to distinguish between specifications that are already satisfied by the system and those that depend on how the abstract modules will be later developed.

Supporting incompletely specified systems implies the use of a more complicated procedure than the one described in Section 2.3.1. *Incomplete Model Checking* can return three values: *yes*, *no* (like classical model checking) or *possibly-yes* ([MSG15]).

First, we introduce a three value BA semantic for the satisfaction of the formula ϕ in the model $\mathcal{M}$. Let us consider the semantic function $\|\mathcal{M}^{\Phi}\|$ that returns values *true*, *false* or *unknown* depending on whether the model described by the IBA $\mathcal{M}$ satisfies, possibly satisfies or does not satisfy the claim represented by the BA Φ. In the first case, all behaviors of the system satisfy the claim. In the second case, there exists at least a behavior of the system that does not depend on the incomplete parts and that violates the claim. In the third case, finally, whether $\mathcal{M}$ satisfies ϕ or not, depends entirely on the incomplete parts to be completed.

Considering this distinction from the point of view of language containment, we refer to $\mathcal{L}^{\omega}(\mathcal{M}) \in \Sigma^{\omega}$ as the language formed by the words accepted by $\mathcal{M}$, and to $\mathcal{L}_p^{\omega}(\mathcal{M}) \in \Sigma^{\omega}$ as the language formed by the words *possibly*-accepted by $\mathcal{M}$. The following definition links the three possible outputs of model checking to the conditions related to the languages accepted by the model automaton.

Definition 2.14 (Three value BA semantic [MSG15])**.** Given an incomplete BA $\mathcal{M}$ and a BA $\bar{\Phi}$ which specifies the accepted behaviors of $\mathcal{M}$,

$$\|\mathcal{M}^{\Phi}\| = T \quad \Leftrightarrow \quad \mathcal{L}^{\omega}(\mathcal{M}) \cup \mathcal{L}_p^{\omega}(\mathcal{M}) \subseteq \mathcal{L}^{\omega}(\Phi) \tag{2.2}$$

$$\|\mathcal{M}^{\Phi}\| = F \quad \Leftrightarrow \quad \mathcal{L}^{\omega}(\mathcal{M}) \nsubseteq \mathcal{L}(\Phi) \tag{2.3}$$

$$\|\mathcal{M}^{\Phi}\| = \perp \quad \Leftrightarrow \quad \mathcal{L}^{\omega}(\mathcal{M}) \subseteq \mathcal{L}(\Phi) \ \ and \ \ \mathcal{L}_p^{\omega}(\mathcal{M}) \nsubseteq \mathcal{L}^{\omega}(\Phi) \tag{2.4}$$

The first equation means that the model satisfies the claim if and only if all its behaviors (also the ones crossing transparent states) are included in the set of behaviors allowed by the claim. The second means that there exists a behavior of the system that is not included in the ones allowed by the claim. The third one describes the situation where all behaviors of the model are included in the set of behaviors allowed by the claim, but there exist a possible behavior (a run that crosses at least a transparent state) that is not included in the language accepted by the claim.

Given an IBA $\mathcal{M}$ and an LTL formula ϕ, the model checking problem concerns the problem of checking whether the model satisfies, possibly satisfies or does not satisfy the property ϕ, i.e., $\|\mathcal{M}^{\Phi}\|$ is equal to true (T), false (F) or maybe ($\perp$).

[MSG15] revisited the model checking procedure of Section 2.3.1 to take into account models that are incomplete Büchi automata. The procedure works in five subsequent steps:

1. *Creation of the automaton* $\bar{\Phi}$ (this phase is unmodified);
2. *Extraction of the automaton* $\mathcal{M}_c$*,* that contains all the accepting behaviors of the system, and *construction of the intersection automaton* $\mathcal{I} = \mathcal{M}_c \cap \bar{\Phi}$ that contains the behavior of $\mathcal{M}_c$ that violate the property;
3. *Emptiness check of* $\mathcal{I}_c$*.* If $\mathcal{I}_c$ is not empty, the condition $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\bar{\Phi}) \neq \emptyset$ equivalent to Condition 2.3 is matched, which means that the property is not satisfied and every infinite word in the intersection automaton is a counterexample;
4. *Computation of the intersection* $\mathcal{I} = \mathcal{M} \cap \bar{\Phi}$ of the incomplete model $\mathcal{M}$ and the automaton $\bar{\Phi}$ associated with the property ϕ;
5. *Emptiness check of* $\mathcal{I}$*.* Since condition $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\bar{\Phi})$ has already been checked, two cases now arise: if $\mathcal{L}_p(\mathcal{M}) \subseteq \mathcal{L}(\bar{\Phi})$, then the property is satisfied (regardless of the model refinement proposed); if $\mathcal{L}_p(\mathcal{M}) \not\subseteq \mathcal{L}(\bar{\Phi})$, there exist some refinement of $\mathcal{M}$ that violates the property and we are in the case of possible satisfiability.

The definition used to compute the automaton in phase 4, is modified as follows:

Definition 2.15 (Intersection between BA and IBA [MSG15])**.** The intersection automaton $\mathcal{I}$ between an IBA $\mathcal{M}$ and a BA $\bar{\Phi}$ is a BA $\left\langle \Sigma_{\mathcal{I}}, Q_{\mathcal{I}}, \Delta_{\mathcal{I}}, Q^0_{\mathcal{I}}, F_{\mathcal{I}} \right\rangle$ where:

- $Q_{\mathcal{I}} = ((R_{\mathcal{M}} \times R_{\bar{\Phi}}) \cup (T_{\mathcal{M}} \times R_{\bar{\Phi}})) \times \{0, 1, 2\}$ is the set of states. As in the classical intersection algorithm for BAs [CGP99], the labels 0, 1 and 2 indicate that no accepting state is entered, at least one accepting state of $\mathcal{M}$ is entered, and at least one accepting state of $\mathcal{M}$ and one accepting state of $\bar{\Phi}$ are entered, respectively. $\mathcal{M}_{\mathcal{I}} = T_{\mathcal{M}} \times R_{\bar{\Phi}} \times \{0, 1, 2\}$ represents the set of *mixed* states.
- $\Delta_{\mathcal{I}} = \Delta^c_{\mathcal{I}} \cup \Delta^p_{\mathcal{I}}$. $\Delta^c_{\mathcal{I}}$ is the set of transitions $(\left\langle q_i, q'_j, x \right\rangle, a, \langle q_m, q'_n, y \rangle)$ where $(q_i, a, q_m) \in \Delta_{\mathcal{M}}$ and $(q'_j, a, q'_n) \in \Delta_{\bar{\Phi}}$. $\Delta^p_{\mathcal{I}}$ corresponds to the set of transitions where $\mathcal{M}$ is in a transparent state while $\bar{\Phi}$ moves from a state to another.

$\Sigma_{\mathcal{I}}$, $Q^0_{\mathcal{I}}$ and $F_{\mathcal{I}}$ are obtained as in Definition 2.13.

When a property is possibly satisfied, [MSG15] describes a procedure to compute sub-properties that represent the *weakest* condition on the replacements of the transparent states.

2.3.3 Constraints and refinement checking

An incomplete automaton can be iteratively refined by replacing transparent states with automata designed by the developer ([MSG15]).

When planning a new replacement, the developer is assisted and guided through its creation by a constraint which contains a set of sub-models for the unspecified components. The computation of the constraint is triggered when the model checking result is unknown. The constraint is computed to expresses an *upper-bound* on those characteristics that the model refinement must satisfy, in order for the final model to satisfy the claim.

Definition 2.16 (Constraint and sub-properties [MSG15])**.** A constraint C contains a set of sub-properties for the replacements of the transparent states $t_1, t_2, \ldots, t_n \in T_{\mathcal{M}}$, to guarantee that ϕ is satisfied. A *sub-property* is mainly made by a tuple $\left\langle \bar{\mathcal{P}}_t, \Delta_t^{inP}, \Delta_t^{outP} \right\rangle$, where $\bar{\mathcal{P}}_t$ is the automaton that encodes the weakest condition on the replacement of the state t that violates ϕ, and Δ_t^{inP} amd Δ_t^{outP}, called *in-transitions* and *out-transitions* respectively, specify how the model automaton $\bar{\mathcal{P}}_t$ is related to the original model.

[MSG15] associates to each incoming $tr_{in} \in \Delta_t^{inP}$ and outgoing transition $tr_{out} \in \Delta_t^{outP}$ a color label, that specifies how the sub-automaton of the sub-property is connected to other states of the model. *Green* indicates that an incoming transition is reachable from an initial state of the intersection without passing through mixed states; *red* indicates those outgoing transitions from which an accepting state of the intersection automaton is reachable without passing through mixed states; *yellow* indicates both incoming transitions, reached by the initial state passing through mixed states, and outgoing transitions from which it is possible to reach an accepting state passing through mixed states. When an outgoing transition of a sub-property $\bar{\mathcal{S}}_t$ is labeled with *yellow*, the acceptance of a run passing through $\bar{\mathcal{S}}_t$ does not depend on t itself, but on other transparent states replacements.

The goal of the refinement process (see Section 2.1.3) is to find a set of automata, one for each transparent state, such that each of them satisfies the sub-property related to it, and the whole set, therefore, satisfies the entire constraint. This allows the developer to only prove the single sub-properties against the proposed replacements and not the initial property ϕ against the refined model. [MSG15] proposes a procedure to check if the replacement of the transparent state *satisfies*, *possibly-satisfies* or *does not satisfy* the corresponding sub-property.

Definition 2.17 (Refinement Checking [MSG15])**.** Given a particular refinement round, where the developer refines the transparent state t of $\mathcal{M}$ through the replacement $\mathcal{R}_t$, the refinement checking problem is to compute whether the refined automaton $\mathcal{N}$ obtained by plugging the replacement $\mathcal{R}_t$ of the transparent state t into the model $\mathcal{M}$, satisfies, does not satisfy or possibly satisfies the property ϕ.

The refinement checking process includes the need to compute an intersection automaton between each replacement $\mathcal{R}_t$ and the sub-property $\bar{\mathcal{S}}_t$ related to the same transparent state t. The intersection structure is defined as follows:

Definition 2.18 (Extended intersection between replacement and sub-property [MSG15])**.** The intersection $\mathcal{I}_t$ between a replacement $\mathcal{R}_t = \left\langle \mathcal{M}_t, \Delta_t^{inR}, \Delta_t^{outR} \right\rangle$ and the corresponding sub-property $\bar{\mathcal{S}}_t = \left\langle \bar{\mathcal{P}}_t, \Delta_t^{inP}, \Delta_t^{outP} \right\rangle$ is an automaton which is obtained by the intersection of the automata associated with $\mathcal{R}_t$ and $\bar{\mathcal{S}}_t$ ($\mathcal{M}_t \cap \bar{\mathcal{P}}_t$), and a set of incoming and outgoing transitions that corresponds to the parallel execution of the transitions of $\mathcal{R}_t$ and $\bar{\mathcal{S}}_t$.

Given the constraint C computed for the system $\mathcal{M}$, for each transparent state t, we distinguish three cases: the replacement $\mathcal{R}_t$ *does not satisfy* the constraint if its extended intersection with the sub-property $\bar{\mathcal{S}}_t$ allows to reach a *red* outgoing transition or an internal accepting state of $\bar{\mathcal{S}}_t$ from a *green* incoming transition or from an internal state of $\bar{\mathcal{S}}_t$; the replacement $\mathcal{R}_t$ *satisfies* the constraint C if the extended intersection automaton does not allow to reach any internal accepting state of $\bar{\mathcal{S}}_t$ or any of its *yellow* outgoing transitions. [MSG15] demonstrates that checking a replacement $\mathcal{R}_t$ versus its constraint C corresponds to checking the refined automaton $\mathcal{N}$ against its property ϕ.

2.4 Proof of $\mathcal{M}$-validity of property ϕ

In this section, we briefly introduce the concept of deductive system and the utility of its rules to build proofs. Then, we describe the approach of Peled et al. ([PZ01]) that uses *ad hoc* rules to infer knowledge from an intersection automaton built in the model checking procedure. A final example is illustrated to explain the rules and results presented in [PZ01].

A proof has the purpose of rigorously explain the reason why a statement holds. A way of proving the validity of a property over a model is, for example, by using a deductive system. Deductive systems use inference rules to generate new knowledge starting from known axioms.

As specified in [MP91], a *deductive system* consists of the following elements:

- a set of *axioms*. This is a set of valid formulas that are taken as basic properties of the operators in the language;

- a set of *rules*. Rules provide patterns by which new valid formulas can be derived from other formulas whose validity has been previously established. A rule has the general form $\frac{p_1,\ldots,p_k}{q}$. It consists of a list of formulas $p_1, \ldots, p_k$, called *premises*, and a formula q, called *conclusion* of the rule. Such a rule states that if we have already established the validity of $p_1, \ldots, p_k$, then we may infer the validity of q.

[PZ01] considers an intersection automaton that represents the combined state spaces of the automaton $\mathcal{M}$ (representing the model of a system) and the automaton $\bar{\Phi}$ (representing the negation of a requirement for that system). According to their procedure, this automaton can be used to produce a temporal proof of the fact that $\mathcal{I} = \mathcal{M} \cap \bar{\Phi}$ is empty, i.e., it does not contain any accepting path.

Notice that the automata used in this section are LGBAs (see Definition 2.11), as in [PZ01]. The rules here described are modified in the contribution of this thesis (Chapter 3) to be used with automata with incompletely specified models.

The intersection automaton used in the procedure corresponds to the classical intersection computed between automata, enriched by *failed* nodes.

Definition 2.19 (Failed state [PZ01])**.** A *failed node* (or *failed state*), is a node (q, p), where $L(q) \notin L(p)$, i.e., the label on the model state q does not satisfy the propositional assignment on the claim state p, which is the LTL formula $\eta(p)$ (see Proposition 2.1).

More intuitively, "failed" is referred to the fact that the state represents a system configuration where the negated property is not satisfied. From a practical point of view, failed nodes are the ones without successors, the leaves of the intersection automaton. On the contrary, a *successful node* is a node where the negated property could be satisfied, i.e., a node (q, p) where the propositional assignment of q satisfies the one in p.

The idea introduced by [PZ01] is to analyze the intersection automaton by visiting the automaton from the failed nodes to the initial nodes, searching for a counterexample that, in case the model $\mathcal{M}$ does not satisfy the property ϕ, is not found.

To build a sound and complete proof system, [PZ01] suggests to consider four kinds of correctness assertions that we outline here. q represents the state coming from the model automaton $\mathcal{M}$ and p represents the state coming from the $\bar{\Phi}$ automaton.

Failure axiom FAIL. Let (q, p) be a *failed* node. Then, we can conclude that

$$q \models \mu(p) \tag{2.5}$$

The justification for this axiom is simple: the node has failed because we have checked the label of the incoming transition state q against the propositional claim leading to p, and the propositional claim has failed to hold. Thus, $q \models \neg prop(p)$. But, note that $\neg prop(p) \to \mu(p)$. Therefore, $q \models \mu(p)$.

Successors rule SUCC. Let (q, p) be a *successful* node, the model state q has m successors $q_1, \ldots, q_m$ and the claim state p has n successors $p_1, \ldots, p_n$. Then, we have

$$\frac{\begin{array}{c} q \to \{q_0, \ldots, q_m\} \\ \text{For each } 1 \leq i \leq m,\ q_i \models \bigwedge_{j=1,n} \mu(p_j) \end{array}}{q \models \mu(p)} \tag{2.6}$$

The validity of this proof rule derives from the correctness of the construction. In particular, Lemma 2.1. The interested reader may find additional information in [BCG95]. Note that the failure axiom can be seen as a trivial case of the successors rule, with no premises.

This rule is necessary to propagate formulas valid on already failed portions of the automaton to states that directly lead to these, with only one transition. It is a one step induction.

Induction rule IND. Let C be a strongly connected component in the considered automaton. Let $Exit(C)$ be the set of nodes not in C, with an incoming transition from a node in C. We consider the case where the SCC C does not satisfy at least one acceptance condition (let us remind that in a Labeled Generalized Büchi Automaton, an accepting run is a run that passes an infinite number of times in at least one state of each element of F, the accepting set).

$$\frac{\begin{array}{c} \text{For each } (q, p) \in Exit(C),\ q \models \mu(p) \\ \text{For each } (q, p) \in C,\ q \to successor(q) \end{array}}{\text{For each } (q, p) \in C,\ q \models \mu(p)} \tag{2.7}$$

Notice that defining a set that contains the nodes reached by states inside the SCC but not beloging to it, is similar to identifying the successors of a trivial SCC. The Induction rule is a generalization of the Successors rule, that allows to consider the SCC as a single macro-node of the graph.

Conjunction rule CONJ. This rule allows conjoining any pair of conclusions made on a given state along with making temporal logic interferences.

$$\frac{q_{init} \models \mu(p_1), \ldots, q_{init} \models \mu(p_n), (\mu(p_1) \wedge \ldots \wedge \mu(p_n)) \to \phi}{q_{init} \models \phi} \tag{2.8}$$

The last premise, $(\mu(p_1) \wedge \ldots \wedge \mu(p_n)) \rightarrow \phi$, stating that the conjunction of $\mu(p_i)$ implies the required property, assumes a given sound and complete propositional temporal logic.

Now, let us assume we are given a deductive system $\mathcal{H}$ consisting of the FAIL axiom and SUCC, IND and CONJ rules just described. The goal is to build a derivation, called *proof*, that establishes the validity of a formula stating that a model $\mathcal{M}$ satisfies the property ϕ, using the axiom and rules.

The axiom and the rules are only schemas. They therefore need to be instantiated, i.e., their symbols are substituted with the real states and formulae belonging to the automaton to which they are applied. For example, the axiom $q \models \mu(p)$ only acquires a meaning when we consider the real model state q_2 as *instantiation* of the variable q and the real claim state p_1 as instantiation of the variable p.

A proof in the system $\mathcal{H}$ represents a sequence of rows, each of which contains a formula p (a temporal formula, in our procedure). The row states the validity of p, supported either from an axiom or the application of another rule.

Given a proof consisting of the lines $\varphi_1, \ldots, \varphi_n$, we say that this is a *proof of* φ_n, the last formula of the proof. We say that φ_n is a *theorem* of the logic. From now on, we may use φ_n, or an instantiated version of it, in subsequent proofs as though it were an axiom.

In particular, the desired output of the proof built in [PZ01] is a final statement that represents the satisfaction of the property ϕ by the model of the analyzed system $\mathcal{M}$.

Proposition 2.2 ([PZ01]). *If a property ϕ is true in all executions of our system, then we say ϕ is a* valid property *of a model $\mathcal{M}$, i.e., $\mathcal{M} \models \phi$, where the symbol $\models$ denotes the satisfaction relation for linear temporal properties.*

[PZ01] shows that, given an empty intersection automaton, demonstrating that the property ϕ is satisfied by every execution of the automaton starting from its initial states (q_0, p_0), corresponds to say that the whole model satisfies it. Moreover, according to Proposition 2.1, the claim ϕ can be seen as a conjunction of LTL formulae.

Theorem 2.1 ([PZ01]). *Assume $\mathcal{L}(\mathcal{M} \cap \bar{\Phi}) = \emptyset$. Then, for every initial state (q_0, p_0) of $\mathcal{I} = \mathcal{M} \cap \bar{\Phi}$, $(\mathcal{M}, q_0) \models \phi$.* Thus, $\mathcal{M} \models \bigwedge_{(q_0,p_0) \in Q_0^{\mathcal{I}}} \mu(p_0)$.

To better explain the applicability of each rule, here follows a basic example.

Example. We consider the intersection automaton graph in Figure 2.2. The depicted graph represents an intersection automaton deriving from a simple switch system model $\mathcal{M}$ with the automaton $\bar{\Phi}$ associated with the claim $\phi = \Box\Diamond p$. $\mathcal{M}$ presents two states q_1 (labeled with p) and q_2 (labeled with t), being q_1 initial, with transitions $q_1 \rightarrow q_2$ and $q_2 \rightarrow q_1$.

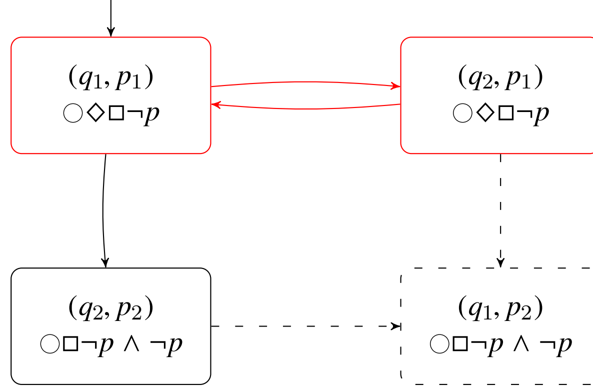


Figure 2.2: An example intersection automaton

The intersection automaton $\mathcal{I}$ presents four nodes, three of which are regular nodes (continuous border) and one is failed (dashed border). The nodes are labeled with their model associated state (q_1 or q_2) and their claim associated state (p_1 or p_2). In addition, the label contains a LTL formula that decorates the state, representing $\eta(p_1)$ or $\eta(p_2)$ according to the claim associated state of the considered node.

RULE FAIL is applicable to node (q_1, p_2) in the intersection of Figure 2.2. This node is indeed failed because the propositional assignment of the model state $L(q_1) = p$ does not comply with the propositional assignment of the negated property state $L(p_2) = \bigcirc\Box\neg p \wedge \neg p$, being p and $\neg p$ obviously a contradiction. We can, therefore, state:

$$\frac{-}{q_1 \models \mu(p_2)} \quad = p \vee \bigcirc\Diamond p$$

RULE SUCC is applicable to node (q_2, p_2) that is a single node without a self loop but with successors (trivial SCC). Considering that the model state q_2 has only one successor q_1 and that the claim state p_2 only has itself as a successor, the rule works as follows:

$$\begin{array}{ll} q_2 \rightarrow \{q_1\} & \textit{successors} \\ q_1 \models \mu(p_2) & = p \vee \bigcirc\Diamond p \\ \hline q_2 \models \mu(p_2) & = p \vee \bigcirc\Diamond p \end{array}$$

RULE IND is applicable to the strongly connected component $C = \{(q_1, p_1), (q_2, p_1)\}$. According to the rule, we need to identify the set $Exit(C) = \{(q_2, p_2), (q_1, p_2)\}$ corresponding to those nodes that are directly reached from one node of C but are not part of C itself. Applying the rule, we obtain:

$$\begin{array}{ll} q_1 \models \mu(p_2) & = p \vee \bigcirc\Diamond p \\ q_2 \models \mu(p_2) & = p \vee \bigcirc\Diamond p \\ q_1 \to \{q_2\} & successors \\ q_2 \to \{q_1\} & successors \\ \hline q_1 \models \mu(p_1) & = \bigcirc\Box\Diamond p \\ q_2 \models \mu(p_1) & = \bigcirc\Box\Diamond p \end{array}$$

This rule allows to propagate knowledge we possess on previously discarded nodes to more complex components on the graph that are located "above" them (meaning further from failed nodes and closer to initial ones).

<u>Rule Conj</u> is the rule that allows us to conclude the proof and draw together all partial conclusions made until now.

$$\begin{array}{ll} q_1 \models \mu(p_2) & = p \vee \bigcirc\Diamond p \\ q_1 \models \mu(p_1) & = \bigcirc\Box\Diamond p \\ (\bigcirc\Box\Diamond p \wedge (p \vee \bigcirc\Diamond p)) \to \Box\Diamond p & \text{conjunction of conclusions} \\ \hline q_1 \models \phi & = \Box\Diamond p \end{array}$$

Referring to the example just described, we would like the reader to note that, if the rules are written in this order, the conclusions of each rule are used as premises to the following rules. The chain is easily solved:

$q_1 \models \mu(p_2)$ from FAIL is applied to SUCC that outputs $q_2 \models \mu(p_2)$. Both conclusions of FAIL and SUCC are used in IND, that derives that $q_1 \models \mu(p_1)$ and $q_2 \models \mu(p_1)$. Finally, the validities gathered on the initial state q_1 can feed the last rule, stating $q_1 \models \phi$, which corresponds to saying that the whole model satisfies the property, thus the property is a theorem of the model.

$$q_1 \models \phi \quad \to \quad \mathcal{M} \models \phi$$

3 Contribution

In this chapter we present the technique proposed to build a proof starting from a model checking procedure that supports incomplete systems. Our procedure is activated in case the model checking algorithm verifies that the analyzed system satisfies (or simply "might satisfy") the submit requirement. These are the cases where the model checker outputs "yes" or "possibly-yes", since the search for a counterexample has failed.

The techniques and the formalisms on which our approach is based, have been extensively explained in Chapter 2. Section 3.1 outlines the procedure proposed in this thesis to compute incremental proofs; Section 3.2 describes the construction of an initial proof that justifies *why* the initially provided incomplete description of the system satisfies the requested property. This first step is final in case the model is already completely defined, or if the user is satisfied with a partial explanation of its checking process. Differently, whenever the initial description of the system was incomplete, with modules left unspecified, the developer might later request to replace these parts with lower level descriptions. Our procedure uses the constraint computation method of [MSG15] to guide the developer through the design of new components. If the proposed refined model does not fail the given condition, it is possible to compute single sub-proofs dedicated to the only replaced states and carry on with the completion of the master proof. Section 3.3 shows the construction of sub-proofs that justify in which way the replacement of a single module satisfies or possibly-satisfies the constraint. Finally, Section 3.4 explains how to link the initial incomplete proof with the related sub-proofs of the newly specified components and understand the dependency relation among these.

3.1 High level outline

The final goal is to build incremental deductive proofs of Linear Time Temporal Logic (LTL) properties for incomplete systems, that are completed progressively when the system gets refined. A deductive system is assembled by analyzing the intersection automaton between the model of the system and the negation of the required property, through the study of its strongly connected components, that supplies rules and validities for the formal system. Whenever the system gets partially or completely refined by replacing a transparent state with a more complex specification, an *ad hoc* sub-proof is computed for the replacement and then is plugged into the initial proof. The

final and complete framework that we obtain works as described in Figure 3.1.

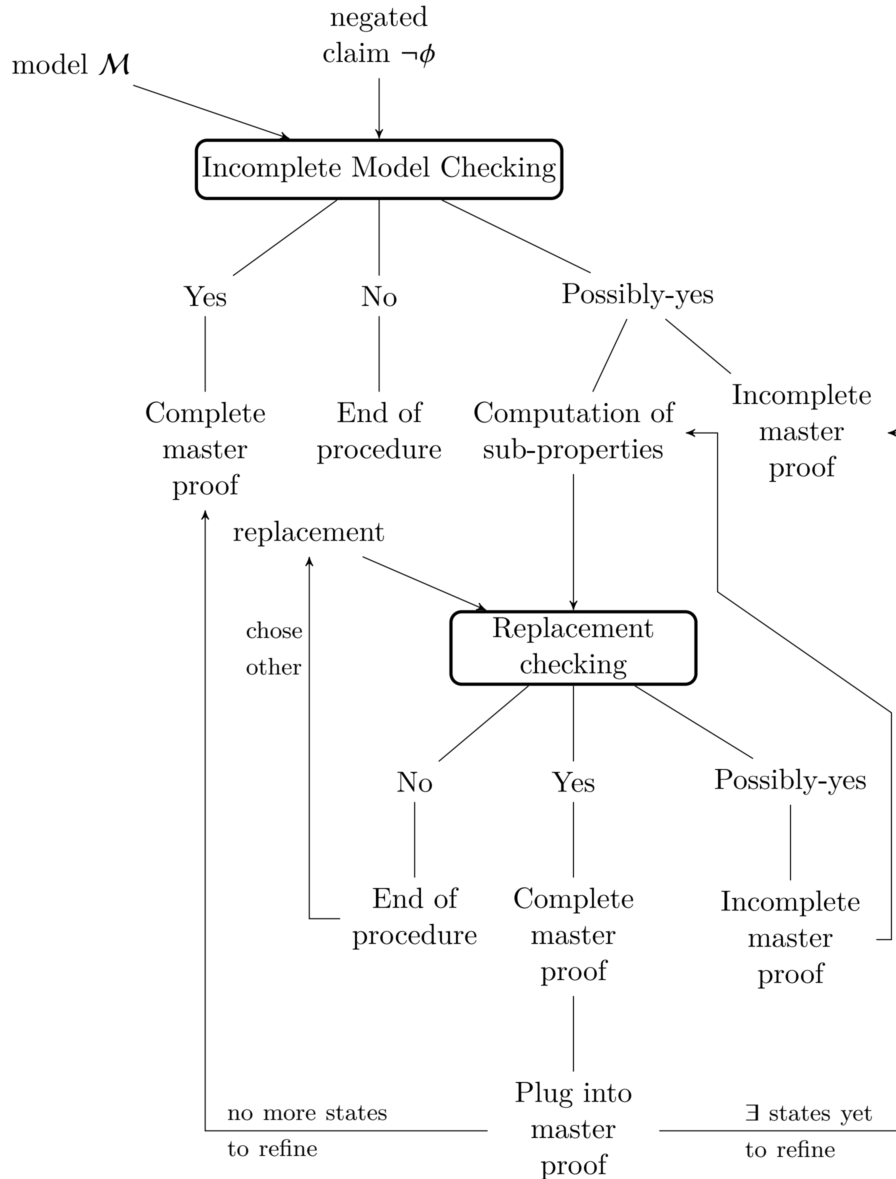


Figure 3.1: Integration of the proof computation in an incomplete model checking framework

We start from the classic procedure of LTL automata-based model checking to deal with incomplete model specifications [MSG15]. As described in [MSG15], our input is the model of the system (represented as an IBA, a possibly incomplete Büchi automaton) $\mathcal{M}$ and the formalization of the nega-

tion of the property that we would like to check, $\neg\phi$ (expressed as BA). For consistency with [MSG15, CGP99], we label the transitions of the model with subsets of propositions belonging to the alphabet of the automaton. Note that, when the claim (property of interest) is considered, the transitions are labeled with boolean expressions, therefore including negative propositions. This is because, being the model a description of something we observe or are willing to create, we must know what holds and where. If a proposition is not mentioned on a transition, this means that is not true. Instead, the fact that the claim includes propositional formulae allows the representation of a set of transition, i.e., all those that satisfy that propositional formula. The two conventions are equivalent from an expressivity point of view.

Within the procedure of *Incomplete Model Checking* presented in Figure 3.1, the automata-based model checking framework builds the intersection automaton $\mathcal{I} = \mathcal{M} \cap \neg\phi$, as described in Definition 2.13. Three possible cases arise: the intersection is not empty and the existing accepting run only depends on regular states (**No**, the property is *not satisfied* by the model), the intersection is empty (**Yes**, the property is *satisfied* by the model), or the emptiness of the intersection depends on further refinements of incomplete states, i.e., there exists a possible accepting run (**Possibly-yes**, the property is *possibly satisfied* by the model). In this last case, there is no accepting run only crossing regular states, but there is at least one path through mixed states that is possibly accepting.

In the first case, no proof is built. In the second one, a complete deductive *master proof* is built and closed at the first step. Incomplete models where the satisfiability of the property does not depend on the refinement of the transparent states also fall into this case. Finally, in case of possible satisfiability, we proceed with an incremental construction of the deductive proof. A first master proof is set up, as a starting grid for later sub-proofs to be plugged in. In parallel with the computation of this incomplete proof, according to [MSG15], a constraint is computed, corresponding to a set of sub-properties (one for each transparent state) that guides further developments of states yet to be specified.

Once the constraint is available, the developer is encouraged to come up with a refinement of the model initially provided. In particular, he/she can decide to substitute the transparent states predisposed during the initial, more high level phase of the system description, with more detailed sub-automata. Our framework allows to check these new components (called *replacements*, expressed as (I)BAs) individually against the corresponding sub-property (expressed as a BA plus incoming and outgoing transitions) and compute a sub-proof only for this part, using the intersection automaton calculated with the sub-property and the replacement referred to the same transparent state. The incremental step, *Replacement Checking*, allows to proceed without checking and proving the whole system from scratch. The three cases just described

are obtained again: if at least one *non-empty intersection* (**No**) case occurs, the property is not satisfied and the proof is nullified: the developer is encouraged to try another replacement. If only *empty intersection* (**Yes**) cases occur, the sub-proof can be completed and closed: its conclusion can now be used to solve the possible dependencies created in the master proof related to that transparent state. Finally, any *possibly empty intersection* (**Possibly-yes**) case leads to a partial completion of the deductive proof and the need to further refine the system. In this case no dependency can be resolved, but other replacements are expected.

Our proposed approach is considered in the context of a bigger framework: we are adding deductive proofs to the modular model checking approach of [MSG15] and at the same time are including the chance to proceed in an incremental way in [PZ01].

The next sections of the chapter describe the base step, the master proof construction (Section 3.2), the inductive step corresponding to the construction of sub-proofs for every transparent state of the model (Section 3.3), and finally a procedure to join the obtained results (Section 3.4).

3.2 Computing the master proof

This section provides the description of the first step of the contribution of this thesis: the construction of the master proof. Both when the model checker returns *true* and when it outputs *possibly-true*, it is possible to build a deductive proof of the fact that the input model *satisfies* or *possibly satisfies* the required property. We refer to the initial proof as *master proof* because it is in bijective correspondence with the first provided version of model (*master*) and it is also the grid referenced by further results.

Algorithm 3.1 shows the point where we insert our contribution inside the model checking framework. If the model checker returns 0 (*no*), it means that a counterexample has been found and the procedure ends directly (Lines 3- 4). Line 6 is executed if the model checking procedure returns values 1 (*yes*) or -1 (*possibly-yes*). The procedure BuildProof is called and returns a complete proof, in case the model checker verifies that the property is satisfied by the model, or an incomplete proof, in case of *possible*-satisfaction of the claim.

Algorithm 3.1 Model checking with deductive proof

```
1: procedure CheckingWithProof(𝓜, Φ̄)
2:     𝓘 ← 𝓜 ∩ Φ̄;
3:     if ModelChecking(𝓘) = 0 then
4:         return false;
5:     else
6:         return BuildProof(𝓘(𝓜 ∩ Φ̄));
```

The complete procedure to compute the master proof is presented in Algorithm 3.2 (BuildProof). The algorithm includes six different steps:

- *Extend the intersection automaton* (Line 2): the intersection automaton described in [MSG15] is enriched by considering also *failed* states (Definition 2.19), configurations of the model that fail to satisfy the negated property. These are needed as a base case for the deductive proof. This function is extensively explained in Section 3.2.1.

- *Creation and update of a dependency-graph* (initialized at Line 3): we create a data structure that maps each validity statement (*key*) derived from the intersection to a set of other validities it depends on (*values*). These validities need first to be resolved, i.e., a complete proof that states that they are valid must exist in order for the key validity to be completely proven too. Section 3.2.5 describes how the dependency graph is used and updated at each new rule creation.

- *Strongly connected components (SCCs) identification and sorting* (Lines 4- 5): we use a classical Tarjan's algorithm [Tar72] that takes a directed graph (the extended version of the intersection automaton) as input and produces a partition of its nodes into SCCs. This unsorted set is later sorted through a partial order that guarantees that, whenever a component is used to generate a rule of the proof, all the other components reachable from it, have already been analyzed. This procedure is described in Section 3.2.2.

- *Dangerous strongly connected components rejection* (structure initialized at Line 6): we build a data structure that contains information about the graph components that need to be discarded. These parts contain accepting cycles, and, therefore, would never appear in the intersection automaton if the model was complete and the model checker had returned "*yes*". These components are indeed still reachable because the used model is incomplete. Once the model is completely refined, they will never be accessible again. This explains why we need to exclude them from the proof (and also delete all dependencies of other components on them). Section 3.2.3 clarifies this process.

- *Rules building* (Lines 7- 9): according to each SCCs characteristics, a different rule among RuleFail, RuleSucc, and RuleInd is chosen, as thoroughly described in Section 3.2.3.

- *Conjunction of rules* (Line 10): a conjunction rule is necessary to connect all conclusions drawn on the SCCs of the graph. Section 3.2.4 describes this last phase.

Algorithm 3.2 Deductive proof construction

```
1: procedure BuildProof(𝓘)
2:     𝓘_ext ←ExtendIntersection(𝓘);                         ▷ Algorithm 3.3
3:     Proof.depGraph ← (Q_𝓘ext, {});
4:     Proof.SCC ←Tarjan(𝓘_ext);
5:     Proof.SCC ←Sort(Proof.SCC);                           ▷ Algorithm 3.4
6:     Proof.rejects ← {};
7:     Proof.rules ← {};
8:     for scc ∈ SCC do
9:         Proof.rules ← Proof.rules ∪ BuildRule(scc);       ▷ Algorithm 3.5
10:    Proof.rules ← Proof.rules ∪ RuleConj(Φ̄, 𝓘_ext);       ▷ Algorithm 3.9
11:    return Proof;
```

The construction of the extended intersection automaton, the SCCs sorting, the rules building and conjunction are phases already introduced in [PZ01] that have been modified to cope with incompleteness and a different formalism (BAs instead of labeled generalized BAs). The *dependency graph* and *rejects* structures are, instead, a novelty of our approach, introduced to supply the proof entity with more flexibility. The first one provides the ability to keep trace of which statements are final and which ones are only guaranteed under certain assumptions. The second one provides the ability to exclude from the reasoning the components that would not appear in a closed proof (because leading to a counterexample).

Example. We introduce an example that we are going to use through this chapter to support the description of the steps of the proposed procedure. The system under analysis is a railway crossing system. It may assume a number of different configurations, some of which fully defined (modeled by the states q_1, q_3, q_5, and q_6), others transparent (modeled by the states q_2 and q_4), to be later substituted with more detailed automata.

The presented model concentrates on the critical region of the described crossing. We establish that a train is considered to be out of the critical region (intersection of the rails with the street) when it has left the crossing. We model this situation with the propositional atom out. Its opposite, $\neg out$, describes the situation in which at least one train car is still passing through the crossing. As far as the state of the bar is concerned, we express through low the situation where the bar blocks the passage of vehicles on the street, clearing the way of the train. $\neg low$ means that the bar has been raised.

We remind that, according to the convention used for transitions labeling of the model, whenever a propositional statement does not appear on a transition, it means that its negation holds. Between the first four states, the fundamental requirement to step from a configuration to the other is that the security bar is in its lowered state. Therefore, on the first three transitions

$low \wedge \neg out$ holds. The condition to exit state q_4 is out . The cycle between states q_5 and q_6 corresponds to a situation where the bar is lowered again, even if the train is not in the critical area.

We request a basic safety property *“The bar is lowered at least until the train gets out of the critical area”*, formalized as $low\, U\, out$. The model of this simple example is presented in Figure 3.2 and the automaton corresponding to the negation of the LTL claim $\neg\phi = \neg(low\, U\, out) = (\neg low\, R\, \neg out)$ has been translated to its equivalent Büchi automaton according to [GO01] and is represented in Figure 3.3.

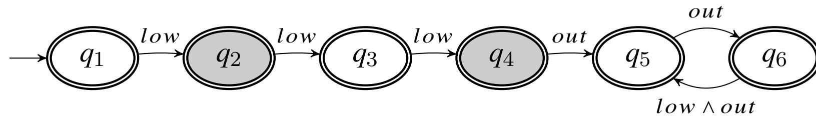


Figure 3.2: Model of railway crossing system

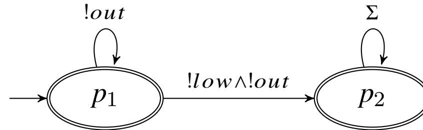


Figure 3.3: Negation of the property for the railway crossing system

As described in Section 2.2.2, we can derive the linear temporal logic formulae valid on the property automaton states following an LTL tableaux procedure of decomposition of the initial LTL formula corresponding to the negated claim $\neg\phi$. This information is used to decorate the information of the proof.

Specifically for this example, we build the tableau in Figure 3.4 similarly to how done in the example of Section 2.2.2.

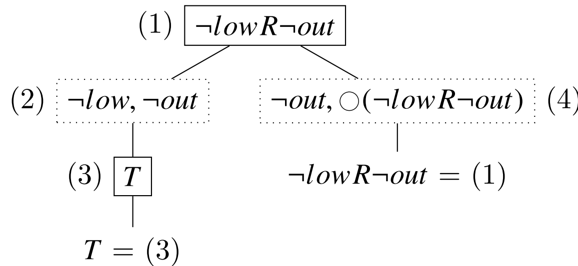


Figure 3.4: LTL tableau for $\neg low R \neg out$

From the two nodes squared with dotted lines, we deduce **(4)** $= \eta(p_1) = \bigcirc(!lowR!out) \wedge !out$ and **(2)** $= \eta(p_2) = !low \wedge !out$. Later, we derive the positive formulae (according to Proposition 2.1: $\mu(p) = \neg\eta(p)$), and we obtain $\mu(p_1) = \bigcirc(lowU out) \vee out$ and $\mu(p_2) = low \vee out$. For now, though, the proof is written using $\mu(p_1)$ and $\mu(p_2)$ to denote the sub-formulae valid on states p_1 and p_2. For clarity purposes, the LTL formulae will be mapped onto the proof only at the end.

3.2.1 Extending the intersection

The intersection automaton needs to undergo two modifications. First, we unify into a single all the states that correspond to the same tuple $< q, p >$ where q is a state of the model and p is a state of the claim and have a different number $\in \{0, 1, 2\}$ as third component (see Definition 2.13). Second, we decorate the intersection by adding *failed* states, that will be crucial later, in order to let the initial rules of the proof fire. The complete procedure is outlined in Algorithm 3.3. The algorithm is applied to the intersection automaton $\mathcal{I} = \left\langle \Sigma_{\mathcal{I}}, Q_{\mathcal{I}}, \Delta_{\mathcal{I}}, Q^0_{\mathcal{I}}, F_{\mathcal{I}} \right\rangle$ built from the IBA representing the model $\mathcal{M} = \left\langle \Sigma_{\mathcal{M}}, Q_{\mathcal{M}}, \Delta_{\mathcal{M}}, Q^0_{\mathcal{M}}, F_{\mathcal{M}} \right\rangle$ and the BA representing the claim $\bar{\Phi} = \left\langle \Sigma_{\bar{\Phi}}, Q_{\bar{\Phi}}, \Delta_{\bar{\Phi}}, Q^0_{\bar{\Phi}}, F_{\bar{\Phi}} \right\rangle$. We remind that Q represents the set of states of the referred automaton, Q^0 is the set of initial states of the automaton, and Δ represents the set of transitions between two states of the automaton.

Algorithm 3.3 Extension of intersection automaton

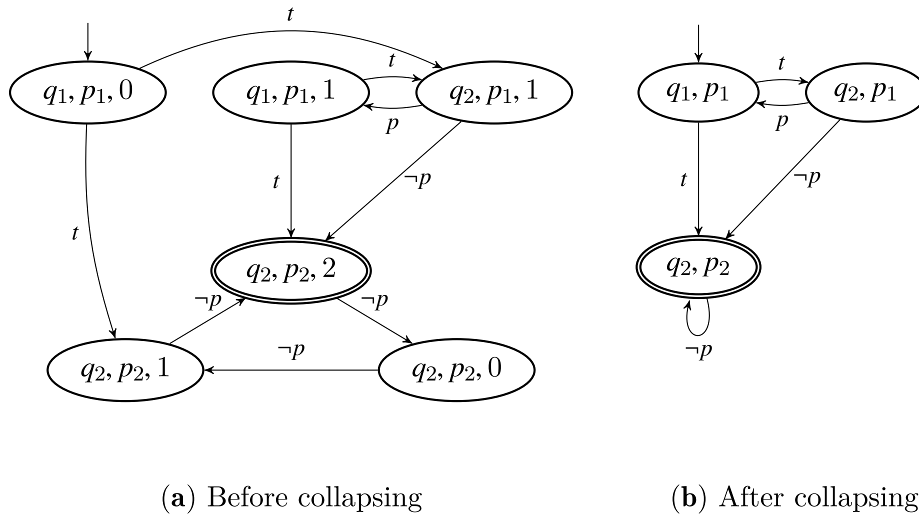
```
1: procedure ExtendIntersection(I(M ∩ Φ̄))
2:     I_ext ← CollapseNodes(I);
3:     Q_CP ←CartesianProduct(Q_M,Q_Φ̄);
4:     for (q′, p′) ∈ Q_CP do
5:         reachable ← false;
6:         if (q′, p′) ∉ Q_I ∧ q′ ∉ Q^0_M then
7:             for (q, p) ∈ Predecessors(q′, p′) do
8:                 if (q, p) ∈ Q_I then
9:                     Δ_I_ext ← Δ_I_ext ∪ {(q, p), a, (q′, p′)};
10:                    reachable ← true;
11:            if reachable then
12:                Q_I_ext ← Q_I_ext ∪ (q′, p′);
13:    return I_ext;
```

Nodes collapsing (Line 2). Nodes is used in this section with the the same meaning of state. As noted in [CGP99], a simpler intersection where $\{0, 1, 2\}$ are not necessary can be obtained in cases where either the model or the

claim only contains accepting states. In those cases, the procedure of nodes collapsing is even simpler, only requiring to ignore the third label with values $\{0, 1, 2\}$ and not having to collapse any nodes since, given a model state q_i and a claim state p_j, there only exist at most two nodes with label (q_i, p_j). Nevertheless, also in situations where both automata contain non-accepting states, we should consider a simplified version of intersection.

The simplification collapses all states that show the same first two components of the label (q and p) keeping the node that has the higher number as a third component. Given three nodes (in the most complicated case) $(q_i, p_m, 0)$, $(q_i, p_m, 1)$ and $(q_i, p_m, 2)$, we obtain a new node (q_i, p_m) having as incoming transitions the union[1] of the incoming transitions of the three unified nodes and, similarly, as outgoing transitions, the union of the outgoing transitions of all unified nodes. Whenever a cycle is found among $\{0, 1, 2\}$ components with the same first two labels, it is translated into a loop on the new node, that is marked as accepting.

Figure 3.5 shows an example of what the output of this step should look like. In this case we collapse $\{(q_2, p_2, 0), (q_2, p_2, 1), (q_2, p_2, 2)\}$ into (q_2, p_2) and $\{(q_1, p_1, 0), (q_1, p_1, 1)\}$ into (q_1, p_1). Notice that, since the set of nodes deriving from $q_2 \in Q_{\mathcal{M}}$ and $p_2 \in Q_{\bar{\Phi}}$ contains an accepting node, we can collapse the whole set into a unique accepting node.



(**a**) Before collapsing (**b**) After collapsing

Figure 3.5: Example of nodes collapsing

This procedure assures that no information is lost for the purpose of the proof. As a matter of fact, the counter labels $\{0, 1, 2\}$ are, in principle, used

[1] Union must be considered with its meaning in set theory.

to guarantee that accepting states from both the automata (together, not individually) appear infinitely often. Since the intersection has been calculated in its classical version, if any cycle between states exists, it is already been spotted during this procedure. If the cycle appears among three states with same q and p, we keep track of it by adding a loop on the node into which the nodes are collapsed. If the cycle appeared among nodes with different q and p components, it is preserved by the procedure.

The rules of the presented approach are based on the semantic content of the nodes of the model and the claim, combined. Their combination, irrespective of the third component of the node, has the same meaning. For the purposes of the proof, then, it is absolutely legitimate to disregard labels $\{0, 1, 2\}$. This choice indeed simplifies our algorithms of proof construction but does not change anything in the semantic of the automaton.

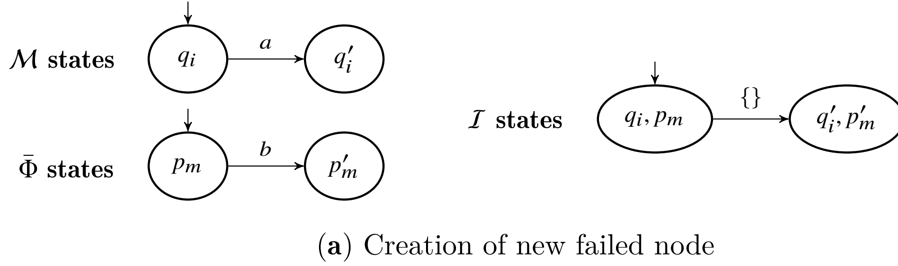
Failed A failed node is a node obtained by combining two transitions whose labels are conflicting, i.e., the propositional assignment on the model transition does not satisfy the propositional formula on the claim transition. A failed transition is an outgoing transition from a regular node, entering a failed node. Failed nodes do not have successors.

Definition 3.1 (Failed state and transition)**.** Let $q_i, q_i' \in Q_{\mathcal{M}}$ and $p_m, p_m' \in Q_{\bar{\Phi}}$. If there exists a transition $(q_i, a, q_i') \in \Delta_{\mathcal{M}}$ and a transition $(p_m, b, p_m') \in \Delta_{\bar{\Phi}}$ with $a \neq b$, then $(q_i', p_m') \in Q_{\mathcal{I}}$ is a *failed node* (*failed state*) and, being $\{\}$ an empty label, $\left(\langle q_i, p_m\rangle, \{\}, \left\langle q_i', p_m' \right\rangle\right) \in \Delta_{\mathcal{I}}$ is a *failed transition.*

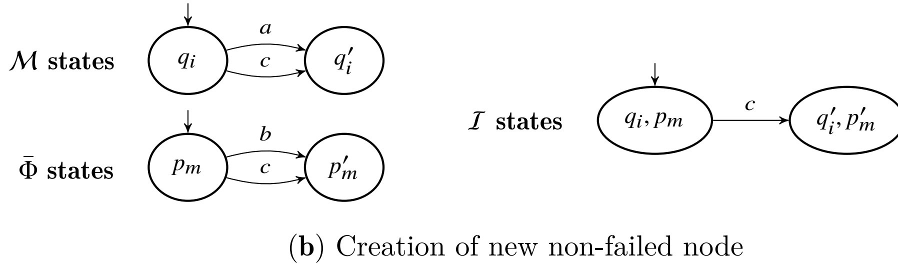
Notice that the failed node (q_i', p_m') could already belong to the intersection automaton. This happens if q_i' and p_m' are the destination nodes of two transitions (the first in the model and the second in the automaton of the negated property) that are labeled with the same symbol. We only add a failed node if it did not previously exist in the original non-extended intersection.

Figure 3.6 shows the cases that can arise when extending the intersection automaton computed in [MSG15] with the addition of failed nodes. Figure 3.6a represents the situation where all the labels going from q_i to q_i' and from p_m to p_m' do not match, therefore a failed node (q_i', p_m') is created and its incoming transition from (q_i, p_m) holds no propositional letters. Figure 3.6b represents the situation where a new node (q_i', p_m') could be created both as a destination of a transition labeled "c" coming from (q_i, p_m), and as a destination of a failed intersection (deriving from the non-matching transitions (q_i, a, q_i') and (p_m, b, p_m')). Notice that the intersection has been computed according to [MSG15] procedure *before* starting the process of adding failed nodes; therefore transition $\left(\langle q_i, p_m\rangle, c, \left\langle q_i', p_m' \right\rangle\right)$ already exists and node (q_i', p_m') does to. No failed transition is added. A last possible case is described in Figure 3.6c: node (q_i', p_m') existed in the original intersection, because q_i' and p_m' are des-

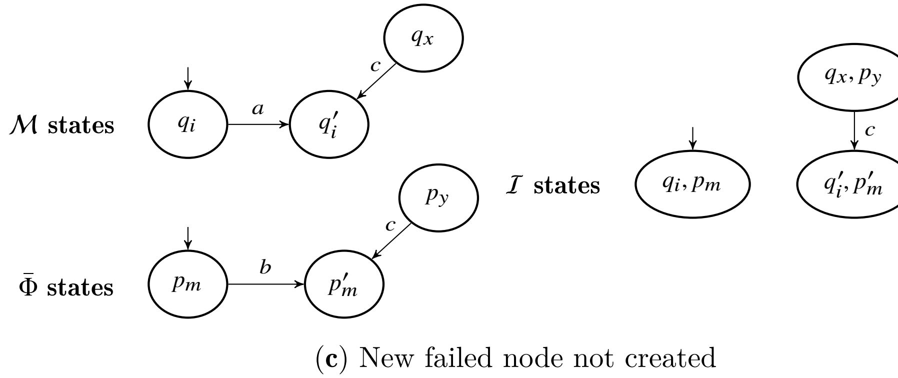
tination of two transitions with the same label, “c”. No failed transition is added from (q_i, p_m).



(**a**) Creation of new failed node



(**b**) Creation of new non-failed node



(**c**) New failed node not created

Figure 3.6: Failed nodes generation: possible cases

Lines 5- 12 of Algorithm 3.3 refer to the procedure used to add failed nodes to the intersection automaton built by [MSG15]. To prevent from computing the intersection automaton all from scratch, failed nodes can be found considering all combinations of the Cartesian product between model states and claim states that do not belong to the already computed intersection. We only extract the combinations that did not previously appear, excluding nodes generated by the combination with an initial model state (because they would not conclude any existing path). For each possible new failed node,

we consider all possible nodes of the original intersection that could lead to it and, then, add the corresponding failed transitions (Lines 7 - 10). Finally, we add the failed node (Line 12).

Another way to obtain the same result is to directly modify the intersection rule used in [MSG15], by adding the possibility to build *failed transitions* whenever model and claim transitions do not match. In case these new transitions led to not previously existing nodes, their destination nodes are added too, as *failed nodes*.

Failed nodes are needed to create a starting point for the proof, in fact its axioms. They correspond to the end of a run that does not cycle on accepting states, and, therefore, does not allow the model to satisfy the negated property. Keeping in mind the purpose of a failed node, in the remainder of the section we treat as "failed" also the nodes belonging to the original intersection that have no successors. Exactly as the failed nodes just added, they are, in fact, representative of non accepting runs that end.

Semantics of the intersection automaton For the purpose of our proof, we should consider the intersection automaton with a particular semantic that derives from the correspondence between Büchi automata and a Kripke structures. A translation procedure between the two formalisms is introduced in [CGP99].

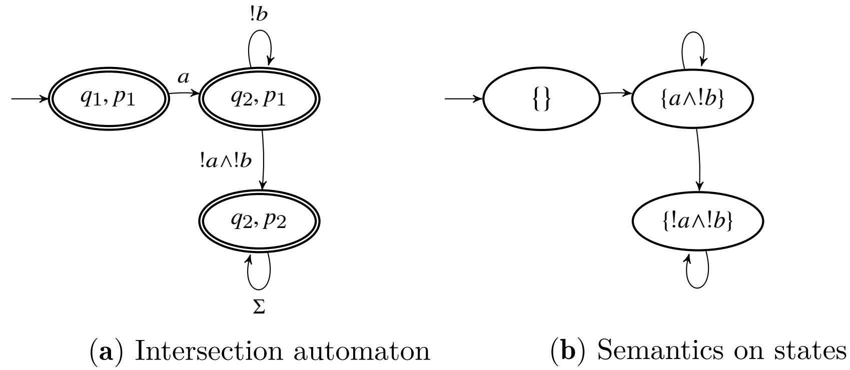
Definition 3.2 (From Kripke structures to Büchi automata [CGP99]). A Kripke structure $\langle Q, R, Q^0, L \rangle$ where $L : S \to 2^{AP}$, can be transformed into a Büchi automaton $\mathcal{A} = \langle \Sigma, Q \cup \{\iota\}, \Delta, \{\iota\}, Q \cup \{\iota\} \rangle$, where $\Sigma = 2^{AP}$, such that $(q, a, q') \in \Delta$ for $q, q' \in Q$ if and only if $(q, q') \in R$ and $a = L(q')$. In addition, $(\iota, a, q) \in \Delta$ if and only if $q \in Q^0$ and $a = L(s)$.

Figure 3.7 exemplifies this concept. (q_1, p_1) is an initial node of the structure which does not hold any valid proposition: it is not entered by any transition. The propositional atoms that are true in state (q_2, p_1) are all the ones derived from the state's incoming transitions: a and $!b$. To obtain the propositions true in state (q_2, p_2), we conjoin the proposition on its incoming labels: $!a \wedge !b \wedge \Sigma = !a \wedge !b$.

This justifies the fact that, for our proof, we do not always consider the real initial states of the intersection automaton, but those directly reached by these, when the first ones are semantically empty.

Definition 3.3 (Semantically empty node). Let q be a node of the automaton $\mathcal{M}$. q is *semantically empty* if it has no incoming transitions, i.e., when the system is in state q, no proposition $prop \in \Sigma$ is true.

To conclude the description of the intersection extension step, in Figure 3.8, we show the intersection automaton of the previously presented railway crossing example of Figure 3.2. According to the definition of intersection automaton for incomplete automata (Definition 2.15) and to our extension, we



(**a**) Intersection automaton (**b**) Semantics on states

Figure 3.7: Semantics of intersection automaton

can observe four mixed states (q_2, p_1), (q_4, p_1), (q_2, p_2), and (q_4, p_2) (double dotted line) with their corresponding constrained transitions (also dotted), a failed node (q_5, p_1) (dashed line) with its corresponding failed transition (also dashed).

States (q_5, p_2) and (q_6, p_2) are marked with a red area because they form a component that has to be rejected. This basically means that, if the system gets refined in a way that satisfies the constraints computed for it, that area can never be reached, and, therefore, does not take part in the proof. This concept is explained in Section 3.2.3.

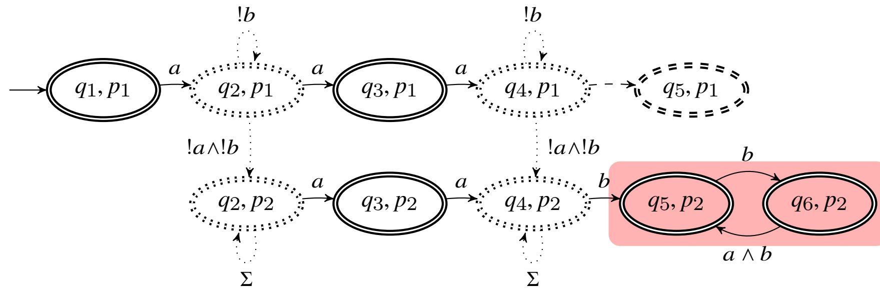


Figure 3.8: Intersection automaton for the railway crossing example

3.2.2 Identification of strongly connected components

The purpose of this section is to understand how to treat the components of the intersection automaton and how to organize them for the construction of the rules. There is a bijective correspondence between a SCC and a rule of our proof. We remind the SCCs definition specifically for our intersection atomaton.

Definition 3.4 (SCCs of an automaton [Tar72]). Given an automaton $\mathcal{I} = \langle \Sigma_\mathcal{I}, Q_\mathcal{I}, \Delta_\mathcal{I}, Q^0_\mathcal{I}, F_\mathcal{I} \rangle$, a strongly connected component of $\mathcal{I}$ is a maximal set of states $C \subseteq Q_\mathcal{I}$, such that for all $\{(q,p),(q',p')\} \subseteq C$, both $(\langle q,p\rangle, a, \langle q',p'\rangle) \in \Delta_\mathcal{I}$ and $(\langle q',p'\rangle, b, \langle q,p\rangle) \in \Delta_\mathcal{I}$, for arbitrary a and b, i.e., both (q,p) and (q',p') are reachable from each other.

In other words, two vertices of a directed graph are in the same component if and only if they are reachable from each other. The considered component is *strongly connected* if each state of the automaton appears in exactly one SCC. Any state, that is not on a directed cycle, forms a SCC all by itself: for example, a vertex whose in-degree or out-degree is 0, or any vertex of an acyclic graph.

One of the most famous algorithms to find SCCs is the Tarjan's algorithm ([Tar72]), which allows to find SCCs in linear time. We use this algorithm to identify the components of our intersection automaton. The procedure described is based on a depth-first search that begins from an arbitrary starting node and performs subsequent depth-first searches conducted on any nodes that have not been found yet.

Algorithm 3.4 Sorting of SCCs

```
1: procedure Sort(origSCCs)
2:     sortedSCCs ← {};
3:     mapToSucc ← {};
4:     for scc ∈ origSCCs do
5:         mapToSucc.set(scc, Successors(scc));
6:     while sortedSCCs.size < origSCCs.size do
7:         key = random(mapToSucc.keys);
8:         if key.getList = ∅ then
9:             for x ∈ mapToSucc.keys such that x.getList.contains(key) do
10:                x.getList.remove(key);
11:            sortedSCCs.add(key);
12:    return ordSCC;
```

After this basic step, we sort the set of SCCs to make sure that it satisfies the naturally induced partial order $\prec$, formally defined as follows:

Definition 3.5 (Naturally induced partial order $\prec$ [PZ01]). Let C and C' be two SCCs belonging to the same automaton. Then $C \prec C'$if there is a transition from some state in C to some state in C'.

Algorithm 3.4 describes how to establish this order. In our proof we need to complete the proof related to all the components C' such that $C \prec C'$, before we start dealing with C. *origSCCs* represents a support list that contains all SCCs, while *sortedSCC* represents the new sorted list to be filled. Lines 3 - 5

create a map that contains the SCCs, as key values, and for each one of these contains the list of all SCCs reachable from the key (key itself excluded). In lines 6 - 11, we pick a random SCC: if it does not have any successor, it is inserted in the sorted list. Already processed SCCs are eliminated from the map and so are the pointers to them. The sorted list *sortedSCCs* is returned by the procedure.

In our railway crossing example, we first obtain the following map:

$$\begin{bmatrix} \langle q_1p_1 \rightarrow \{q_2p_1\}\rangle \\ \langle q_2p_1 \rightarrow \{q_2p_2, q_3p_1\}\rangle \\ \langle q_3p_1 \rightarrow \{q_4p_1\}\rangle \\ \langle q_4p_1 \rightarrow \{q_4p_2, q_5p_1\}\rangle \\ \langle q_5p_1 \rightarrow \{\}\rangle \\ \langle q_2p_2 \rightarrow \{q_3p_2\}\rangle \\ \langle q_3p_2 \rightarrow \{q_4p_2\}\rangle \\ \langle q_4p_2 \rightarrow \{q_5p_2q_6p_2\}\rangle \\ \langle q_5p_2q_6p_2 \rightarrow \{\}\rangle \end{bmatrix}$$

This may lead, for example, to the following partially ordered sequence: q_5p_1, $q_5p_2q_6p_2$, q_4p_2, q_4p_1, q_3p_2, q_3p_1, q_2p_2, q_2p_1, q_1p_1.

3.2.3 Rules writing

We propose an extension of the rules presented in Section 2.4, that coincides with the formulation of [PZ01]. They are here modified and enriched for two main reasons: first, we would like them to support Büchi automata, whilst labeled generalized Büchi automata were previously considered; second, the original rules only supported completely specified systems. Now, they can be used on incomplete systems.

Each rule instance is composed of its *premise* and *conclusion* parts. Each procedure describing the construction of a rule initializes these two fields, fills them appropriately, and returns the entire data structure. Conclusions are valid only if all their premises are valid. In our rules, two kinds of premises are used: *validities* with a structure $state_{\mathcal{M}} \models \mu(state_{\bar{\Phi}})$ or $state_{\mathcal{M}} \models_P \mu(state_{\bar{\Phi}})$ (Definition 3.6 and 3.7) and *state-successors-definitions* of the form $state_{\mathcal{M}} \rightarrow successors(state_{\mathcal{M}})$ (Definition 3.8). Conclusions, instead, can only be validities.

Validities can be *sure-validities* (derived from executions of the model that satisfy the claim without depending on the replacement of transparent states) or *possible-validities* (depending on further refinements). Two different situations can determine that a validity is possible:

1. Validities referring to a model state that is still transparent are *possible* because no final conclusion can be derived in this case. Depending on the replacement that will substitute it, the validity can later become

sure, or remain *possible* (if the replacement contains transparent states itself).

2. All validities inside the conclusion of a rule that contains possible validities in its premise are tagged as "possible", inductively.

Only when all the uncertainties in the premise have been solved and the conclusion is not referred to a transparent state itself, we can consider the conclusion as *sure*.

Definition 3.6 (Sure-validity)**.** Given a state $q \in R_{\mathcal{M}}$ (being R the set of regular states of automaton $\mathcal{M}$) and a linear temporal logic formula $\mu(p)$ corresponding to state $p \in Q_{\bar{\Phi}}$, we say that $q \models \mu(p)$ if the configuration of the system $\mathcal{M}$ in state q satisfies the logic formula expressed by $\mu(p)$. The statement $q \models \mu(p)$ is called *sure-validity* and we say that q *models* $\mu(p)$.

Note that with "configuration of $\mathcal{M}$ in q" we mean the conjunction of all propositions of q's incoming transitions (see Paragraph "Semantics of the intersection automaton" of Section 3.2.1 for detail).

In incomplete systems, it can happen that the state q that models some formula $\mu(p)$ is not specified yet, but only contains a condition of entrance and of exit. To support this kind of situation, we introduce the concept of *possible-validity*.

Definition 3.7 (Possible-validity)**.** Given a state $q \in Q_{\mathcal{M}}$ and a linear temporal logic formula $\mu(p)$ corresponding to state $p \in Q_{\bar{\Phi}}$, we say $\models_P$ defines the relation of *possible-satisfiability* between a model state q and the sub-formula of state p of the claim when either $q \in T_{\mathcal{M}}$ or the fact that q satisfies $\mu(p)$ depends on the replacement of other transparent states of the model. The statement $q \models_P \mu(p)$ is called *possible-validity* and we say that q *possibly models* $\mu(p)$.

The second type of premises, *state-successors-definition*, describes, formally, the relation of successivity among model states in the intersection automaton.

Definition 3.8 (State successors definition)**.** We say that $q \rightarrow \{q_1, q_2, \ldots, q_m\}$, i.e., $q_1, q_2, \ldots, q_m \in Q_{\mathcal{M}}$ are successors of $q \in Q_{\mathcal{M}}$, iff there exist transitions $(\langle q, p\rangle, a, \langle q_1, p'\rangle)$, $(\langle q, p\rangle, a, \langle q_2, p'\rangle)$, ..., $(\langle q, p\rangle, a, \langle q_m, p'\rangle$ that belong to $\Delta_{\mathcal{I}}$, where a can be any label and p and p' any state of the claim automaton.

According to the characteristics of each SCC, a different rule is chosen and built. Algorithm 3.5 shows how the rule assignment works. For each SCC, three sets of states are created to help discriminating among situations. C represents the set of all states belonging to the same SCC. $Exit(C)$ contains all states that are directly reached by a node inside the SCC, but do not belong to C. $Enter(C)$ contains all predecessors of C that are outside of C.

Algorithm 3.5 Choice of rule for each SCC

```
1: procedure BUILDRULE(scc)
2:     C ← states ∈ scc;
3:     Exit(C) ←((successors of scc states)∉ scc) ∉ Proof.rejects;
4:     Enter(C) ←((predecessors of scc states)∉ scc) ∉ Proof.rejects;
5:     if !(C.size = 1 ∧ Enter(C) = ∅) then
6:         if C.size > 1 then
7:             if ∃x ∈ C t.c. x ∈ F_{I_ext} then
8:                 Proof.rejects ← Proof.rejects ∪ {scc};
9:                 Proof.SCC ← Proof.SCC \ {scc};
10:            else
11:                return RULEIND(scc);
12:        else
13:            c ← C.onlyNode;
14:            if c ∈ F_{I_ext} then
15:                if (∃ self-loop on c ∈ Δ_{I_ext} ∧!c.isMixed) then
16:                    Proof.rejects ← Proof.rejects ∪ {scc};
17:                    Proof.SCC ← Proof.SCC \ {scc};                ▷ 3.9a-b
18:                else
19:                    if Exit(C) = ∅ then
20:                        return RULEFAIL(c);                      ▷ 3.9c, 3.10a-c
21:                    else
22:                        if (∃ self-loop on c ∈ Δ_{I_ext} ∧ c.isMixed) then
23:                            return RULEIND(C);                   ▷ 3.10b
24:                        else
25:                            return RULESUCC(c);                  ▷ 3.9d, 3.10d
26:            else
27:                if Exit(C) = ∅ then
28:                    return RULEFAIL(c);                          ▷ 3.9e-g, 3.10e-g
29:                else
30:                    if ∃ self-loop on c ∈ Δ_{I_ext} then
31:                        return RULEIND(C);                       ▷ 3.9f, 3.10f
32:                    else
33:                        return RULESUCC(c);                      ▷ 3.9h, 3.10h
```

The if-condition at Line 5 excludes from the proof the formal initial states (semantically empty as explained in Paragraph "Semantics of the intersection automaton" of Section 3.2.1) and possible unreachable states.

The second macro-distinction is made between SCCs with more than one node (Lines 6- 11) and SCCs with only one node (Lines 13- 33). These include both trivial SCCs[2], and also the single states with a self-loop.

[2] An SCC is *trivial* if made of a single vertex c and (c, c) is not an edge, *non-trivial* otherwise.

For SCCs of more states two cases are possible. If there is at least one accepting node, this corresponds, indeed, to a dangerous component (a behavior of the system violating the property) that will never be reached if the model is correctly refined; the component is added to a red-list *rejects* (Lines 8- 9). If there are no accepting states RuleInd is applied (Line 11).

For one-node-SCCs, the choice is more articulated, since we also need to differentiate between regular and mixed states. As far as regular states are concerned, according to various nested conditions, we cover the following cases:

- A single accepting node with a self-loop corresponds to an accepting run and it is therefore rejected (Lines 16- 17);
- We apply RuleFail both to an accepting node with no successors (Line 20), and to a non-accepting node with no successors (except possibly for itself) (Line 28);
- We apply RuleSucc to a trivial SCC (no loops) with successors (Lines 25 and 33).
- We apply RuleInd to a non accepting node with a self-loop and other successors (this counts as a non-trivial SCC, Lines 23 and 31);

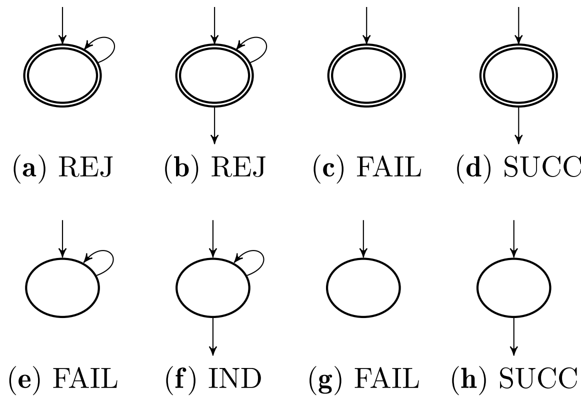


Figure 3.9: Schematization of one-node-SCC cases

As far as mixed states are concerned, we remind that they are states whose content is still unknown. Their content will be clear once they get replaced with other sub-automata.

Mixed accepting states with a self-loop do not need to be rejected, because they are not final yet, needing a replacement that can change their internal behavior. With mixed states no difference is made between accepting and non-accepting states. The kind of rule is decided in the same way as with regular

states. We make sure, though, that the conclusion is marked as *possible*, meaning that, whatever validitiy it holds, it needs to be further verified.

Figure 3.9 shows the different scenarios we might encounter when analyzing components of the graph with only one regular node. Figure 3.10 is its equivalent for mixed stats. Notice that, along Algorithm 3.5 we have taken note of which line refers to the cases represented in the two figures.

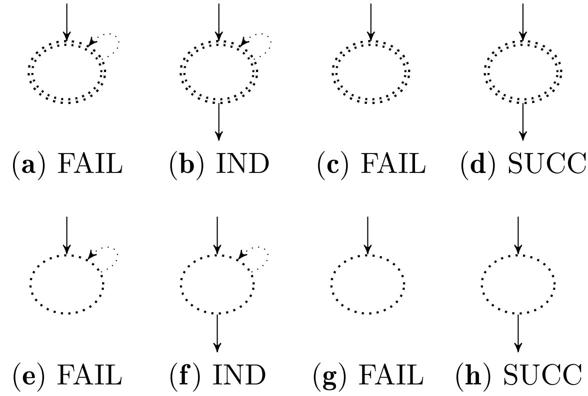


Figure 3.10: Schematization of one-mixed-node-SCC cases

Components to reject

We maintain a list of components of the intersection automaton that are not supposed to belong to a safe intersection automaton. In fact, they represent behaviors that possibly violate the property. Conversely, in our approach, we want to prove that the property is possibly satisfied.

In the railway crossing example, we exclude the component $\{(q_5, p_2), (q_6, p_2)\}$ (notice that is it marked with a red area in Figure 3.8). Note that, if the transparent state q_4 is properly replaced, there will be no transitions going from the mixed state (q_4, p_2) to the component that we are excluding, which means that the property is possibly satisfied.

RuleFail (Algorithm 3.6)

The failure axiom is applied to all failed states of the intersection automaton (added through the procedure described in Section 3.2.1) and to already existing states of the original intersection automaton that have no successors (and have not already been rejected).

Axiom (Failure axiom (RuleFail))**.** *Let (q,p) be a failed node. Then, we can conclude that* $q \models \mu(p)$.

The justification for this axiom is simple: the node has failed because we have checked the propositional labels of state q's incoming transitions against the temporal formula on p, and the LTL formula has failed to hold. Thus, $q \models \neg\eta(p)$. But, note that $\neg\eta(p) = \mu(p)$ (see Proposition 2.1). Therefore, $q \models \mu(p)$, which specifies that the formula $\mu(p)$ holds in the state q.

Notice that states without successors are obviously not leading to any dangerous cycle, e.g., a counterexample. They represent the end of a branch of the graph in which the search for counterexamples has failed. If the used model state is transparent, the validity output is *possible*, otherwise *sure*.

Algorithm 3.6 Apply RuleFail to SCC

```
1: procedure RuleFail(q, p)
2:     rule.prem ← {};
3:     if q ∈ T_M then
4:         rule.concl ← BuildValidity(q, p, possibly);
5:     else
6:         rule.concl ← BuildValidity(q, p, satisfy);
7:     return rule;
```

In our example (Figure 3.8), we apply the failure axiom to node (q_5, p_1), that is accepting and has no successors. Since q_5 is a regular node, the conclusion is *sure*.

$$\frac{-}{q_5 \models \mu(p_1)}$$

Another example of RuleFail application is given in another rule, that derives from node (q_4, p_2), that is mixed and failed (considering the component $\{(q_5, p_2), (q_6, p_2)\}$ as already rejected), thus corresponding to the case of Figure 3.10a. Since q_4 is a transparent node, the conclusion is *possible*.

$$\frac{-}{q_4 \models_P \mu(p_2)}$$

RuleSucc (Algorithm 3.7)

The successors rule is applied to trivial SCCs, i.e., single states without any self-loop. We modified the "Successor rule" [PZ01] (see Section 2.4).

Proposition (Successors Rule (RuleSucc))**.** *Let (q,p) be a successful node, such that p has n successors $p_1, \ldots, p_n$, and q has m successors $q_1, \ldots, q_m$. Then, from the premises $s \to \{s_1, \ldots, s_m\}$, and* For each $1 \leq i \leq m$, $q_i \models \bigwedge_{j=1,n} \mu(p_j)$, *we derive the conclusion $q \models \mu(p)$.*

This means that, given information on what holds in (q, p) successors and being (q, p) a node that does not exhibit any criticality, we can propagate its "safeness" upwards. The validity of this proof rule stems from the correctness of the construction. Indeed, note that the *failure axiom* is a special case of the *successors rule* when the addressed node has no successors.

Algorithm 3.7 Apply RuleSucc to SCC

```
1:  procedure RuleSucc(q, p)
2:      qSucc ← {};
3:      pSucc ← {};
4:      rule.concl ← {};
5:      for q′ s.t. ∃(⟨q, y⟩, a, ⟨q′, y′⟩) ∈ Δ_{I_ext} do
6:          rule.prem ← rule.prem ∪ [q → {q′}];
7:      if q.isTrasp then
8:          rule.prem ← rule.prem ∪ [q → {q}];
9:      for q′ s.t. ∃(⟨q, y⟩, a, ⟨q′, y′⟩) ∈ Δ_{I_ext} do
10:         qSucc ← qSucc ∪ {q′};
11:     for p′ s.t. ∃(⟨x, p⟩, a, ⟨x′, p′⟩) ∈ Δ_{I_ext} do
12:         pSucc ← pSucc ∪ {p′};
13:     for (q′, p′) ∈ qSucc × pSucc do
14:         if (q′ ∈ T_M ∨ depGraph(q′, p′) ≠ ∅) then
15:             rule.prem ← rule.prem ∪ BuildValidity(q′, p′, possibly);
16:             depGraph(q, p) ← depGraph(q, p) ∪ (q′, p′);
17:         else
18:             rule.prem ← rule.prem ∪ BuildValidity(q′, p′, satisfy);
19:     if (q ∈ T_M ∨ depGraph(q, p) ≠ ∅) then
20:         rule.concl ← rule.concl ∪ BuildValidity(q, p, possibly);
21:     else
22:         rule.concl ← rule.concl ∪ BuildValidity(q, p, satisfy);
23:     return rule;
```

The behavior of the *Successors Rule* is described in Algorithm 3.7. Notice that, at Line 8, we consider the fact that a transparent node is a successor of itself (i.e., it is represented with a self-loop because inside that state the run can progress).

In our railway crossing example (Figure 3.8), we find an application of this rule to node (q_3, p_1) of the intersection automaton, that corresponds to the case of Figure 3.9d, an accepting node with successors and no loops. The *possible-validity* of the conclusion is determined by the presence of a possible-validity in the premises of the rule.

$$\frac{\begin{array}{c} q_3 \rightarrow \{q_4\} \\ q_4 \models_P \mu(p_1) \end{array}}{q_3 \models_P \mu(p_1)}$$

RuleInd (Algorithm 3.8)

The induction rule is applied to any non-trivial component that has no accepting state (otherwise it would have been rejected).

Proposition (Induction Rule (RULEIND))**.** *Let C be a SCC of $\mathcal{I}$. Let Exit(C) be the set of states not in C, with an incoming arrow from a node in C. Then, from the facts that* For each $(q, p) \in Exit(C)$, $q \models \mu(p)$ *and* For each $(q, p) \in C$, $q \rightarrow succ(q)$*, we can derive that* For each $(q, p) \in C$, $q \models \mu(p)$.

Algorithm 3.8 Apply RULEIND to SCC

```
1: procedure RULEIND(scc)
2:     C ← states ∈ scc;
3:     Exit(C) ←successors of states ∈ scc;
4:     rule.prem ← {};
5:     rule.concl ← {};
6:     for (q', p') ∈ Exit(C) do
7:         if (q' ∈ T_M ∨ depGraph(q', p') ≠ ∅) then
8:             rule.prem ← rule.prem ∪ BUILDVALIDITY(q', p', possibly);
9:             depGraph(q, p) ← depGraph(q, p) ∪ (q', p');
10:        else
11:            rule.prem ← rule.prem ∪ BUILDVALIDITY(q', p', satisfy);
12:    for q ∈ C ∧ q' s.t. ∃(⟨q, y⟩, a, ⟨q', y'⟩) ∈ Δ_I_ext do
13:        rule.prem ← rule.prem ∪ [q → {q'}];
14:    if q.isTrasp then
15:        rule.prem ← rule.prem ∪ [q → {q}];
16:    for (q, p) ∈ C do
17:        if (q ∈ T_M ∨ depGraph(q, p) ≠ ∅) then
18:            rule.concl ← rule.concl ∪ BUILDVALIDITY(q, p, possibly);
19:        else
20:            rule.concl ← rule.concl ∪ BUILDVALIDITY(q, p, satisfy);
21:    return rule;
```

Notice that, at Line 15 of Algorithm 3.8, we consider the fact that a trasparent node is a successor of itself (i.e., it is represented with a self-loop because, inside that state, the run can progress).

In the railway crossing example, we can apply the induction rule to node (q_4, p_1), that is mixed, with successors and a loop, corresponding to the case

of Figure 3.10b. The conclusion is only *possible* both because it is referred to the transparent state q_4, and because one premise for the rule ($q_4 \models_P \mu(p_2)$) is possible.

$$\frac{\begin{array}{c} q_4 \rightarrow \{q_4, q_5\} \\ q_5 \models \mu(p_1) \\ q_4 \models_P \mu(p_2) \end{array}}{q_4 \models_P \mu(p_1)}$$

A more elaborate example of application of RULEIND is presented in the case study of Chapter 5, where a SCC with more than one state is analyzed.

3.2.4 Rules conjunction

The final goal of this step is to establish a proof that $\sigma \models \phi$ for every execution σ of the automaton intersection $\mathcal{M}$. This is expressed by $\mathcal{M} \models \phi$ (see Theorem 2.1).

RULECONJ is useful to unify all conclusions that have been found in the previous rules. In general, it allows to build conclusions on the initial states, that here represent the whole model. Due to our choice of using Büchi automata and the semantics described in Paragraph "Semantics of the intersection automaton" of Section 3.2.1, in some cases we require a particular choice of initial states. Roughly speaking, the "initial states" do not always correspond to the initial states $(q^0, p^0) \in Q^0_{\mathcal{I}}$ of the intersection automaton. To explain this concept, we refer to Figure 3.11. Intuitively, the choice is based on the requirement that an initial node needs to be semantically non-empty.

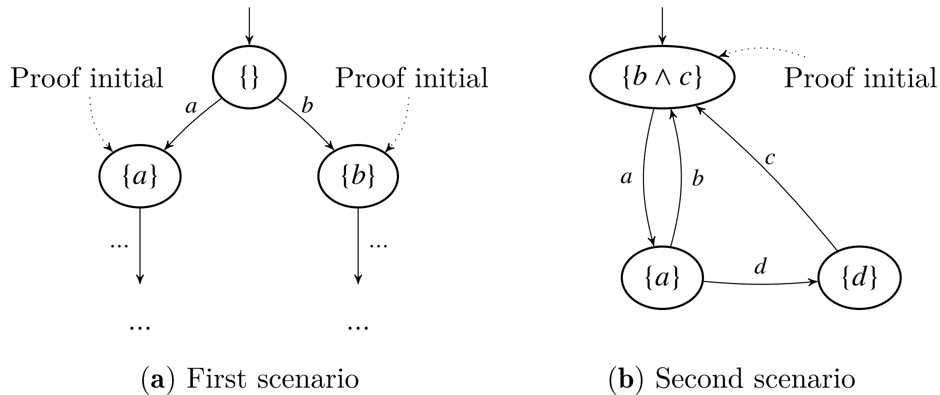


Figure 3.11: Artificial initial node choice

In Figure 3.11a we can observe that the real initial node is semantically empty because it does not have any incoming transition. Therefore, we use

its immediate successors as *artificial initial states* for RuleConj. On the contrary, in Figure 3.11b, we observe that the initial node is not empty because it is reached by the incoming transitions labeled with "b" and "c". We can, therefore, use it as requested in rule RuleConj. Notice that, when analyzing an intersection automaton, we could have a mixed situation, e.g., if we had two initial states, one of which is semantically empty and the other is not, we should use one convention (using its successors) in the first case, and the other convention (use the real initial node) in the second case.

Recalling the definition of semantically empty node, i.e., a state that does not contain propositions (see Definition 3.3), we can define the initial node used for the proof.

Definition 3.9 (Artificial Initial Node)**.** Let q^0 be a semantically non-empty state of $Q^0_{\mathcal{M}}$, or an immediate successor of an initial state of $\mathcal{M}$ that is semantically empty. Then q is an *artificial initial node (state)* for the automaton $\mathcal{M}$.

To these artificial states, we can apply the conclusive rule of the proof.

Proposition (Conjunction Rule (RuleConj))**.** *1. Let q^0 be an artificial initial node belonging to $Q_{\mathcal{M}}$. Let $p_1, \ldots, p_n$ be all the states of $\bar{\Phi}$ such that (q^0, p_i) is a node in the intersection. Then, apply* RuleConj*, which takes $q^0 \models \mu(p_1), \ldots, q^0 \models \mu(p_n)$ and $\bigwedge_{i=1,n} \mu(p_i) \to \phi$ as premises, and concludes $q^0 \models \phi$. 2. Apply* RuleConj *to all artificial initial states of $\mathcal{M}$.*

Algorithm 3.9 Apply RuleConj to conclude the proof

```
1:  procedure RuleConj(φ, I)
2:      rule.prem ← {};
3:      init ← ArtificialInitStates(M);
4:      for i ∈ init do
5:          for p such that (i, p) ∈ Q_{I_ext} do
6:              if i ∈ T_M ∨ depGraph(i, p) ≠ ∅ then
7:                  rule.prem ← rule.prem ∪ BuildValidity(i, p, possibly);
8:                  depGraph(q^0, φ) ← depGraph(q^0, φ) ∪ (i, p);
9:              else
10:                 rule.prem ← rule.prem ∪ BuildValidity(i, p, satisfy);
11:     rule.prem ← rule.prem∪ [Conjunction(claimStates)→ φ];
12:     if depGraph(q^0, φ) ≠ ∅ then
13:         for i ∈ init do
14:             rule.concl ← BuildValidity(i, φ, possibly);
15:     else
16:         for i ∈ init do
17:             rule.concl ← BuildValidity(i, φ, satisfy);
18:     return rule;
```

Algorithm 3.9 outlines the steps of this final rule. Line 3 represents the choice of the artificial initial states to be performed at the beginning of the rule construction. Notice that, by deriving that all artificial initial states of the model satisfy the property, we intend to say that the whole model (i.e., all the paths that it can generate) satisfies the property (Theorem 2.1).

The conjunction in our railway crossing example (Figure 3.8) is built in the following rule: the conclusion is only *possible* both because the model contains transparent states, and because the validities contained in the rule's premise are still *possible*.

$$\frac{\begin{array}{l} q_2 \models_P \mu(p_1) \\ q_2 \models_P \mu(p_2) \\ \mu(p_1) \land \mu(p_2) \to \phi \end{array}}{q_2 \models_P \phi}$$

3.2.5 Dependency graph

The dependency graph is a structure that supports the computation of the replacements sub-proofs. It is needed to keep trace of the dependencies between rules in the proof. The validities in the conclusion of a rule can be "possible" only if they refer to a model state which is still transparent, or if any of their premises are "possible" too. As an example, let us consider a rule of this form:

$$\frac{\begin{array}{l} q_2 \to \{q_2, q_3\} \\ q_3 \models_P \mu(p_1) \\ q_2 \models_P \mu(p_2) \end{array}}{q_2 \models_P \mu(p_1)}$$

The *dependency graph* structure (*depGraph*) is a map that links each *key*-entry (a SCC) with a list of other SCCs (*depList*) on which its validity depend. If we consider the rule above, for example, the *key* (q_2, p_1) is mapped to the *depList* $\{(q_3, p_1), (q_2, p_2)\}$ indicating that, for $q_2 \models_P \mu(p_1)$ to become a *sure-validity* ($q_2 \models \mu(p_1)$) we first need to solve the *possible-validities* $q_3 \models_P \mu(p_1)$ and $q_2 \models_P \mu(p_2)$. Solving a dependency is a process that requires to go through a refinement of the model. Suppose, for this explanation, that q_3 is transparent in the initial model, and that the developer replaces this state with a complete Büchi automaton that satisfies the constraint computed as described in [MSG15]. This means that a sub-proof stating that $q_3 \models \mu(p_1)$ is built and this *sure-validity* can ultimately be used at higher levels. The element (q_3, p_1), that corresponds to the first one in the *depList* of the entry (q_2, p_1) of the *depGraph* can now be eliminated. The *possible-validity* $q_2 \models_P \mu(p_1)$ now only depends on (q_2, p_2), that refers to the same model state q_2 and will be, therefore, solved by a preceding rule, rather than from a refinement yet to come.

3.2.6 Output of the proof

- Node $(\mathbf{q_5}, \mathbf{p_1})$. RuleFail. $q_5 \models \mu(p_1) = \bigcirc(low\,\mathcal{U}\,out) \vee out$
- SCC = $\{(\mathbf{q_5}, \mathbf{p_2}), (\mathbf{q_6}, \mathbf{p_2})\}$. Rejected.
- Node $(\mathbf{q_4}, \mathbf{p_2})$. RuleFail. $q_4 \models_P \mu(p_2) = low \vee out$
- SCC = $(\mathbf{q_4}, \mathbf{p_1})$, Exit(SCC) = $\{(q_5, p_1), (q_4, p_2)\}$. RuleInd.

$$\frac{\begin{array}{ll} q_4 \rightarrow \{q_4, q_5\} & \\ q_4 \models_P \mu(p_2) & = low \vee out \\ q_5 \models \mu(p_1) & = \bigcirc(low\,U\,out) \vee out \end{array}}{q_4 \models_P \mu(p_1) \quad = \bigcirc(low\,U\,out) \vee out}$$

- SCC = $(\mathbf{q_3}, \mathbf{p_2})$, Exit(SCC) = $(\mathbf{q_4}, \mathbf{p_2})$. RuleSucc.

$$\frac{\begin{array}{ll} q_3 \rightarrow \{q_4\} & \\ q_4 \models_P \mu(p_2) & = low \vee out \end{array}}{q_3 \models_P \mu(p_2) \quad = low \vee out}$$

- SCC = $(\mathbf{q_3}, \mathbf{p_1})$, Exit(SCC) = $(\mathbf{q_4}, \mathbf{p_1})$. RuleSucc.

$$\frac{\begin{array}{ll} q_3 \rightarrow \{q_4\} & \\ q_4 \models_P \mu(p_1) & = \bigcirc(low\,U\,out) \vee out \end{array}}{q_3 \models_P \mu(p_1) \quad = \bigcirc(low\,U\,out) \vee out}$$

- SCC = $(\mathbf{q_2}, \mathbf{p_2})$, Exit(SCC) = $(\mathbf{q_3}, \mathbf{p_2})$. RuleInd.

$$\frac{\begin{array}{ll} q_2 \rightarrow \{q_2, q_3\} & \\ q_3 \models_P \mu(p_2) & = low \vee out \end{array}}{q_2 \models_P \mu(p_2) \quad = low \vee out}$$

- SCC = $(\mathbf{q_2}, \mathbf{p_1})$, Exit(SCC) = $\{(\mathbf{q_3}, \mathbf{p_1}), (\mathbf{q_2}, \mathbf{p_2})\}$. RuleInd.

$$\frac{\begin{array}{ll} q_2 \rightarrow \{q_2, q_3\} & \\ q_3 \models_P \mu(p_1) & = \bigcirc(low\,U\,out) \vee out \\ q_2 \models_P \mu(p_2) & = low \vee out \end{array}}{q_2 \models_P \mu(p_1) \quad = \bigcirc(low\,U\,out) \vee out}$$

- RuleConj.

$$\frac{\begin{array}{ll} q_2 \models_P \mu(p_1) & = \bigcirc(low\,U\,out) \vee out \\ q_2 \models_P \mu(p_2) & = low \vee out \\ \mu(p_1) \wedge \mu(p_1) \rightarrow \phi & = low\,U\,out \end{array}}{q_2 \models_P \phi \quad = low\,U\,out}$$

Listing 3.1: Deductive proof of $\mathcal{M} \models \phi$ for the railway crossing system

For clarity and completeness of the proof output, we now map subformulas indicated and memorized as $\mu(claimState)$ with the real LTL subformulas. These can be calculated as described at the beginning of this section with the tableau in Figure 3.4, as explained theoretically in Section 2.2.2.

After this last step, the output of the railway crossing example is shown as in the Listing 3.1.

3.3 Computing the sub-proofs

This section integrates the procedure described in Section 3.2 with a method to compute sub-proofs by exploiting the *Replacement Checking* procedure described in Section 2.3.3 ([MSG15]). Considering a transparent state $t \in Q_{\mathcal{M}}$, our sub-proof shows *why* a chosen replacement $\mathcal{R}_t$ (Definition 2.9), i.e., a sub-automaton that substitutes state t, satisfies (or possibly-satisfies) the sub-property $\bar{\mathcal{S}}_t$ computed for state t (Definition 2.16).

This procedure supports the incremental development *modus operandi* of designers with the opportunity to compute an *ad hoc* proof circumscribed to only the replaced state. Indeed, proving a single module at a time, without having to consider the entire refined system, saves the user a considerable effort at each refinement round.

After this module is proven, the information derived can be plugged into the master proof, allowing to solve its rules whose conclusions are still only *possible*. We are speaking of the rules whose premise contains *possible*-validities connected to the mentioned state t.

While in Section 3.2 we used the intersection between an automaton representing the model $\mathcal{M}$ and an automaton representing the negated claim $\bar{\Phi}$ to derive the master proof, in the this section we use the intersection between the replacement $\mathcal{R}_t$ and the sub-property $\bar{\mathcal{S}}_t$ to build the sub-proof related to state t.

The procedure to compute a sub-proof that specifies why a replacement satisfies the related condition ($\mathcal{R}_t \models \mathcal{S}_t$) is very similar to the one to compute a proof that the model satisfies the requested claim ($\mathcal{M} \models \phi$). The only difference is represented by the steps to extend the intersection automaton. Since $\mathcal{R}_t$ and $\bar{\mathcal{S}}_t$ are not simple automata, the intersection between them is defined in a slightly different way than the one between $\mathcal{M}$ and $\bar{\Phi}$, as explained in Definition 2.18. The structure we need to analyze, in this case, is a Büchi automaton with *in-transition* and *out-transitions* in addition.

In the next two sections we describe how the intersection structure needs to be modified and how the rules can be applied by analyzing it.

3.3.1 Intersection for the sub-proof

Algorithm 3.10 shows the modifications needed to the extended automaton $\mathcal{I}_t$ calculated as the intersection between the sub-automaton $\mathcal{M}_t$ of the replacement $\mathcal{R}_t$ of state t, and the sub-automaton $\bar{\mathcal{P}}_t$ of the sub-property $\bar{\mathcal{S}}_t$ calculated for state t.

Algorithm 3.10 Extension of intersection automaton for the sub-proof

1: **procedure** EXTENDINTERSECTIONSUBPROOF($\mathcal{I}_t$)
2: $\quad \mathcal{I}_{t_{ext}} \leftarrow$EXTENDINTERSECTION($\mathcal{I}_t, \mathcal{R}_t, \bar{\mathcal{S}}_t$);
3: $\quad transitionsToPorts()$;
4: $\quad inportsToInitialStates()$;
5: $\quad findArtificialInitialStates()$;
6: $\quad addBlueOutports()$;
7: $\quad$ **return** $\mathcal{I}_{t_{ext}}$;

The algorithm includes a first phase (Line 2) that performs the same transformations of the procedure EXTENDINTERSECTION that we used for the master proof (Algorithm 3.3). In this function, the phase of nodes collapsing remains unchanged and so does the rule to add failed states to the intersection computed as in [MSG15] (specified in Definition 2.18).

We keep the same semantics described in Paragraph "Semantics of the intersection automaton" of Section 3.2.1. This means that inside each state of the intersection automaton, all and only the propositions that label its incoming transitions are valid.

The additional aspect we need to consider is the presence of incoming and outgoing transitions. First of all, we would like to complete the in-transitions by adding also their source state, and the out-transitions by adding their destination state (Line 3). For this reason we introduce the concept of *port*.

Definition 3.10 (In-ports and out-ports)**.** Let $\mathcal{I}_t$ be the intersection computed between the replacement $\mathcal{R}_t$ and it sub-property $\bar{\mathcal{S}}_t$. And let Δ_t^{inI} and Δ_t^{outI} be its *in-transitions* and *out-transitions.* Then, for each $tr_{in} \in \Delta_t^{inI}$, the source state of tr_{in} is called *in-port* and for each $tr_{out} \in \Delta_t^{outI}$, the destination state of tr_{out} is called *out-port.*

According to the color that marks tr_{in} (tr_{out}), its source (destination) state inherits the same label (*Red*, *Yellow*, or *Green*, as specified in Section 2.3.3).

The intersection computed in [MSG15] represents incoming transitions sources and outgoing transitions destinations by only showing the model state that composes the intersection state. Our procedure requests expanding this port (that is currently labeled with a model state q) by adding the information from the claim for each existing node of the intersection that is related to node q. Basically, considering n ports labeled with the model state q, if the

intersection automaton $\mathcal{I} = \mathcal{M} \cap \bar{\Phi}$ contains the states $(q, p_1), ..., (q, p_n)$, these states are added to the intersection automaton $\mathcal{I}_t = \mathcal{R}_t \cap \bar{\mathcal{S}}_t$ in substitution of the n ports labeled with q.

Line 4 indicates the step to transform the sub-property in-ports to the initial states of the analyzed automaton. The produced states are semantically empty (since we are not considering any edge that enters them), so artificial initial states (Definition 3.9) need to be chosen. Line 5 corresponds to this choice. They are the immediate successors of the *in-ports* of the intersection automaton and the proof is started from here.

Compared to *in-ports*, the role of *out-ports* is more interesting. We need to add a new type of *out-port* that we mark with *blue* color label (Line 6). We now distinguish three possible colors for out-ports.

***Red* ports** are never reached if the provided replacement satisfies or possibly-satisfies the sub-property (i.e., when the construction of the sub-proof is triggered). As a matter of fact, we remind that the situation that a developer must avoid at every cost when he/she designs a replacement $\mathcal{R}_t$ for state t is to design a component that allows $\bar{\mathcal{S}}_t$ to reach an outgoing transition marked as *red* from an incoming transition marked as *green*. For this reason, *red* ports never appear in the analyzed intersection and we should not worry about them.

***Yellow* ports** appear in intersections and have the meaning of postponing the problem to the next reached mixed state. They should be therefore considered as failed states because, in some way, once the run has reached this state, it is out of the danger indicated by the sub-property.

***Blue* ports**: sub-properties, and consequently the intersection automaton $\mathcal{I}_t$, do not include the outgoing transitions that do not lead to a *possibly*-violating or violating run. Constraints, indeed, show only *yellow* and *red* out-transitions. Differently from [MSG15], we are also interested in making "non existing" out-transitions explicit, to specify any possible link of the sub-intersection automaton with the exterior. We add these transitions and the corresponding destination states to the intersection and call them *blue transitions* and *blue ports*.

Definition 3.11 (*Blue* port)**.** Given a transparent state t of an incomplete model $\mathcal{M}$, its replacement $\mathcal{R}_t$, and its sub-property $\bar{\mathcal{S}}_t$, a *blue port* is a node of the intersection automaton $(\mathcal{R}_t \cap \bar{\mathcal{S}}_t)$ that can be reached from states of the intersection sub-automaton$\mathcal{M}_t \cap \mathcal{P}_t$, but does not already correspond to a *yellow* or a *red* port.

Blue ports are important for our purposes because we need to add all possible intersection states that can be reached by a run when leaving a sub-property. Indeed, as we observed in Paragraph "Failed" of Section 3.2.1, failed states indicate the end of a non-accepting path of the intersection automaton. We could, therefore, think of out-ports as the end of a sub-run (the one that

crossed the sub-property) that has not been trapped inside of it (it is not accepting, at least inside the sub-property). *Blue* and *yellow* ports acquire this role.

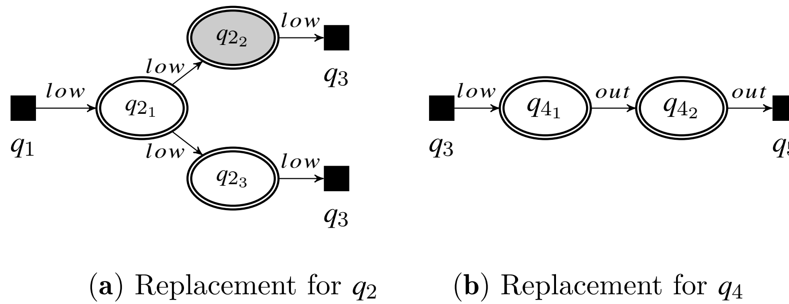
At the end of the EXTENDINTERSECTIONSUBPROOF algorithm, the states of the new intersection present a label of the form $(q_{t_j}, (q_t, p_i, x), y)$, being composed by three elements:

- a first label q_{t_j} where t represents the transparent state $t \in T_{\mathcal{M}}$ of the *master* model $\mathcal{M}$ and j represents the label of the state of the replacement $\mathcal{M}_t$;
- a second label (q_t, p_i, x) coming from the sub-property related to t;
- a third label computed according to the intersection rule described in Definition 2.13.

Because of the procedure of nodes collapsing, we can always ignore the third component after we have marked as accepting all states that have the third component equal to 2.

We now continue the railway crossing example presenting the computed constraint, two possible replacements for its transparent states, and the construction of their intersections.

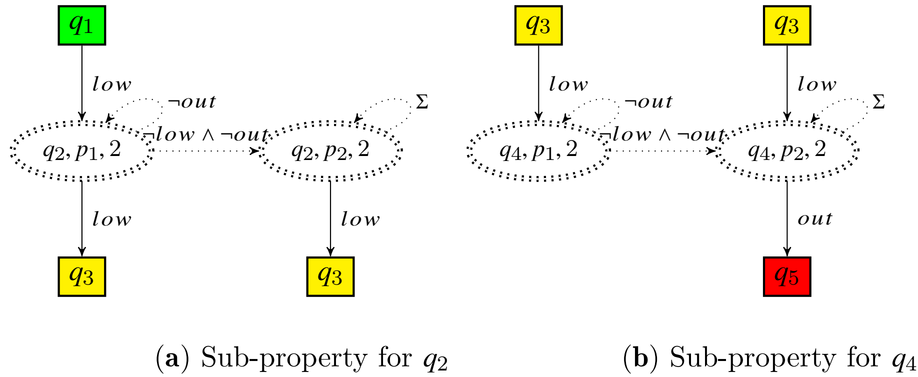
Example. Let us consider the model in Figure 3.2: we would like to replace state q_2 with the replacement in Figure 3.12a, and state q_4 with the replacement in Figure 3.12b. The replacement for q_2 contains a transparent state itself, q_{2_2}.



(**a**) Replacement for q_2 (**b**) Replacement for q_4

Figure 3.12: Replacements for the railway crossing example

In Figure 3.13, we consider the constraint C computed for the refinement of the system. The incoming and outgoing transitions are marked with arrows that reach squared boxes that represent the entrance and exit points of the sub-property.



(**a**) Sub-property for q_2 (**b**) Sub-property for q_4

Figure 3.13: Sub-properties for the railway crossing example

The sub-property $\bar{\mathcal{S}}_{q_2}$ in Figure 3.13a contains an automaton $\bar{\mathcal{P}}_{q_2}$, that has two states with self-loop and a transition connecting them. State $(q_2, p_1, 2)$ can be entered through the *green* in-port q_1 and exited through the *yellow* out-port q_3. State $(q_2, p_2, 2)$ can be exited though port q_3. The sub-property $\bar{\mathcal{S}}_{q_4}$, in Figure 3.13b, contains two runs that may yield to a violation of ϕ: in the first case, the sub-property is entered though the *yellow* q_3 port on the left, $(q_4, p_1, 2)$ is crossed, then $(q_4, p_2, 2)$, and the sub-property is exited through the *red* port q_5; in the second case, the sub-property is entered directly from the *yellow* port q_3 on the right, $(q_4, p_1, 2)$ is crossed and finally the *red* out-port is reached.

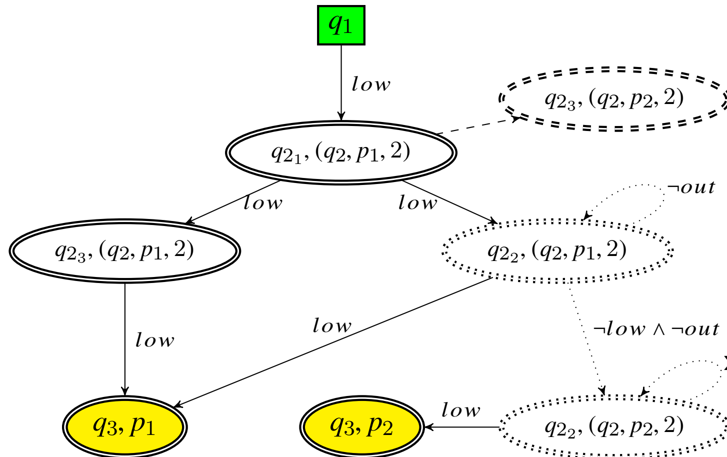


Figure 3.14: Intersection automaton for $\mathcal{R}_{q_2}$ and its sub-property

The intersections between the replacements of states q_2 and q_4 and their corresponding sub-properties are represented respectively in Figure 3.14 and

Figure 3.15. They are obtained using the procedure described in [MSG15] and follow the Definition 2.18.

We assume that a node collapsing step has already been performed on the intersections that we analyze in Figure 3.14 and Figure 3.15.

In the first shown intersection, we observe some of the features just described. We have added the failed transition and node $(q_{2_3}, (q_2, p_2, 2))$ marked with a dashed line. Then we have transformed the out-going *yellow* transitions to q_3 into two *yellow* out-ports (q_3, p_1) and (q_3, p_2). These are the states of intersection that are associated to the label q_3 that indicated the out-transitions. The *green* in-port has become the nominal initial node and the one directly reached from it, $(q_{2_1}, (q_1, p_2, 2))$, has acquired the role of artificial initial node.

Finally, no *blue* port needs to be added because q_3 is the only possible exit from the sub-property computed for q_2. We also find the mixed states $(q_{2_2}, (q_2, p_1, 2))$ and $(q_{2_2}, (q_2, p_2, 2))$ deriving from the presence of the transparent state q_{2_2} in the replacement.

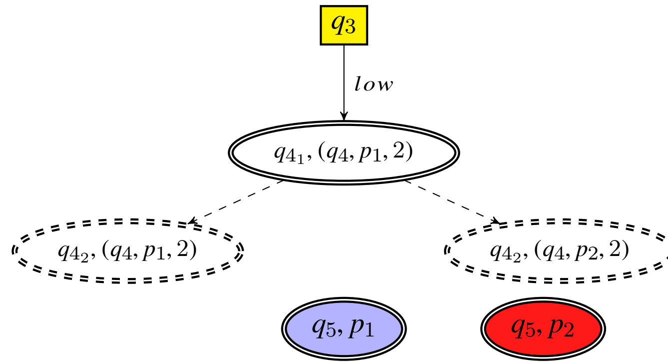


Figure 3.15: Intersection automaton for $\mathcal{R}_{q_4}$ and its sub-property

The intersection automaton for q_4 shows two failed states $(q_{4_2}, (q_4, p_1, 2))$ and $(q_{4_2}, (q_4, p_2, 2))$, and an artificial initial node $(q_{4_1}, (q_4, p_1, 2))$. Moreover, we observe that the *red* out-port to (q_5, p_2) is never reached. The *blue* out-port (q_5, p_1) is not reachable, therefore is eliminated before computing the proof.

3.3.2 Rules application

Once the intersection automaton is prepared in the described way, we can proceed with the same steps of Algorithm 3.2 (BuildProof). The SCCs are identified and sorted, and, then, the Algorithm 3.5 (BuildRule) is applied to each of them. Each rule contributes to the update of the dependency-graph as described in Section 3.2.5. Finally, the sub-proof is closed by using the conjunction rule (Section 3.2.4).

Note that, when analyzing a state of the intersection $(q_{t_j}, (q_t, p_i, x), y)$, related to the state q_{t_j} of the replacement and to state (q_t, p_i, x) of the sub-property, we can write validities like $q_{t_j} \models \mu(p_i)$ in place of $q_{t_j} \models \mu(q_t, p_i, x)$ just as we did with the claim, i.e., we can consider only the state of the original claim that concurred to build the sub-property state. This is based on the following assumption.

Assumption. *A sub-property is a tuple* $\bar{\mathcal{S}}_t = \left\langle \bar{\mathcal{P}}_s, \Delta_s^{inP}, \Delta_s^{outP} \right\rangle$ *and the replacement it is checked against is a tuple* $\mathcal{R}_t = \left\langle \mathcal{M}_t, \Delta_t^{inR}, \Delta_t^{outR} \right\rangle$ (see Section 2.3.3). *Our proof corresponds to justifying why the replacement automaton* $\mathcal{M}_t$ *satisfies the formula corresponding to the sub-automaton* $\bar{\mathcal{P}}_t$. *After using the collapsing nodes algorithm, our sub-property automaton* $\bar{\mathcal{P}}_t$ *always has the same shape (or a sub-part of it) and semantics content of the negated claim automaton. The additional information that a sub-property* $\bar{\mathcal{S}}_t$ *holds, aside the sub-automaton, are its incoming and outgoing transitions. The information carried by the in/out-transitions is already dealt with by the refinement checking procedure of Definition 2.17 and is correct.*

Let us clarify this statement with the following observations:

- When a run enters the intersection automaton built between the replacement $\mathcal{R}_t$ for state t and the sub-property $\bar{\mathcal{S}}_t$, the dangerous situations that can arise (and that we would like to avoid) are:

  1. *The run enters the intersection and remains trapped inside of it, cycling infinitely on its states.* We should not worry about this situation: our construction of the sub-proof is indeed only triggered if a previous refinement checking procedure has output *yes* or *possibly-yes*. We do not run into situations where an accepting cycle is possible, because this case is already excluded by the model checking procedure;

  2. *The run starts from a green port and exits the intersection through a red port.* Through a redefined concept of intersection between sub-property and replacement (excluding *red* out-transitions, transforming *yellow* out-transitions into ports, adding *blue* ports), we deal with this situation. All dangerous situations are excluded and only failed runs are considered (where *failed* is referred to the negated claim).

- Our procedure is correct given the correctness of the approach of [MSG15] to check that the sub-property is satisfied by the analyzed replacement.

We formalize these observations as follows:

Theorem 3.1 (Transparent state conclusion)**.** *For every transparent state $t \in T_{\mathcal{M}}$, it holds:*

$$t \models \bigwedge_{j=1}^{n} \mu(p_j) \iff \begin{cases} t \models_P \bigwedge_{j=1}^{n} \mu(p_j) & \wedge \\ \mathcal{M}_t \models \bigwedge_{j=1}^{n} \mu(p_j) & \wedge \\ \forall \; out\text{-}transition \in \Delta_t^{outI} \in \{Yellow, Blue\} \end{cases} \tag{3.1}$$

On the left side, we have a proposition that states that a transparent state semantically entails the conjunction of all formulas valid on the states of the claim. This is equivalent to the conjunction of the three conditions on the right side of the equation. The first conjoined condition represents the information we derive from the master proof. During the construction of the master proof, we collected a series of $t_i \models_P \bigwedge_{j=1}^{n} \mu(p_j)$, one for each combination of $t_i \in T_{\mathcal{M}}$ and claim state $p_j \in Q_{\bar{\Phi}}$. Unfortunately, we were only able to compose a *possibly-validity* referred to state t, waiting for its replacement. Indeed, only now we can reach *sure-validities.* The second conjoined condition basically represents the fact that *a run that enters the replacement does not cycle infinitely inside of $\mathcal{M}_t$.* Proceeding with the analysis of the replacement of t and its behavior against the computed sub-property, we have now information on what is *surely*-valid in it. The third condition expresses the fact that *a run that enters $\bar{\mathcal{S}}_t$, does not exit from a red out-transition.*

3.4 Plugging the sub-proofs into the master proof

At this stage, we know how to build a master proof and also sub-proofs for the initially incomplete states. Now, we would like to put together all the results.

Basically, we would like to show that, while [PZ01] has already described how to prove that a property is satisfied by a fully specified model, we can also prove that the same property can still be satisfied by a model that has not been refined yet. Our proof is incomplete and is finished only when acceptable refinements for the initial model are provided. In Section 3.2, we showed how to build the proof for the initial incomplete model $\mathcal{M}$ expressed as an incomplete Büchi automaton. In Section 3.3, we showed how the general procedure of Section 3.2 can be applied, with some modifications, also to sub-automata representing replacements for transparent states of $\mathcal{M}$. In this section, we would like to propose a methodology to keep track of which parts of the initial proof depend on further refinements of the model (and, iteratively, which parts of a replacement depend on other replacement themselves). Thanks to an appropriate data structure, we can solve dependencies at higher levels (when the model is still not completely refined) with the output obtained at lower levels (completely specified replacements).

We organize the dependency system as a tree structure, whose nodes correspond to the master proof (root) and other sub-proofs. Each node has its

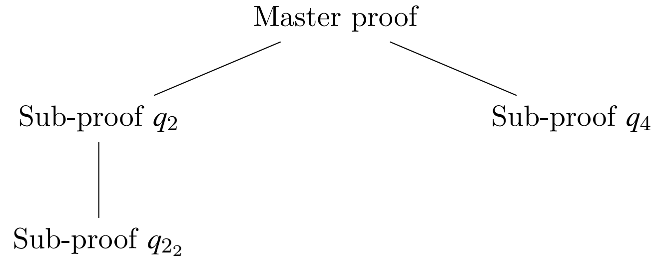


Figure 3.16: Tree of dependencies between proofs

own *depGraph* (see Section 3.2.5), that is a map where the *keys* are the single intersection automaton states, each of which has a *depList* connected, a list of all those states that need to be "solved" before concluding a sure validity about the key.

Let us consider the railway crossing model in Figure 3.2: we would like to replace state q_2 with the replacement in Figure 3.12a, and state q_4 with the replacement in Figure 3.12b. The replacement for q_2 contains a transparent state itself, q_{2_2}. We therefore have a master proof for the whole model, two sub-proofs respectively for states q_2 and q_4, and an additional sub-sub-proof for state q_{2_2}. Figure 3.16 represents the hierarchy to follow when solving the dependencies. Each node has its own *depGraph* structure.

We describe the steps of dependency resolution performed by the procedure of proof plugging once that all replacements are available and the refined model $\mathcal{N}$ of $\mathcal{M}$ is finally complete. Table 3.1, Table 3.2, and Table 3.3 help us to describe the succession of steps using numbered arrows.

First of all, we solve the dependencies of those sub-proofs whose automata do not contain any transparent states anymore. This corresponds to having all dependency lists empty in the *depGraph* related to the considered sub-proof. When all *depList*'s fields of a node of the tree are empty, the sub-proof can be considered closed. According with its conclusion, all its ancestors nodes can be updated.

This is the case of the sub-proof related to q_4 and the one related to q_{2_2} (we assume a replacement for q_{2_2} that satisfies the constraint computed like in [MSG15]) [**Arrows (1)**]. This allows to conclude that $q_{2_2} \models \mu(p_1) \wedge \mu(p_2)$, which means $q_{2_2} \models \phi$ and that $q_4 \models \mu(p_1) \wedge \mu(p_2)$, from which $q_4 \models \phi$. Notice that the conclusion on state q_2 is still *possible* ($q_2 \models_P \phi$), at this stage; instead the sub-proof related to q_4 and the sub-sub-proof related to q_{2_2} provide *sure-conclusions.* This means that we can proceed with substituting all *possible-validities* present in the sub-proof referring to q_2 with *sure-validities* [**Arrows (2)**]. Notice that the dependencies where the key and the content of *depList* is the same are artificial because they are the ones related to a transparent state.

It is the case of $q_{2_2}p_1 \rightarrow q_{2_2}p_1(T)$. Nevertheless, we mark them as dependent anyways to keep track of replaced states. At this point, all *depLists* in the graph of q_2 are empty, therefore, we conclude $q_2 \models \phi$ [**Arrow (3)**].

Master proof ($\mathcal{M}$)							
key	*depList*	Sub-proof (q_2)				Sub-proof (q_4)	
q_5p_1	/	*key*	*depList*	Sub-sub-proof (q_{2_2})		*key*	*depList*
q_4p_2	q_4p_2(T)	q_3p_1	/	*key*	*depList*	$q_{4_1}p_1$	/
q_4p_1	q_4p_2	q_3p_2	/	...	/	$q_{4_2}p_1$	/
q_3p_2	q_4p_2	$q_{2_3}p_1$	/			$q_{4_2}p_1$	/
q_3p_1	q_4p_1	$q_{2_2}p_1$	~~$q_{2_2}p_1$(T)~~				
q_2p_2	q_3p_2	$q_{2_2}p_2$	~~$q_{2_2}p_2$(T)~~				
q_2p_1	q_3p_1,q_2p_2						
		closed $q_2 \models \phi$		closed $q_{2_2} \models \phi$		closed $q_4 \models \phi$	

Table 3.1: Resolution of dependencies. Steps 1-3

We can now use both final conclusions about q_4 [**Arrows (4)**] and q_2 [**Arrow (5)**], to update the master proof graph, by turning *possible-validities* into *sure-validities*. Note that this can be done by updating the entries of the *depGraph* of the master proof by removing any dependency on validities related to q_4 in their *depLists*. [**Arrows (4)**] allow to unlock several dependencies: the keys (q_4p_2), (q_4p_1), (q_3p_2) and (q_3p_1) have now empty *depLists*. [**Arrow (5)**] cancels the dependency of (q_2p_1) on (q_2p_2), while (q_2p_1) still depends on q_3p_1.

Master proof ($\mathcal{M}$)							
key	*depList*	Sub-proof (q_2)				Sub-proof (q_4)	
q_5p_1	/	*key*	*depList*	Sub-sub-proof (q_{2_2})		*key*	*depList*
q_4p_2	~~q_4p_2(T)~~	q_3p_1	/	*key*	*depList*	$q_{4_1}p_1$	/
q_4p_1	~~q_4p_2~~	q_3p_2	/	...	/	$q_{4_2}p_1$	/
q_3p_2	~~q_4p_2~~	$q_{2_3}p_1$	/			$q_{4_2}p_1$	/
q_3p_1	~~q_4p_1~~	$q_{2_2}p_1$	/				
q_2p_2	q_3p_2	$q_{2_2}p_2$	/				
q_2p_1	q_3p_1,q_2p_2						
		closed $q_2 \models \phi$		closed $q_{2_2} \models \phi$		closed $q_4 \models \phi$	

Table 3.2: Resolution of dependencies. Steps 4-5

[**Arrow (6)**] shows how to use the fact that (q_3p_2) is now free from dependencies, to delete the element (q_3p_2) from the *depList* of (q_2, p_2). Finally [**Arrow (7)**] unlocks the last key: also the dependency list of (q_2p_1) is now empty. We can, therefore, conclude that $\mathcal{M} \models \phi$ [**Arrow (8)**].

Master proof ($\mathcal{M}$)							
key	*depList*	Sub-proof (q_2)				Sub-proof (q_4)	
		key	*depList*	Sub-sub-proof (q_{2_2})		*key*	*depList*
				key	*depList*		
q_5p_1	/	q_3p_1	/	...	/	$q_{4_1}p_1$	/
q_4p_2	/	q_3p_2	/			$q_{4_2}p_1$	/
q_4p_1	/	$q_{2_3}p_1$	/			$q_{4_2}p_1$	/
q_3p_2	❻ /	$q_{2_2}p_1$	/				
q_3p_1	/	$q_{2_2}p_2$	/				
q_2p_2	❼ ~~q_3p_2~~						
q_2p_1	~~q_3p_1~~ ❽						
closed $\mathcal{M} \models \phi$		closed $q_2 \models \phi$		closed $q_{2_2} \models \phi$		closed $q_4 \models \phi$	

Table 3.3: Resolution of dependencies. Steps 6-8

4 Tool support: ChIPS

ChIPS, *Checker Initializing Proof Systems*, is a Java module that complements CHIA[1], an existing tool that supports automata-based verification. ChIPS is a prototype stand-alone application realized in Java 7. It is a deductive proof generator initialized by the results of the model checking procedure.

ChIPS has been realized as a proof of concept, with the purpose of showing how software development can benefit from the deductive proof generation method proposed in this thesis. For this reason, we implemented the procedure in charge of computing the master proof described in the first part of our contribution (Section 3.2) and leave the incremental part of sub-proofs (Section 3.3) to future developments. We describe the architecture of the application, that allows to build a proof associated with the intersection automaton created during the model checking procedure between a model and a property that the user would like to check, both described through XML files.

Section 4.1 introduces CHIA, the existing model checking tool on which ChIPS is designed. In Section 4.2, we describe how ChIPS extends CHIA to produce deductive proofs and, in Section 4.3, we describe an example of user interaction with the ChIPS console.

4.1 The CHIA tool

CHIA has been developed by [MSG15] as a proof of concept to validate their research on model checking for incompletely specified systems. It has been developed as a Maven multi-module project. The interaction with the user is managed through a command-line shell which allows to load the model and the claim from XML files, check whether the model satisfies the claim, and save the results of verification on XML output files.

CHIA is composed of different sub-modules. The CHIAAutomata module contains the classes used to manage BAs and IBAs. The CHIAAutomataIO module provides the classes to load and save a BA, IBA and IntersectionBA from and to the appropriate XML files. Finally the CHIAChecker module contains classes that allow to check if a model *satisfies*, *possibly-satisfies*, or *does not satisfy* the properties of interest.

The proof building procedure is triggered after the model checking procedure has been performed and has returned a *yes* or *possibly-yes* output. ChIPS

[1] CHIA: CHecker for Incomplete Automata.

exploits the CHIA architecture to load and save automata from XML files, its intersection procedure, and its model checking result.

4.2 A Checker Initializing Proof Systems: ChIPS

ChIPS uses the infrastructure of CHIA by including the modules from the corresponding repository[2].

ChIPS is launched by executing its Main class, that enables the ChIPS console, a command-line shell to interact with the tool, that is implemented with the support of the Cliche library [cli]. The ChIPS console extends the one of CHIA, and benefits from all its functions. We can load a model that we would like to analyze, the claim to to be considered, perform the model checking procedure, and save the constraints.

The description of the ChIPS module is organized in four sections: Figure 4.2.1 introduces the classes that describe BAs and IBAs, by explicitly modeling their state space. Section 4.2.2 describes the module used to load and save BAs and IBAs, Section 4.2.3 describes the classes which allow to compute the proof, and Section 4.2.4 describes the user interface.

4.2.1 Modeling

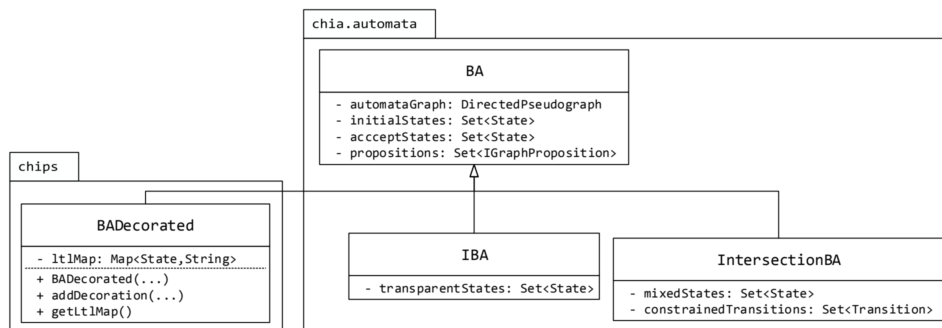


Figure 4.1: The class diagram of the modeling classes

The BA and IBA classes are used to represent claims and models. They are contained in the CHIAAutomata module of the tool CHIA. The BA class describes Büchi automata as graphs, with regular states, initial states, accepting states, and transitions decorated with propositions. In particular, the automaton graph is represented through the DirectedPseudograph class of the JGraphT

[2]The CHIA tool is available at http://home.deib.polimi.it/menghi/Tools/IncModChk.html

library [jgr], that allows to create a directed graph with both *loops* and *multiple edges* between two states.

The BA class is extended by IBA, that contains also transparent states, and IntersectionBA, that provides mixed states and constrained transitions.

The BADecorated class of ChIPS (Figure 4.1) enriches the information held by the automaton states. In particular, in addition to the name and the id of each state, we associated a string representing the LTL formula valid on that state, according to the procedure described in Section 2.2.2. The automaton has an additional attribute, ltlMap, that associates an LTL formula to each state of the claim. The method addDecoration() allows to write this additional information into the automaton.

4.2.2 Input and output

To load and save automata, we exploit the CHIAAutomataIO module. This module provides the classes to manage the input and output of the automata from/to XML files (whose structure is validated against their schema definitions).

The chips.io package contains the BADecoratedReader and ElementToBADecoratedTransformer classes, and the corresponding state transformer ElementToBADecoraedStateTransformer. These classes manage the input of our extended version of BAs for the representation of the claim. The corresponding class diagram is presented in Figure 4.2.

- BADecoratedReader is used to load a BADecorated from an XML file. It extends the corresponding BAReader class (that extends the XMLReader itself) by adding the possibility to read the string attribute that contains the LTL formula for each state of the automaton;
- ElementToBADecoratedTransformer transforms an XML element, which represents a BADecorated, into the corresponding JAVA object. It inherits from the corresponding ElementToBATransformer for regular BAs. This class overrides the method transform() to load states decorated with an LTL formula;
- ElementToBADecoratedStateTransformer transforms an XML element, which represents a BADecorated state, into the corresponding State object. The class extends the ElementToBAStateTransformer.

Listing 4.1 describes the XML input file of the claim represented in Figure 3.3, while Listing 4.2 presents an example of the XML file that corresponds to the railway crossing system model introduced in Figure 3.2.

Each file is composed by three different parts: the tag < propositions > defines the list of all possible < proposition > of the automaton alphabet. Their value is specified in a < value > attribute. < states > delimits the

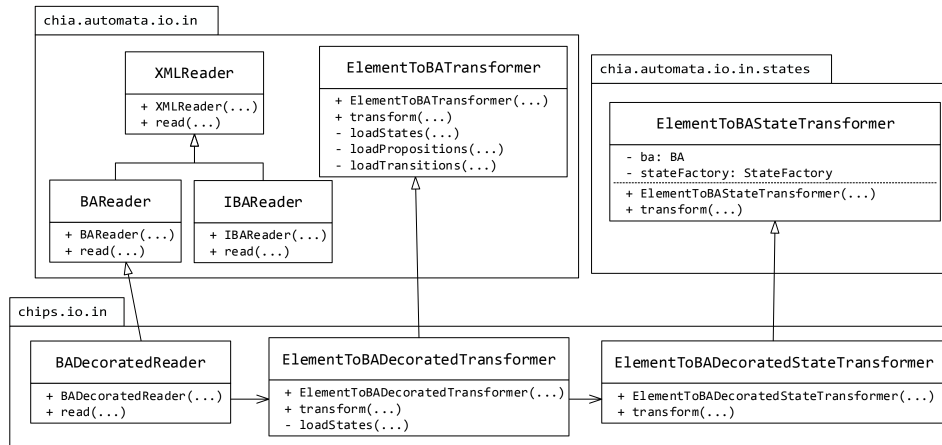


Figure 4.2: The class diagram of the `chips.io.in` package

list of states of the automaton. Each state, denoted by a < `state` > tag, contains the mandatory attributes `id` (unique numeric identifier), `name` and optional boolean attributes like `accepting`, `initial` and `transparent` that are set to `true` in case these characteristics are associated to the given state. The < `transitions` > tag delimits a list of transitions of the automaton. Each transition, specified by using the < `transition` > tag, has the `id`, the `source`, the `destination`, and the `propositions` to describe the source state, the destination state and the propositions that trigger the passage from one state to the other. The `id` fields need to be unique.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ba>
3    <propositions>
4      <proposition value="SIGMA"/>
5      <proposition value="l"/>
6      <proposition value="t"/>
7    </propositions>
8    <states>
9      <state id="1" name="p1" initial="true" accepting="true"
10                             ltl="X((!l)R(!t))"/>
11     <state id="2" name="p2" accepting="true" ltl="(!l)^(!t)"/>
12   </states>
13   <transitions>
14     <transition id="1" source="1" destination="1" propositions="!t"/>
15     <transition id="2" source="1" destination="2" propositions="!l^!t"/>
16     <transition id="3" source="2" destination="2" propositions="SIGMA"/>
17   </transitions>
18 </ba>
```

Listing 4.1: XML file corresponding to the claim BA presented in Figure 3.3

Since ChIPS uses the LTL formula that labels each state in the generation of the proof, we added an attribute to the XML tag referring to the state. These attributes decorate the state with information about the LTL formula valid on it. In the CHIA input file, the tags < state > contain an id (unique numeric identifier), a name, optional boolean attributes like accepting, initial and transparent that are set to true in case these characteristics corresponds to the considered state, and ltlFormula attribute that contains a string representing the formula valid on the considered state. The LTL formula is expressed through the following operators:

Temporal operators: X as "*next*", F as "*eventually*", G as "*always*", U as "*until*", R as "*release*".

Logic operators: ^ as "*and*", || as "*or*", ! as "*negation*", -> and <-> as simple and double *implication*.

In Listing 4.1, we highlight in red the added ltl attribute of state.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <iba>
3    <propositions>
4      <proposition value="l"/>
5      <proposition value="t"/>
6    </propositions>
7    <states>
8      <state id="1" name="q1" initial="true" accepting="true"/>
9      <state id="2" name="q2" accepting="true" transparent="true"/>
10     <state id="3" name="q3" accepting="true"/>
11     <state id="4" name="q4" accepting="true" transparent="true"/>
12     <state id="5" name="q5" accepting="true"/>
13     <state id="6" name="q6" accepting="true"/>
14   </states>
15   <transitions>
16     <transition id="1" source="1" destination="2" propositions="l"/>
17     <transition id="2" source="2" destination="3" propositions="l"/>
18     <transition id="3" source="3" destination="4" propositions="l"/>
19     <transition id="4" source="4" destination="5" propositions="t"/>
20     <transition id="5" source="5" destination="6" propositions="t"/>
21     <transition id="6" source="6" destination="5" propositions="l^t"/>
22   </transitions>
23 </iba>
```

Listing 4.2: XML file for the model IBA presented in Figure 3.2

The XML file containing the IBA has the same structure as the one used in CHIA. It contains the < state > tag and attribute of a BA input file, with the exception that a state can have an attribute transparent and that the transitions cannot contain negated propositions.

We use classes from CHIAAutomataIO that manage the output of the intersection automaton written on a suitable XML file.

In addition to these, we designed two classes to print our output in text files:

- ProofWriter is used to print the proof associated with a model and a claim previously loaded into a file in the specified path;

- ▶ `DepGraphWriter` is used to print the graph that contains the dependencies between the validities of the proof, into a file in the specified path.

Figure 4.3 contains a UML class diagram that describes the structure and relations of the described classes with the `CHIA` tool. All three *writers* are called during the procedure that performs model checking and then builds a deductive proof (triggered by the method `checkAndBuildProof` in `ChIPSAutomataConsole`).

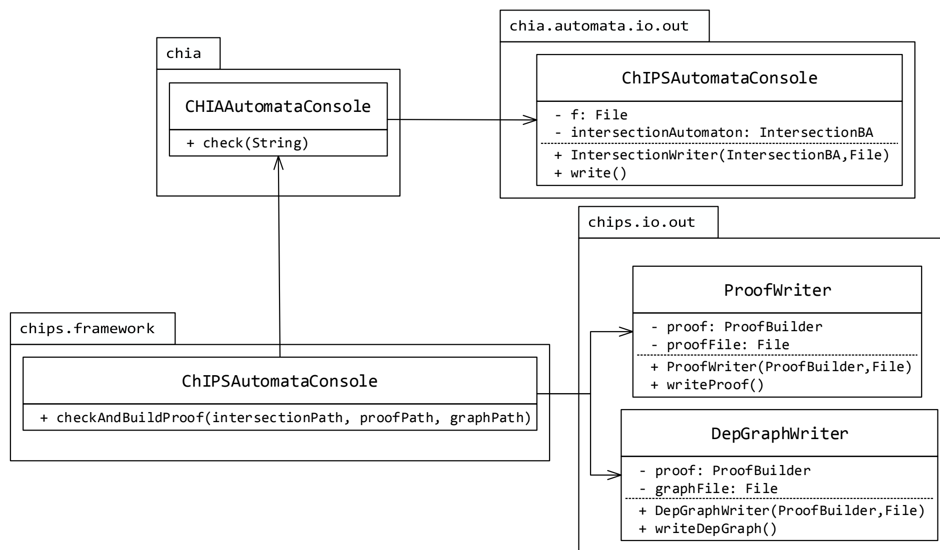


Figure 4.3: The class diagram of the `chips.io.out` package

4.2.3 Building the proof

The main package of our module is `chips.prover`, that contains all the elements to gather the necessary information to build the rules of the proof. The rules are created and managed through the classes of package `chips.rule`. The specific lines of the proofs, the *rows*, are dealt with by package `chips.row`.

Figure 4.4 presents the UML class diagram of the `chips.prover` package, that contains the classes involved in the generation of the proof.

- ▶ `ProofBuilder` contains the entry point used to run the proof construction. It requires an `IntersectionBA` as input. It has an instance of `IntersectionBuilderProver` that provides all the methods useful to deal with the intersection automaton, an instance of `SCCDealer` that

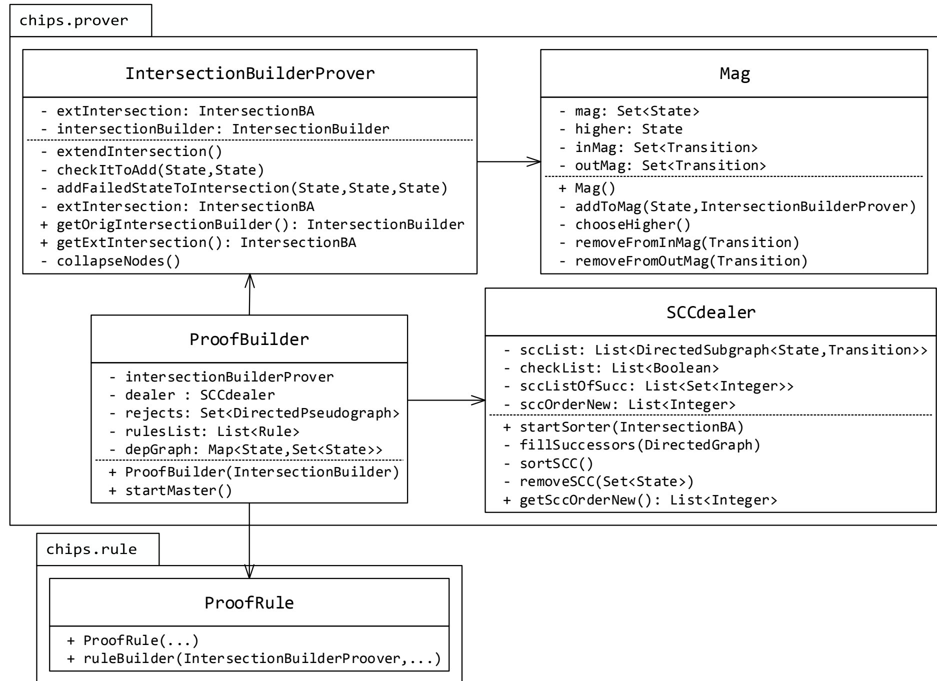


Figure 4.4: The class diagram of the chips.prover package

analyzes SCCs and sorts them, a set of rejected SCCs (rejects), a list of rules (rulesList), and map that records validities dependencies (depGraph). This class orchestrates the creation of the proof through the method startMaster(), that implements Algorithm 3.2 (BuildProof);

- IntersectionBuilderProver extends the IntersectionBuilder of the CHIAChecker module (that computes the intersection between a model IBA and a claim, BA), by providing several methods that allow to manipulate the graph and extend it as needed (see Section 3.2.1);

- SCCdealer manages the identification of SCCs and their sorting with the method startSorter() that uses the private methods fillSuccessors() and sortScc();

- Mag is a class functional to the procedure of nodes collapsing described in Section 3.2.1. A Mag is a macro state representing all the intersection automaton states deriving from the same model and claim states. This kind of object is used in the method collapseNodes of the IntersectionBuilderProver.

In Figure 4.4, we also represented the `ProofRule` class from package `chips.rule` to show that, after the `ProofBuilder` has predisposed everything to start computing the proof, it calls the constructor of the rule, `ruleBuilder()`.

Figure 4.5 presents the UML class diagram associated with the `chips.rule` package.

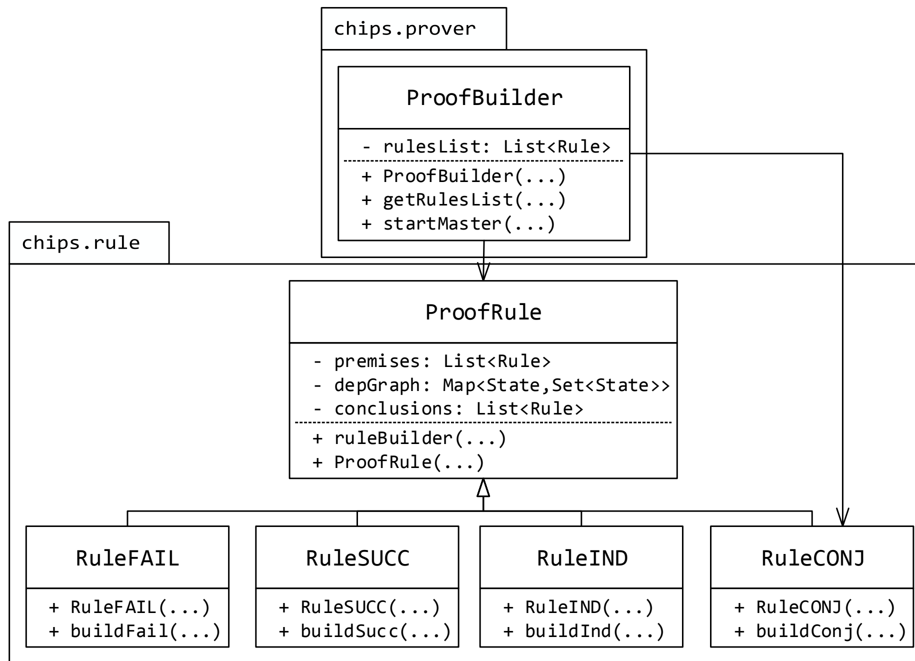


Figure 4.5: The class diagram of the `chips.rule` package

- `ProofRule` is an abstract class that describes a rule. It contains the attributes `premises`, `conclusions` and an instance of `depGraph`. It is extended by four different deductive rules. `ruleBuilder()` is a *static factory method* that implements the logic of Algorithm 3.5, by choosing to build a different rule based on the characteristics of the analyzed SCC;
- `RuleFAIL` implements the logic described in Algorithm 3.6;
- `RuleSUCC` implements the logic described in Algorithm 3.7;
- `RuleIND` implements the logic described in Algorithm 3.8;
- `RuleCONJ` implements the logic described in Algorithm 3.9.

Figure 4.6 presents a UML class diagram of the classes contained in the chips.row package. It is a package that only worries about correctly formatting the gathered information into each proof row. For organization purposes we choose to use a unique parent class with a toString() method that each kind of inheriting instance overrides for its purpose.

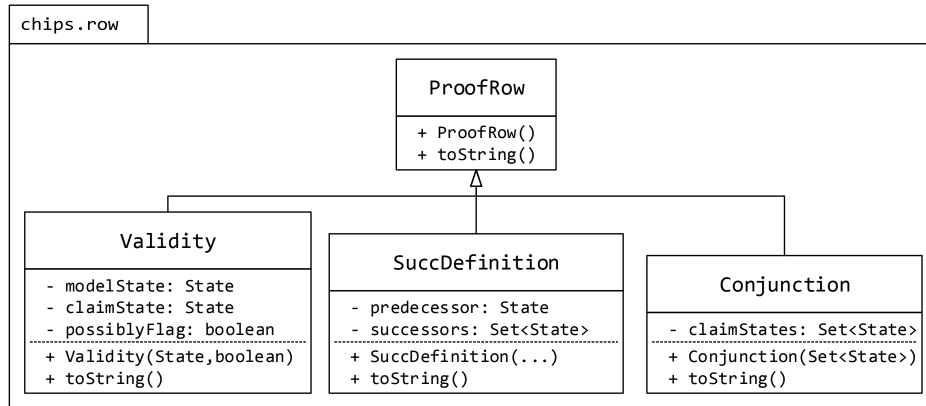


Figure 4.6: The class diagram of the chips.row package

- ProofRow is the parent class: all possible rows composing the proof derive from it;
- Validity allows to build a *sure-validity* (Definition 3.6) or a *possible-validity* (Definition 3.7), depending on the value of the *possiblyFlag*;
- SuccDefinition allows to express the definition of a state's successors (as specified in Definition 3.8);
- Conjunction is a particular row that only appears in the premise of the Conjunction rule. It basically allows to state that the conjunction of all the LTL formulae that decorate the states of the claim automaton implies the initial requested property.

4.2.4 Initial framework

The chips.framework package contains the classes used to run ChIPS. The main method in Main launches the command-line shell ChHIPSConsole, that extends the CHIAConsole. Two working modes are available: by typing "aut" we access the ChIPSAutomataConsole, that allows to start the procedure to build the master proof (Section 3.2), whilst with "rep" we access the ChIPSReplacementConsole, that deals with the construction of sub-proofs.

By extending `CHIAAutomataConsole`, the `ChIPS` console inherits all the methods that execute commands to load the model, the claim, and perform the model checking procedure. This, in particular, is managed through the module `CHIAChecker`, that allows to check if the loaded model `IBA` *satisfies*, *does not satisfy* or *possibly-satisfies* the loaded claim `BA` (or `BADecorated`, in our case). In addition to the inherited methods, our console implements:

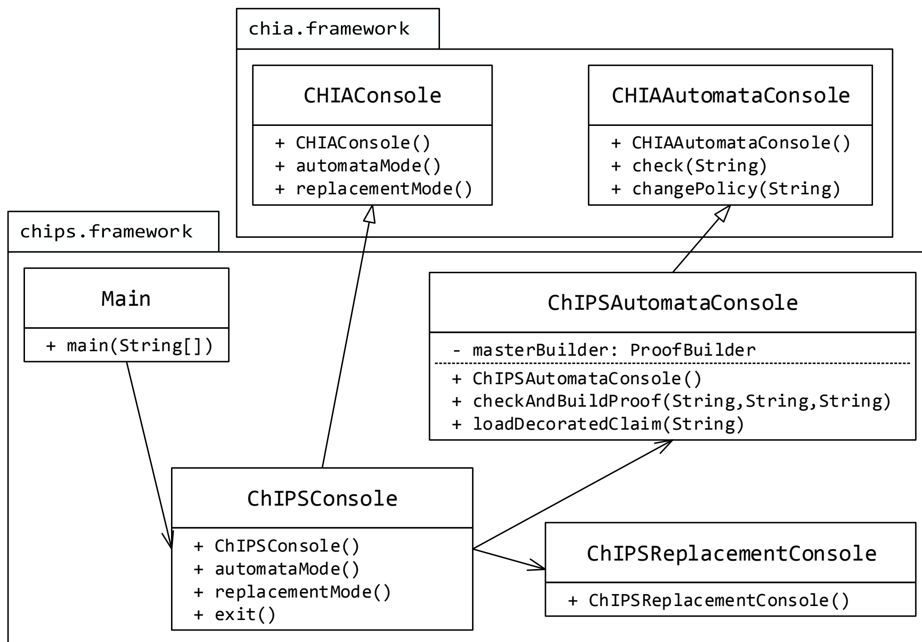


Figure 4.7: The class diagram of the `chips.framework` package

- `loadDecoratedClaim()` allows the user to load an `XML` file that corresponds to an enriched `BA`, a decorated claim automaton;
- `checkAndBuildProof()` is the central method of our approach. It first calls the method `check()` from its parent class that computes the model checking procedure. Then, depending on the satisfaction value result (i.e., if it is different from "*notsatisfied*"), the procedure that builds the master proof is started. At the end of it, both the proof and the dependency-graph are printed out.

4.3 Interaction with the tool

We present an example of the interaction the user can have with `ChIPS`. After running `ChIPS` the user can choose between the automata and the replacement mode.

```
ChIPS> aut
```

The `aut` command allows the user to enter the *automata mode*, opposed to the command `rep`, that allows to access the *replacement mode*.

```
ChIPSmaster> cp NORMAL
```

The `cp` command activates the method `changePolicy`. It allows the user to switch between a `KRIPKE` policy (set by default) and a `NORMAL` policy.

```
ChIPSmaster> lm <modelFilePath>
```

The `lm` commands stands for `loadModel` and allows the user to load the `IBA` that represents the system he/she wants to analyze by specifying the path of the `XML` file defined as showed in Listing 4.2.

```
ChIPSmaster> lcd <claimFilePath>
```

The `ldc` command stands for `loadDecoratedClaim`; specifically it loads the `XML` file in the specified path (like, for example, the one in Listing 4.1) as the `DecoratedBA` that represents the claim with LTL formulae as states attributes.

```
ChIPSmaster> ckbp <intersectionFilePath>
                  <proofFilePath> <graphFilePath>
```

The `ckbp` command stands for `checkAndBuildProof`. This is the central command to activate our procedure. It performs model checking using `CHIA`'s modules and, when the result is *yes* or *possibly-yes*, uses the result to build the proof. The user must specify the path of three files to be created or overwritten where the tool can output the structure of the intersection automaton used, the content of the deductive proof, and the dependency-graph structure.

```
ChIPSmaster> exit
```

`exit` allows the user to exit the mode he/she has entered. If used from `ChIPS` >, it exits the console.

5 Case study

In this chapter, we analyze an academic case study to evaluate the applicability of the proposed approach. The case study has been considered in [MSG15, AY01] and represents a communication protocol, in charge of sending a message. An initial, high level and incomplete model of the system $\mathcal{M}$ is represented in Figure 5.1 as an incomplete Büchi automaton (see Definition 2.6). When the system is started, the automaton moves from the initial state to the first transparent module, $send_1$, which represents a function performing the first attempt to send a message. If the first try succeeds, the final state q_3 is entered. If the first attempt is not successful, the system activates a second function, $send_2$, that tries to send the message again, for example by using a different communication device. If this attempt succeeds, the accepting state q_3 is reached, otherwise the abort state q_2 is reached. The system can move from one state to another using the limited set of actions $\{start, ok, fail, success, abort\}$ that define its alphabet.

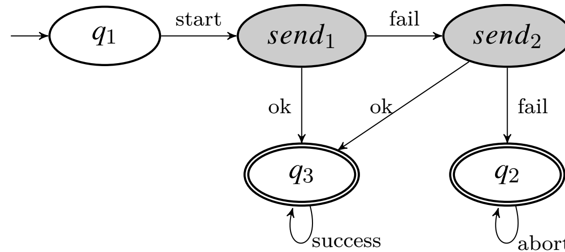


Figure 5.1: Model of the system in charge of sending a message

The developer might want to know whether this initial design satisfies the requirements of the system. Consider, for example, the liveness property "whenever a message is sent, then it will eventually reach the receiver" formalized in linear temporal logic as $\phi = \Box(send \rightarrow \Diamond success)$. The developer may want to know if it is satisfied in its incomplete design.

The *incomplete model checking* procedure helps to find out if the model of this system satisfies or possibly satisfies ϕ. The model checking procedure intersects the IBA representing the model $\mathcal{M}$ with the BA representing the negation of the property ϕ. In Figure 5.2, we show the BA $\bar{\Phi}$ equivalent to the LTL formula $\neg\Box(send \rightarrow \Diamond success)$.

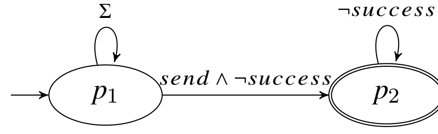


Figure 5.2: Automaton representing the negated claim

The deductive proof exploits the LTL formulae associated with each state of the claim. As explained in Section 2.2.2, these formulae are obtained from the tableau associated with $\neg\phi$, which is represented in Figure 5.3. Note that su is a shortcut for the proposition $success$ and se for $send$. According to Proposition 2.1 from [PZ01], we can express the sub-formulae, on the states of $\bar{\Phi}$, as $\eta(p_1)$ and $\eta(p_2)$.

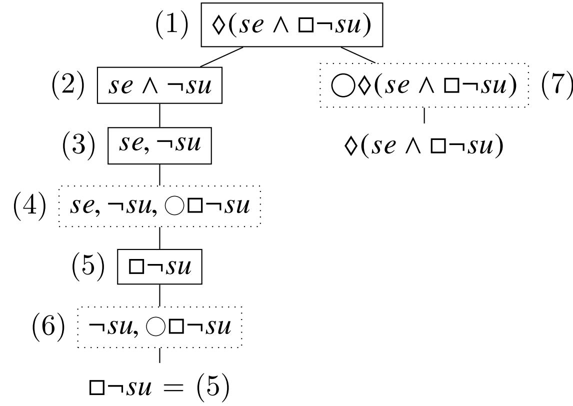


Figure 5.3: LTL tableau for $\neg\Box(send \rightarrow \Diamond success)$

- $\eta(p_1) = \bigcirc\Diamond(send \land \Box\neg success)$ - It is derived by following the right branch of the tableau and by applying the rules of Table 2.1. Below node **(7)**, we find a formula that already exists in the same branch: it indeed corresponds to the one on the root **(1)**. Therefore, we return the formula of the last numbered node of the tree, **(7)**, as representative of state p_1. Note that this formula is assigned to the initial node of the BA $\bar{\Phi}$ since **(1)** has no incoming nodes. For Proposition 2.1, $\mu(p_1) = \bigcirc\Box(\neg send \lor \Diamond success)$.

- $\eta(p_2) = \Box\neg success \land send$ - It is derived by following the left branch of the tree. After applying a *dynamic rule* (with the “next” symbol) to node **(6)**, we find the $\Box\neg success$ formula, that is already present in node **(5)**. We therefore perform a conjunction of the formulae held in the parents of the nodes with the repeated formula. Node **(4)**, parent of **(5)**, contains $\{send, \neg success, \bigcirc\Box\neg success\}$ and node **(6)**, parent of the leaf of the branch, contains $\{\neg success, \bigcirc\Box\neg success\}$. By conjoining these two formulae, we obtain that $\eta(p_2) = \bigcirc\Box\neg success \land \neg success \land send$. According to the first *fixed-point* equation in 2.1, we derive $\eta(p_2) = (\bigcirc\Box\neg success \land \neg success) \land send = \Box\neg success \land send$. For Proposition 2.1, $\mu(p_2) = \neg send \lor \Diamond success$.

In the next sections, we simulate a scenario where *incomplete model checking* ([MSG15]) is applied to the just described model and claim automata, followed by the construction of the master proof (see Section 5.1). Then, we assume the initial system refined using two example replacements for the transparent states, that are used, along with the constraint computed like in [MSG15], to perform a procedure of replacement checking ([MSG15]) that allows to build two specific sub-proofs (Section 5.2). At each refinement step, the newly computed sub-proof allows to solve the dependencies related to it (see Section 5.3).

5.1 Master proof building

Let us consider the model and claim automata introduced in the last section. The incomplete model checking procedure verifies if the model of this system already satisfies or if it possibly satisfies ϕ. In the second case, the model checking procedure return a $\perp$ value (see Definition2.14), meaning that $\mathcal{M}$ *possibly satisfies* ϕ. In this case, the developer may use the presented procedure to compute a *master proof* that shows *why* the model does not contain any accepting run that satisfies the negated property, but contains possibly accepting runs that could violate the property.

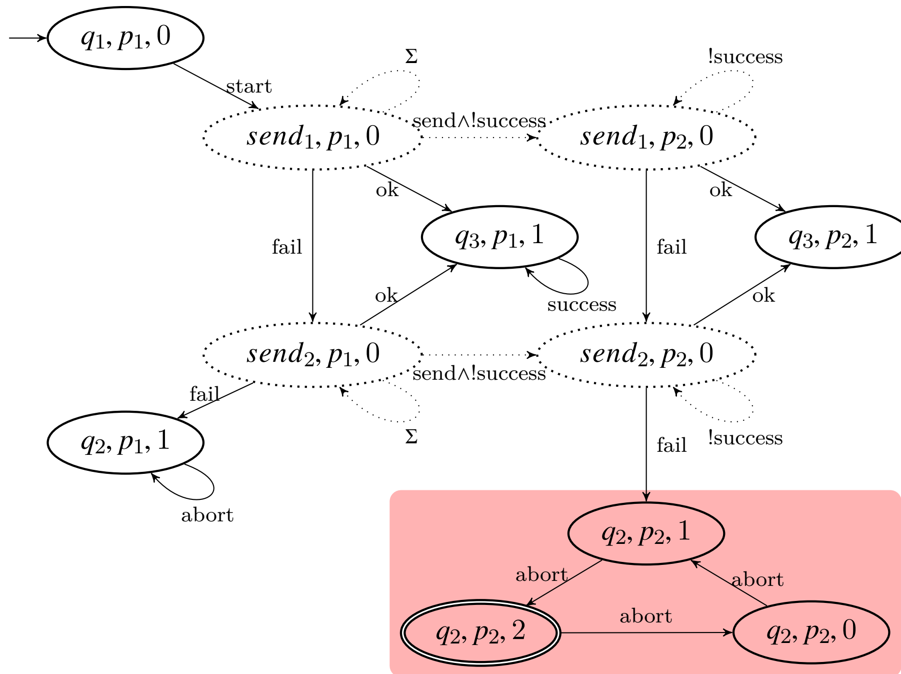


Figure 5.4: Intersection automaton for the sending message example

In Figure 5.4, we represent the intersection automaton computed between the model $\mathcal{M}$ and the negated automaton $\bar{\Phi}$, associated with the property ϕ. The states marked with a dotted border (*mixed states*) and the transitions identified with dotted lines (*constrained transitions*) are obtained by combining transparent states of the model and transitions executed in their replacements with states and transitions of the claim. The runs that involve these states and transitions are *possible* runs since they depend on how transparent states are refined.

Figure 5.4 also shows, on a red background, a cycle among the accepting states that, if reached, determines a violation of the property. The three states involved represent a dangerous component, that is rejected from the graph during the procedure of proof building, as explained in Section 3.2.3.

As specified in the Paragraph "Nodes collapsing" of Section 3.2.1, the first step of the proof generation concerns the collapsing of nodes that derive from the same model and claim states. In the case study under analysis, the procedure collapses $(q_2, p_2, 0)$, $(q_2, p_2, 1)$ and $(q_2, p_2, 2)$ into the node (q_2, p_2) and converts the transitions among these three nodes into a self-loop insisting on the new one. For all other states of the automaton, only the first two components of their labels are memorized.

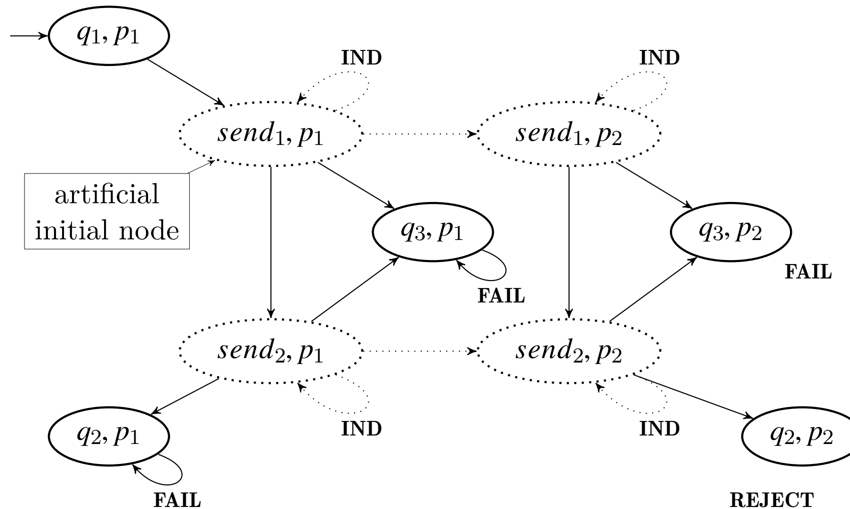


Figure 5.5: Graph analyzed for the rule generation

Furthermore, the algorithm extends the intersection automaton with failed states (Paragraph "Failed" of Section 3.2.1). This step does not have any effect on the current intersection since all possible combinations between a state of the model and a state of the claim have already been included in the original intersection.

The graph depicted in Figure 5.5 represents the simplified intersection au-

tomaton. This version of the automaton is analyzed to identify all strongly connected components and to establish a *partial order relation* $\prec$ among them, as specified in Definition 3.5. $C \prec C'$ means that if there is an edge from a node in C to a node in C', it is necessary to consider C' before C. This order assures that, each time a component is analyzed, all the SCCs that are reachable from it, have been already processed.

The SCCs are identified by the Tarjan's([Tar72]) algorithm and are later sorted according to the mentioned partial order: (q_3, p_2), (q_2, p_1), (q_3, p_1), (q_2, p_2), $(send_2, p_2)$, $(send_2, p_1)$, $(send_1, p_2)$, $(send_1, p_1)$.

After the succession of SCCs has been computed, each of them is processed to identify the kind of correctness assertion that can be derived from it. In this particular case, all SCCs are composed by a single node. We remind the reader to make reference to Figure 3.9 and Figure 3.10 that should help distinguish between cases.

Since (q_3, p_2) is a regular, non-accepting, failed node (Figure 3.9g), the RuleFail is triggered. The same rule is applied to the states (q_2, p_1) (situation of Figure 3.9e) and (q_3, p_1) (Figure 3.9g), since in both cases the states are regular, non-accepting, with no successors except for themselves. Differently, (q_2, p_2) is a non-trivial SCC composed by a single accepting node. As specified in Figure 3.9a, this condition leads to a dangerous situation. The component is rejected from the proof, and so is the transition from $(send_2, p_2)$ leading to it, since any safe replacement of the transparent states would not allow any run to reach (q_2, p_2). Note that, in case the refined system did not comply with the specification, the procedure of proof building would have not started. For this reason, the only interesting components for the proof, are the ones that form an empty intersection automaton. Finally, $(send_2, p_2)$, $(send_2, p_1)$, $(send_1, p_2)$, $(send_1, p_1)$ are all mixed nodes that present both a self-loop and successors: RuleInd is applied according to Figure 3.10b. Notice that in Figure 5.5 the rules to be applied are indicated on the side of each component.

The sorted list of SCCs is used to build the rules that produce the master proof. Listing 5.1 schematizes the deductive reasoning that proves that the model in Figure 5.1 *possibly-satisfies* the requirement $\phi = \Box(send \rightarrow \Diamond\neg success)$. At a later stage, when the transparent states get refined, the procedure progressively builds dedicated sub-proofs that eventually solve the dependencies on hold. Indeed, we note that certain rules present conclusions that are final, i.e., *sure-validities* (see Definition 3.6), as for example the three RuleFail at the beginning of the listing. Others, instead, present conclusions that are not final yet, i.e., *possible-validities* (see Definition 3.7) that depend on the resolution of the uncertainty of rules built on earlier processed SCCs. For example, this is the case of the three RuleInd applied to the components $(send_2, p_2)$, $(send_2, p_1)$, $(send_1, p_2)$, $(send_1, p_1)$.

Note that the validities in the conclusion of a rule are *possible* if the same

rule presents at least a *possible*-premise, or if the state upon which the validity is built is *mixed*. A validity in the premise is *possible* if there exists a previous rule, showing the same validity as a *possible*-conclusion. We remind that the symbol $\models_P$ marks the *possible-validities* present in both premise and conclusion of rules.

1. Using the RuleFail axiom on the node (q_3, p_2), we obtain $q_3 \models \mu(p_2) = \neg send \vee \Diamond success$
2. Using the RuleFail axiom on the node (q_2, p_1), we obtain $q_2 \models \mu(p_1) = \bigcirc\Box(\neg send \vee \Diamond success)$
3. Using the RuleFail axiom on the node (q_3, p_1), we obtain $q_3 \models \mu(p_1) = \bigcirc\Box(\neg send \vee \Diamond success)$
4. Node (q_2, p_2) is rejected.
5. Applying the RuleInd to the SCC = $(send_2, p_2)$, where Exit(SCC) = $\{(q_3, p_2)\}$, we obtain:

   $$\frac{\begin{array}{ll} send_2 \to \{q_3\} & \\ q_3 \models \mu(p_2) & = \neg send \vee \Diamond success \end{array}}{send_2 \models_P \mu(p_2) \quad = \neg send \vee \Diamond success}$$

   Applying line 1 as a premise to line 5, we obtain $send_2 \models_P \neg send \vee \Diamond success$
6. Applying the RuleInd to the SCC = $(send_1, p_2)$, where Exit(SCC) = $\{(send_2, p_2), (q_3, p_2)\}$, we obtain:

   $$\frac{\begin{array}{ll} send_1 \to \{send_2, q_3\} & \\ q_3 \models \mu(p_2) & = \neg send \vee \Diamond success \\ send_2 \models_P \mu(p_2) & = \neg send \vee \Diamond success \end{array}}{send_1 \models_P \mu(p_2) \quad = \neg send \vee \Diamond success}$$

   Applying lines 1, 5 as a premise to line 6, we obtain $send_1 \models_P \neg send \vee \Diamond success$
7. Applying the RuleInd to the SCC = $(send_2, p_1)$, where Exit(SCC) = $\{(send_2, p_2), (q_2, p_1), (q_3, p_1)\}$, we obtain:

   $$\frac{\begin{array}{ll} send_2 \to \{q_2, q_3\} & \\ q_2 \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ q_3 \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ send_2 \models_P \mu(p_2) & = \neg send \vee \Diamond success \end{array}}{send_2 \models_P \mu(p_1) \quad = \bigcirc\Box(\neg send \vee \Diamond success)}$$

Applying line 1, 2, 5 as premises to line 7, we obtain
$send_2 \models_P \bigcirc\Box(\neg send \vee \Diamond success)$

8. Applying the RuleInd to the SCC = $(send_1, p_1)$,
where Exit(SCC) = $\{(send_1, p_2), (send_2, p_1), (q_3, p_1)\}$, we obtain:

$$\begin{array}{ll} send_1 \to \{send_2, q_3\} & \\ send_1 \models_P \mu(p_2) & = \neg send \vee \Diamond success \\ send_2 \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ q_3 \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ \hline send_1 \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \end{array}$$

Applying line 3, 6, 7 as premises to line 8, we obtain
$send_1 \models_P \bigcirc\Box(\neg send \vee \Diamond success)$

9. Using rule CONJ, we obtain:

$$\begin{array}{ll} send_1 \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ send_1 \models_P \mu(p_2) & = \neg send \vee \Diamond success \\ \mu(p_1) \wedge \mu(p_1) \to \phi & = \Box(\neg send \vee \Diamond success) \\ \hline send_1 \models_P \phi & = \Box(\neg send \vee \Diamond success) \end{array}$$

Applying line 6, 8 as premises to line 9, we obtain
$send_1 \models_P \Box(\neg send \vee \Diamond success) = \phi$

Listing 5.1: Deductive proof of $\mathcal{M} \models \phi$ for the sending message system

During the application of RuleConj, note that $send_1$ has been chosen as the *artificial initial node*, instead of the nominal initial node q_1, representative of all model $\mathcal{M}$ automaton, because it is the first reachable node that is not semantically empty (as specified in Definition 3.9), whereas the nominal initial node of this automaton q_1 does not contain any information, since it has no incoming transitions.

The *possible-conclusion* $send_1 \models_P \mu(\phi)$ of the final rule means that the master proof is not complete.

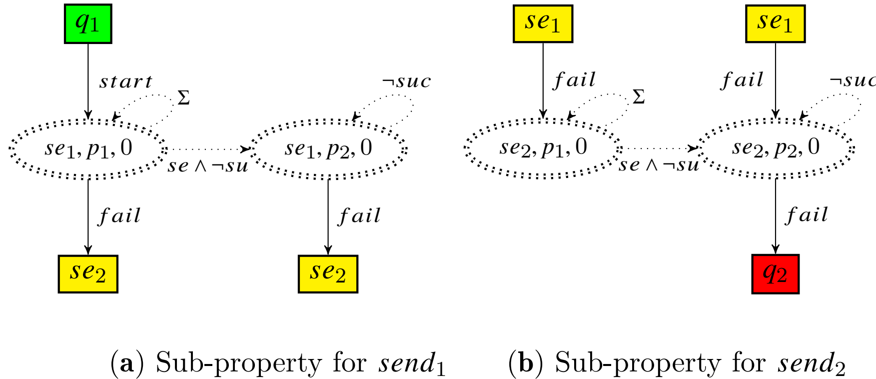
5.2 Computing the sub-proofs

When the developer refines a transparent state, he/she can consequently trigger the computation of its dedicated sub-proof and then update the master proof by solving the lines that depend on the refinement of the considered

transparent state. The sub-proof computation procedure exploits the sub-property and the replacement for a specific transparent state.

A sub-property indicates those behaviors that should be forbidden to any replacement of the considered state, in order for the whole system to work properly. In particular, any run leading to a red outgoing transition must be avoided because it corresponds to a behavior of the system that violates the requirement. The runs leading to yellow outgoing transitions, moreover, should be *possibly* avoided, since they help reaching dangerous areas of the intersection automaton. Nevertheless, for the purposes of this proof, they are treated as perfectly admissible runs.

Figure 5.6a-b presents the sub-properties associated with the transparent states $send_1$ and $send_2$, computed according to [MSG15]. The one associated with the replacement of $send_1$ indicates that any run that passes through this state by entering the incoming green transition q_1 and exiting through one of the outgoing yellow transitions $send_2$ (marked with the abbreviation se_2) is a run that *possibly* violates the claim. The term *possibly* indicates that we are not already violating ϕ, but we are also not guaranteeing that the run will not reach a red outgoing transition, i.e., a violating run, in the transparent states reached after the one under analysis. The sub-property for the replacement of $send_2$, instead, specifies that any run entering the state $send_2$ through a yellow in-transition $send_1$ and exiting through the red out-transition q_2, is possibly violating (since we are not sure that the yellow incoming port is reachable).



(**a**) Sub-property for $send_1$ (**b**) Sub-property for $send_2$

Figure 5.6: Sub-properties for the sending message example

At this stage, the question is whether it is possible or not to refine the system (i.e., to find a suitable replacement for $send_1$ and $send_2$) in such a way that the model $\mathcal{M}$ not only *possibly-satisfies*, but finally *satisfies* the required property $\phi = \Box(send \rightarrow \Diamond\neg success)$.

Figure 5.7 represents the proposed replacement for state $send_1$. It specifies that, by firing the transition coming from q_1 labeled with $start$, it is possible

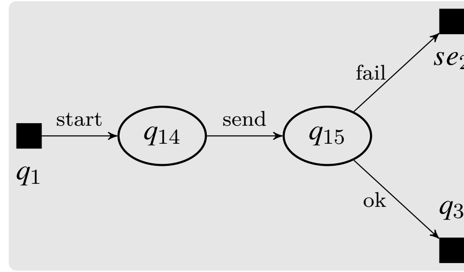


Figure 5.7: Replacement for state $send_1$

to reach the state q_{14}. The automaton $\mathcal{M}_{send_1}$, by *sending* a message, moves from q_{14} to q_{15}. State q_{15} is connected to the outgoing transition $send_2$ labeled with the proposition $fail$, and the outgoing transition q_3 labeled with ok.

The framework uses this replacement in the sub-proof computation. An intersection automaton between this and the sub-property of Figure 5.6a is computed as $\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1}$. As done for the computation of the master proof, the sub-proofs computation requires to modify the intersection structure computed with respect to the definition of [MSG15] (Definition 2.18).

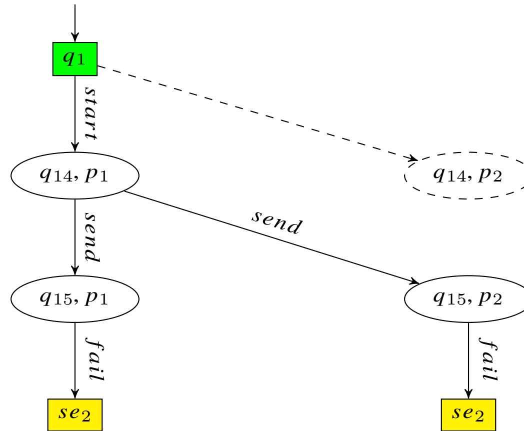


Figure 5.8: First extention step for automaton $\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1}$

More precisely, it is necessary to collapse the nodes with the same first two components (in this case study this phase has no impact), and add the failed nodes, configurations of the replacement that fail to satisfy the sub-property, that did not originally belong to the intersection computed in [MSG15] (see Paragraph "Failed" of Section 3.2.1). Here, the failed state (q_{14}, p_2) and the failed transition with empty label $\langle q_1, \{\}, (q_{14}, p_2)\rangle$ are added.. Figure 5.8 shows the intersection automaton $\mathcal{I}_{send_1} = (\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1})$ after this step has been

applied.

Then, the procedure considers the need of adding *blue ports* (see Definition 3.11) to the intersection structure. Blue ports complete the information about all the possible exits from the replacement automaton than a run can take. In this case, an out-transition that is not considered by the sub-property (because not dangerous) is q_3. The procedure, therefore, adds two out-transitions $\langle (q_{15}, p_1), \{\}, q_3 \rangle$ and $\langle (q_{15}, p_2), \{\}, q_3 \rangle$, which correspond to two out-ports q_3, marked with color *blue*.

Finally, the procedure transforms the out-ports into states, by replicating them for all the claim states that could be reached from that point. We explicitly specify the combination between the model state of the ports and the state of the sub-property which they are intersected with. Notice that all states derived from the expansion of out-ports are failed nodes, since an out-port lacks successors, by definition.

Figure 5.9 describes the state of the intersection automaton after these two steps have been performed. First the blue port q_3 has been added. Later, both the yellow out-port $send_2$ and the blue q_3 ports have been split into states representing the intersection of the model state of the port and a reachable claim state.

We consider the incoming port as the real initial node of the intersection automaton and, therefore, the nodes directly reached by it, (q_{14}, p_1) and (q_{14}, p_2), as artificial initial nodes (see Definition 3.9) to be used in the proof.

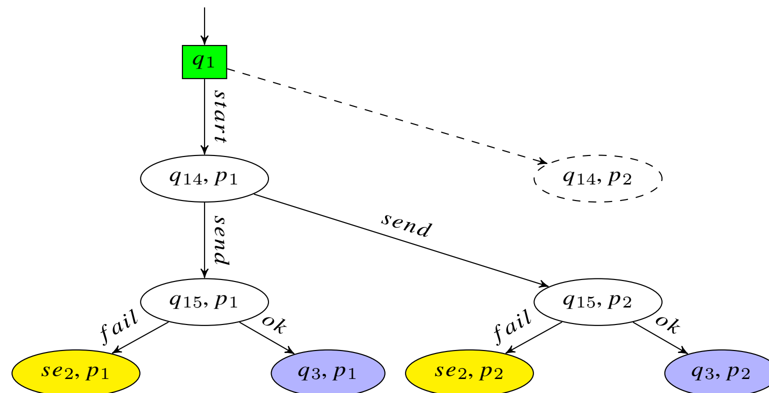


Figure 5.9: Second extention step for automaton $\mathcal{R}_{send_1} \cap \bar{\mathcal{S}}_{send_1}$

A quick observation of this simple graph shows that all nodes correspond to trivial SCCs. The computation of the *partial order* relation specified in Definition 3.5, returns the following relations:

$(q_{14}, p_1) \prec (q_{15}, p_1), (q_{15}, p_2)$;

$(q_{15}, p_1) \prec (send_2, p_1), (q_3, p_1)$;

$(q_{15}, p_2) \prec (send_2, p_2), (q_3, p_2)$.

It is therefore possible to choose the following order to process the SCCs: $(q_{14}, p_2), (q_3, p_2), (q_3, p_1), (send_2, p_2), (send_2, p_1), (q_{15}, p_2), (q_{15}, p_1), (q_{14}, p_1)$

The sub-property computation procedure does not require to reject any component. Since all the components are non-accepting, non-mixed single states, the RuleFail is applied to the components that have no successors (it is the case of Figure 3.9g) and RuleSucc to the ones with successors (Figure 3.9h).

Listing 5.2 presents the sub-proof computed from the automaton previously described.

- **Nodes $(\mathbf{q_{14}}, \mathbf{p_2}), (\mathbf{q_3}, \mathbf{p_1}), (\mathbf{q_3}, \mathbf{p_2}), (\mathbf{send_2}, \mathbf{p_1}), (\mathbf{send_2}, \mathbf{p_2})$. RuleFail.**
  $q_{14} \models \mu(p_2)$, $q_3 \models \mu(p_1)$, $q_3 \models \mu(p_2)$, $send_2 \models_P \mu(p_1)$, $send_2 \models_P \mu(p_2)$
- **SCC = $(\mathbf{q_{15}}, \mathbf{p_1})$, Exit(SCC) = $\{(\mathbf{send_2}, \mathbf{p_1}), (\mathbf{q_3}, \mathbf{p_1})\}$. RuleSucc.**

$$\frac{\begin{array}{ll} q_{15} \rightarrow \{send_2, q_3\} & \\ q_3 \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ send_2 \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \end{array}}{q_{15} \models_P \mu(p_1) \quad \bigcirc\Box(\neg send \vee \Diamond success)}$$

- **SCC = $(\mathbf{q_{15}}, \mathbf{p_2})$. Exit(SCC) = $\{(\mathbf{send_2}, \mathbf{p_2}), (\mathbf{q_3}, \mathbf{p_2})\}$. RuleSucc.**

$$\frac{\begin{array}{ll} q_{15} \rightarrow \{send_2, q_3\} & \\ q_3 \models \mu(p_2) & = \neg send \vee \Diamond success \\ send_2 \models_P \mu(p_2) & = \neg send \vee \Diamond success \end{array}}{q_{15} \models_P \mu(p_2) \quad = \neg send \vee \Diamond success}$$

- **SCC = $(\mathbf{q_{14}}, \mathbf{p_1})$, Exit(SCC) = $\{(\mathbf{q_{15}}, \mathbf{p_1}), (\mathbf{q_{15}}, \mathbf{p_2})\}$. RuleSucc.**

$$\frac{\begin{array}{ll} q_{14} \rightarrow \{q_{15}\} & \\ q_{15} \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ q_{15} \models_P \mu(p_2) & = \neg send \vee \Diamond success \end{array}}{q_{14} \models_P \mu(p_1) \quad = \bigcirc\Box(\neg send \vee \Diamond success)}$$

- **RuleConj.**

$$\frac{\begin{array}{ll} q_{14} \models_P \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\ q_{14} \models_P \mu(p_2) & = \neg send \vee \Diamond success \\ \mu(p_1) \wedge \mu(p_2) \rightarrow \phi & = \Diamond(send \wedge \Box\neg success) \end{array}}{q_{14} \models_P \phi \quad = \Diamond(send \wedge \Box\neg success)}$$

Listing 5.2: Deductive sub-proof of $send_1 \models \phi$

The conclusion $q_{14} \models_P \phi$ is equivalent to say $send_1 \models_P \phi$. The replacement is complete, but the out-port $send_2$ is still a transparent state, therefore the conclusion is still *possible* and not *sure* yet. Briefly, the validities derived from the first bullet line of Listing 5.2 represents the conclusions derived on the out-ports and the failed node (q_{14}, p_2). It can be observed that they all represents sure-validities with the exception of the two related to $send_2$. These trivial statements are used to derive the conclusions of the following rules. Notice that whenever the premise of a rule contain a possible validity, the conclusion is marked as possible too. RULECONJ is based on the artificial initial node q_{14} directly reached by the in-port q_1.

After the sub-property presented in Figure 5.6b has been computed, the replacement in Figure 5.10 is proposed for the transparent state $send_2$. State q_{16} is the destination of the in-transition $\langle send_1, fail, q_{16}\rangle$. By *sending* a message, the automaton moves from q_{16} to q_{17}. After *waiting* for one transition (reaching q_{18}), the system either moves towards q_{19} by *acknowledging* the data transmission, or moves towards q_{20} if the *timeout* interval has passed. Both states guarantee a *retry* action that leads back to q_{16}. State q_{21} corresponds to a failure state that is not reachable from the other states of the replacement. q_{19} is the source of the out-transition $\langle q_{19}, ok, q_3\rangle$ and q_{21} is the source of the out-transition $\langle q_{21}, fail, q_2\rangle$.

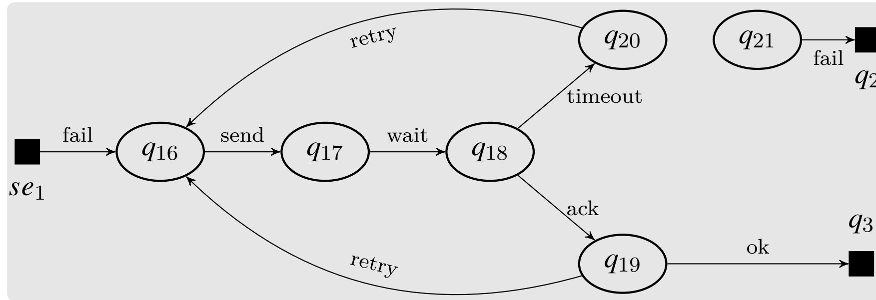


Figure 5.10: Replacement for state $send_2$

The replacement of $send_2$ is checked against the related sub-property in Figure 5.6b. Figure 5.11 represents the intersection computed between them. With respect to the intersection output by the replacement checking procedure in [MSG15], we applied various modifications. First, the nodes with different third component {0,1,2}, but deriving from the same model and claim states, have been collapsed into a unique one. Since all possible combinations between the states of the replacement and the ones of the sub-property already appeared in the intersection, there were no failed nodes to be added. The blue port q_3 was added. Finally, it was replicated for the two claim states.

Notice that the red port q_3 belonging to the sub-property of $send_2$ can never be reached.

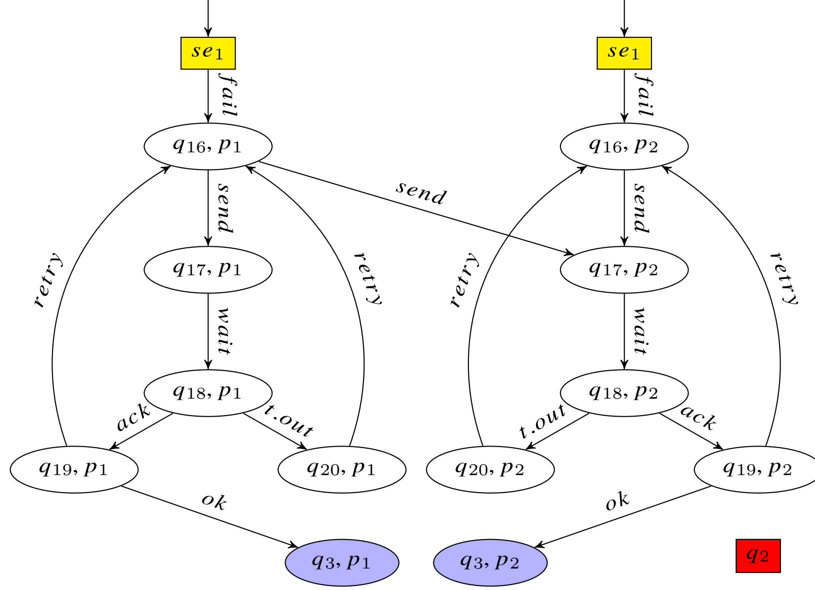


Figure 5.11: Extention of intersection automaton $\mathcal{R}_{send_2} \cap \bar{\mathcal{S}}_{send_2}$

The proposed procedure requires to identify the strongly connected components of the graph and to sort them according to the *partial order* relation of Definition 3.5. Through Tarjan's SCCs search algorithm we find two trivial components, $(q_3, p_2), (q_3, p_1)$ and two non-trivial ones, $SCC_1 = \{(q_{16}, c_{2,1}), (q_{17}, c_{2,1}), (q_{18}, c_{2,1}), (q_{19}, c_{2,1}), (q_{20}, c_{2,1})\}$ and $SCC_2 = \{(q_{16}, c_{2,2}), (q_{17}, c_{2,2}), (q_{18}, c_{2,2}), (q_{19}, c_{2,2}), (q_{20}, c_{2,2})\}$.

A suitable processing order can, therefore, be $(q_3, p_2), (q_3, p_1), SCC_2, SCC_1$. By analyzing these SCCs and applying a rule to each one, the sub-proof in Listing 5.3 is obtained.

- **Nodes** $(\mathbf{q_3}, \mathbf{c_{2,1}}), (\mathbf{q_3}, \mathbf{c_{2,2}})$. **RuleFail.** $q_3 \models \mu(p_1)$, $q_3 \models \mu(p_2)$
- $\mathbf{SCC_1} = \{(\mathbf{q_{16}}, \mathbf{p_1}), (\mathbf{q_{17}}, \mathbf{p_1}), (\mathbf{q_{18}}, \mathbf{p_1}), (\mathbf{q_{19}}, \mathbf{p_1}), (\mathbf{q_{20}}, \mathbf{p_1})\}$, $\mathbf{Exit(SCC_1)} = \{(\mathbf{q_{17}}, \mathbf{p_2}), (\mathbf{q_3}, \mathbf{p_1})\}$. **RuleInd.**

$$
\begin{aligned}
&q_3 \models \mu(p_1) && = \bigcirc\Box(\neg send \vee \Diamond success)\\
&q_{17} \models \mu(p_2) && = \neg send \vee \Diamond success\\
&q_{16} \rightarrow \{q_{17}\}\\
&q_{17} \rightarrow \{q_{18}\}\\
&q_{18} \rightarrow \{q_{19}, q_{20}\}\\
&q_{19} \rightarrow \{q_{16}, q_3\}\\
&q_{20} \rightarrow \{q_{16}\}
\end{aligned}
$$

$$\begin{array}{ll}
\hline
q_{16} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\
q_{17} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\
q_{18} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\
q_{19} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\
q_{20} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success)
\end{array}$$

- **SCC** $= \{(\mathbf{q_{16}}, \mathbf{c_{2,2}}), (\mathbf{q_{17}}, \mathbf{c_{2,2}}), (\mathbf{q_{18}}, \mathbf{c_{2,2}}), (\mathbf{q_{19}}, \mathbf{c_{2,2}}), (\mathbf{q_{20}}, \mathbf{c_{2,2}})\}$, **Exit(SCC)** $= \{(\mathbf{q_3}, \mathbf{c_{2,2}})\}$. **RuleInd.**

$$\begin{array}{ll}
q_3 \models \mu(p_2) & = \neg send \vee \Diamond success \\
q_{16} \rightarrow \{q_{17}\} & \\
q_{17} \rightarrow \{q_{18}\} & \\
q_{18} \rightarrow \{q_{19}, q_{20}\} & \\
q_{19} \rightarrow \{q_{16}, q_3\} & \\
q_{20} \rightarrow \{q_{16}\} & \\
\hline
q_{16} \models \mu(p_2) & = \neg send \vee \Diamond success \\
q_{17} \models \mu(p_2) & = \neg send \vee \Diamond success \\
q_{18} \models \mu(p_2) & = \neg send \vee \Diamond success \\
q_{19} \models \mu(p_2) & = \neg send \vee \Diamond success \\
q_{20} \models \mu(p_2) & = \neg send \vee \Diamond success
\end{array}$$

- **RuleConj.**

$$\begin{array}{ll}
q_{16} \models \mu(p_1) & = \bigcirc\Box(\neg send \vee \Diamond success) \\
q_{16} \models \mu(p_2) & = \neg send \vee \Diamond success \\
\mu(p_1) \wedge \mu(p_2) \rightarrow \phi & = \Box(\neg send \vee \Diamond success) \\
\hline
q_{16} \models \phi & = \Box(\neg send \vee \Diamond success)
\end{array}$$

Listing 5.3: Deductive sub-proof of $send_2 \models \phi$

Similarly to the sub-proof for the transparent state $send_1$, the conclusion $q_{16} \models \phi$ is equivalent to say $send_2 \models \phi$, being q_{16} the artificial initial node of the replacement. Since the replacement is complete, i.e., it does not contain any transparent state, the proof is complete too. All the validities presented are indeed *sure-validities*.

5.3 Plugging the sub-proofs into the master proof

Notice that the sub-proof just computed for $send_2$ is final, i.e., it does not contain *possible-validities*. The derived conclusion, $send_2 \models \phi$ can, therefore, be considered *sure*. On the other hand, the proof related to $send_1$ is not complete because, when computed, its result depended on the replacement of $send_2$. This means it is possible to proceed and use the results of the second

sub-proof to update the ones of first sub-proof, along with the ones of the master proof.

Table 5.1 represents the *depGraph* structure that describes the dependencies between master proof and its sub-proofs, before starting the plugging procedure. It is immediate to see that the *depList* corresponding to the sub-proofs for $send_2$ is empty (therefore, the proof is marked as closed). The *depList* usually contains, for each key specified on the left (a specific SCC of the intersection graph) the list of which components the key depends on. More practically, to the component of the second replacement $q_{16}p_1$, corresponds a *sure-validity* in the proof, i.e., its decision does not depend on solving other *possible-validities.* Instead, if $q_{16}p_1$ had depended on another component, it would have been necessary to wait until the component in its *depList* was sure, in order to proceed and mark $q_{16}p_1$ as sure too.

Let us have a look at the first two columns: the *key* $send_1p_1$ depends on the components $send_1p_2$ and $send_2p_1$. The symbol (T) indicates that the *key* of that line refers to a transparent state itself. When *sure* conclusions are derived about $send_1$ and $send_2$ (this is our case, since the columns of the sub-proofs for $send_1$ and $send_2$ are marked as *closed*), we can easily see that all elements in the *depList* of the master can be eliminated. When a *depList* is empty, the conclusion on the bottom can be declared *sure.*

Master proof ($\mathcal{M}$)					
key	*depList*	Sub-proof ($send_1$)		Sub-proof ($send_2$)	
$send_1p_1$	$send_1p_2$,$send_2p_1$(T)	*key*	*depList*	*key*	*depList*
$send_1p_2$	$send_2p_2$(T)	$q_{14}p_1$	$q_{15}p_1$,$q_{15}p_2$	$q_{16}p_1$	/
q_3p_1	/	$q_{14}p_2$	/	...	...
q_3p_2	/	$q_{15}p_1$	$send_2p_1$	$q_{20}p_1$	/
$send_2p_1$	$send_2p_2$(T)	$q_{15}p_2$	$send_2p_2$	$q_{16}p_2$	/
$send_2p_2$	(T)	$send_2p_1$	(T)	...	...
q_2p_1	/	$send_2p_2$	(T)	$q_{20}p_2$	/
		q_3p_1	/	q_3p_1	/
		q_3p_2	/	q_3p_2	/
				closed ($send_2$) $q_{16} \models \phi$	

Table 5.1: Resolution of dependencies - step 1

In this case study, after the replacement proposed for $send_2$ has been substituted with the transparent state, we can remove from the *depLists* of the master proof and the sub-proof for $send_1$ all the occurrences of components related to $send_2$. This step is represented in Table 5.2. Now also the *depLists* related to the sub-proof of $send_1$ are empty and the proof is closed.

Master proof ($\mathcal{M}$)					
key	*depList*	Sub-proof ($send_1$)		Sub-proof ($send_2$)	
$send_1 p_1$	$send_1 p_2$, ~~$send_2 p_1$~~(T)	*key*	*depList*	*key*	*depList*
$send_1 p_2$	~~$send_2 p_2$(T)~~	$q_{14} p_1$	~~$q_{15}, p_1, q_{15} p_2$~~	$q_{16} p_1$	/
$q_3 p_1$	/	$q_{14} p_2$	/	...	...
$q_3 p_2$	/	$q_{15} p_1$	~~$send_2 p_1$~~	$q_{20} p_1$	/
$send_2 p_1$	~~$send_2 p_2$(T)~~	$q_{15} p_2$	~~$send_2 p_2$~~	$q_{16} p_2$	/
$send_2 p_2$	~~(T)~~	$send_2 p_1$	~~(T)~~	...	...
$q_2 p_1$	/	$send_2 p_2$	~~(T)~~	$q_{20} p_2$	/
		$q_3 p_1$	/	$q_3 p_1$	/
		$q_3 p_2$	/	$q_3 p_2$	/
		closed ($send_1$) $q_{14} \models \phi$		closed ($send_2$) $q_{16} \models \phi$	

Table 5.2: Resolution of dependencies - step 2

Finally, also the occurrences related to $send_1$ can be eliminated. We empty the last dependency that was left in the master proof (see Table 5.3), and can declare the master proof closed too.

Master proof ($\mathcal{M}$)					
key	*depList*	Sub-proof ($send_1$)		Sub-proof ($send_2$)	
$send_1 p_1$	~~$send_1 p_2$(T)~~	*key*	*depList*	*key*	*depList*
$send_1 p_2$	/	$q_{14} p_1$	/	$q_{16} p_1$	/
$q_3 p_1$	/	$q_{14} p_2$	/	...	...
$q_3 p_2$	/	$q_{15} p_1$	/	$q_{20} p_1$	/
$send_2 p_1$	/	$q_{15} p_2$	/	$q_{16} p_2$	/
$send_2 p_2$	/	$send_2 p_1$	/	...	...
$q_2 p_1$	/	$send_2 p_2$	/	$q_{20} p_2$	/
		$q_3 p_1$	/	$q_3 p_1$	/
		$q_3 p_2$	/	$q_3 p_2$	/
closed ($\mathcal{M}$) $send_1 \models \phi$		closed ($send_1$) $q_{14} \models \phi$		closed ($send_2$) $q_{16} \models \phi$	

Table 5.3: Resolution of dependencies - step 3

The obtained proof is composed by a master proof related to $\mathcal{M}$, containing the initial skeleton of the final proof, and by two sub-proofs corresponding to the transparent states of the initial model $send_1$ and $send_2$. Since the two proposed replacements are completely specified, no additional refinement round is needed. By using the dependency graph to keep track of dependencies and delete the ones solved, the proof can be declared complete when all *depLists* are empty and each column referring to a (sub)proof is marked as “closed”.

6 State of the Art

Incremental methods of verification are designed to support the current agile development processes. *Ad hoc* formalisms have been defined to iteratively specify the model of the system. Verification and its techniques are profoundly dependent on the specific frameworks in which the analyzed problem is considered [GPB02]. This means that the solutions found to verify if the properties of interest holds, are various in different contexts. It is very difficult to find a universally recognized *better practice* in this area. In Section 6.1 we present the state of the art on the different formalisms used to express incompleteness in the modeling process. When the model and the properties of interest have been formalized, the verification methods need a renovation that includes an enrichment of the procedure to support a greater number of situations. Section 6.2 presents the state of the art on the model checking approaches proposed in literature. Finally, Section 6.3 presents an overview on the main approaches that have combined the model checking technique with features derived from deductive verification to offer a more complete verification output, similarly to the result this thesis aims to obtain.

6.1 Modeling incomplete systems

Different formalisms to represent incompleteness of systems have been proposed over time. Each of them is associated to a particular refinement process dependent on the characteristics of the formalism itself.

A specific notation engineered to express specifications is represented by Hierarchical State Machines (HSMs, [AY01]). This formalism includes ordinary states and *superstates* that are HSMs themselves. The entry state is a unique ordinary state, whilst exit states can be more than one. Entry and exit states connect the single HSM to the ones at higher or lower levels. Two are the advantages of HSMs: first, the possibility to specify systems in a step-wise refinement way using *superstates* and their ability to specify modules ad different levels of detail; second, the possibility to replace multiple *superstates* of the machine with the same specified HSM. Note that the refinement process of HSMs is applied by connecting a superstate of a HSM to a replacement of it (which is another HSM).

Another formalism considered to represent systems is Labelled Transition Systems (LTS). Since LTSs are not suitable to express incompleteness features and distinguish between different levels of granularity, Larsen and Thomsen

[LT88] extended them through Modal Transition Systems (MTSs). Differently from LTSs that only include one kind of transitions, MTSs present necessary and admissible transitions (also called *possible* by [CBFU07]). Several notions of refinements for MTSs have been proposed in literature, as indicated in [UABD+13].

Similarly to an MTS, a Kripke MTS (KMTS, [HJS00, SG04]) divides *necessary* and *possible* behaviors of a system by using two kinds of transitions sets: must ($\xrightarrow{must}$) and maybe ($\xrightarrow{maybe}$) transitions. The set of maybe transitions is included in the set of must transitions. An KMTS considers abstract states of the system $\mathcal{M}$ as representatives of a set of concrete states of the system $\mathcal{M}'$ which refines $\mathcal{M}$. The main difference with the MTS formalism, is that here states are labeled instead of transitions. An abstract state is labeled with atomic propositions that are satisfied or not satisfied by all the concrete states of $\mathcal{M}'$. Some propositions on the label of a state of $\mathcal{M}$ may be left unspecified and assigned only when the $\mathcal{M}$ is refined is refined with $\mathcal{M}'$. Notice that [HJS00, SG04] have presented solutions for the refinement process of KMTSs.

LTSs are considered in [GPB02] as systems that have been set up in an unknown environment. The interaction between the environment and the LTS is triggered through actions that label the transition of the LTS. The *interface operator* $\uparrow$ specifies the set of the actions $\mathcal{A}$ of the model which are observable from the environment. This operator describes how the model interacts with this environment. The refinement step corresponds to the specification of the environment in which the model is executed.

[SS13] has introduced a particular variation of LTS, where the set of states is partitioned into *regular* states and *transparent* states, special states that can represent more complex components still unknown. This is similar to what is done my [MSG15] in the context of Büchi automata. These systems are called Incomplete Labelled Transition Systems (ILTS).

A particular kind of Statecharts ([Har87]) has been presented in [GMSS13, GMSS14]. Classical Statecharts are a structured graphical formalism used to describe reactive systems. *Evolving Statecharts* can be considered as incomplete hierarchical Statecharts, that support step-wise specification. Their hierarchical architecture is what makes them appropriate for incremental modeling. An algorithm to transform Statecharts into the equivalent ILTS ([SS13]) is presented in [GMSS13] to allow the verification process.

6.2 Model checking and incompleteness

Model checking was born following two different general approaches. The first was independently developed by Clarke and Emerson [CE82] in the United States and by Queille and Sifakis [QS82] in France. It was introduced under the name of *temporal model checking* mainly because specifications are here

expressed in temporal logic, according to the definition of Pnueli [Pnu77]. Systems are here described as finite state transition systems and the procedure consists of checking if the system is a *model* for the given specification. The second approach, instead, uses a specification formalized through an automaton. The system automaton is compared to the automaton representing the desired specification to establish if the first's behavior is conform to the second one's. Literature presents different versions of *conformity*, among which we cite language inclusions introduced in [HK90] (the one that probably became more successful), refinement orderings [CPS93], and observational equivalence [FGK+96]. Vardi and Wolper ([VW86]) showed how the two approaches could be related, expressing the temporal model checking in terms of automata.

Model checking has been used in contexts where the model to be checked was incomplete at first, and later refined in a step-wise manner. The checking techniques are obviously strictly dependent on the modeling formalism used and on the specification expression. The claim is usually formalized as an LTL formula, a CTL formula (see Section 2.2) or directly as an automaton.

[AY01] considers the model checking of Hierarchical State Machines with respect to both LTL and CTL properties. As far as the procedure to verify CTL formulae is concerned, usually after the refinement process, a HSM is converted into a flat Finite State Machine (expanded structure), by recursively substituting each box of the structure with the corresponding FSM. In [AY01], though, this step is avoided, allowing the complexity of the original algorithm to decrease exponentially. As to specifications given as automata, given a HSM K and an automaton A which may be obtained from an LTL formula, the model checking problem is to solve the automaton-emptiness problem, i.e., to check whether $\mathcal{L}(A) \cap K^F$ is empty, being K^F an expanded version of K.

[UBC09] introduces the concept of *safety properties* and *scenarios* used to synthesize MTSs that represent the *upper* and *lower bounds*, respectively, on the behaviors of a system. Safety properties therefore include all the possible behaviors the system can exhibit, and scenarios include less behaviors than the ones the final model should present. Model checking is performed by merging the MTS that represents the safety property with the MTS that represents the scenario. This procedure returns the MTS equivalent to their least common refinement. From the analysis of the obtained system, it is possible to infer if the scenario is satisfied, possibly satisfied or not satisfied in the model synthesized from the property.

The model checking on systems specified as Kripke MTS was mainly addressed using CTL to specify the property ([CDEG03]).

As we mentioned in Section 6.1, Giannakopoulou et al. ([GPB02]) considers system models specified with LTSs extended by an additional interface operator $\uparrow$, that defines the interaction of the model with its unknown environment. The model checking procedure consists in verifying that the model combined with the environment does satisfy the claim also specified in terms

of a (deterministic) LTS. The traditional approach that returns yes/no answer is modified to obtain *yes* (when the model satisfies the claim in all the possible environment), *no* (when the the model violates the claim in all environments), and *maybe* (in the other cases).

[SS13] proposes a model checking algorithm to be used with ILTSs. In addition to verifying if the requirements hold, their procedure outputs a set of constraints for the unspecified components, if necessary. After the components have been specified, the verification can be performed in an isolated way only between the new components and the constraints, similarly to what is done in [MSG15].

6.3 Combining model checking and deductive verification

Each formal method was developed and applied with different intents and within different contexts. Nevertheless, experiments from literature ([CW96]) have demonstrated that often the integration of different formal methods to work towards a same goal may allow adding up their strengths, while alleviating some of their weaknesses. Many works witness there is no such a distinction anymore between model checking based approaches and theorem-proving based approaches. Several contributions have in fact considered a combination of these techniques. To quote from [TC02]: "Traditionally, model checkers have been viewed as decision procedures that return yes/no answers reflecting the "correctness" of the system being analyzed." With the development of techniques that could exploit model checking results, many have agreed that the idea of using proofs to provide information that justifies the result can be of great interest to the users of model checkers. Most of these works, like [PZ01] and [PPZ01], have focused on cases where finding a counter-example is not feasible. Others works, like [TC02] and [GC03], have explored the complimentary approach, by enriching the verification output in case the model checking procedure returned a negative answer.

In general, the interest has been dedicated to find ways to certify the correct behavior of software, be this represented as programming code or as an abstract model of it. [HNJ+02] and [KV04], among others, argue for the need of a *proof certificate*, that confirms the correctness of a successful model checking run. This is achieved by using the potentiality of deductive verification to provide intuition that justifies why the program works.

In particular, [KV04] accompanies a positive answer to the model checking query with a certificate whose correctness can be checked automatically and symbolically. Differently from [PZ01] and our work, it does not present a deductive proof that is useful to the user to evaluate design choices, but a certificate to be verified automatically. Their idea is profoundly supported

by the need of compact solutions for the verification of large systems, while [PZ01] guarantees a better support with smaller examples.

[Nec97], instead, introduces the *proof-carrying code* (PCC), an interesting example of certifying the behavior of a program that is not trusted. Their work is used by [HNJ+02] to prove safety temporal properties through small correctness deductions, by manipulating the results of model checking through a technique of lazy abstraction that supplies annotations. [Nec97] and [HNJ+02] both work on code rather than on a separately constructed abstract model of it. In addition, their local checks are performed at the level of the edges of the analyzed graph, similarly to what we propose and differently from [PZ01], that employs transition labeled models.

As already mentioned, [GC03] presents another approach that makes use of the information gathered by a model checker but the goal is exactly the opposite of the other mentioned works. This time the annotations are provided in case of counter-example generation. Their motivation is driven by the excessive conciseness of a small counterexample given in terms of states and transitions of the model, therefore bound to the modeling formalism used. Even though with different aims, the witness generation for ACTL (a subset of universally-quantified CTL) that they propose is similar to the concept that [PZ01] and our work use for generating proofs of satisfaction for LTL properties.

Several other works have devised proof systems for model checkers. These enrichment of the model checking procedure is encoded in different formulations through different works in literature. Among the contributions that deal with result certification it is worth highlighting some.

[Nam01] developed, in parallel with [PZ01] and [PPZ01], a proof system and algorithm for *symbolic* representation, in particular for the μ-calculus. Other works that have faced this issue in the context of μ-calculus are [Kic] and [YL97]. [TC02] extended Namjoshi's work in the case of local model-checking, presenting a special data structure, the *support sets*, to generate "diagnostic-information generation" and "justification generation". Model checking results ([CGMZ95, Sti95]) are used both to explain why a Kripke structure fails to satisfy a temporal property, and to return a portion of the system, *witness*, that is responsible for the property being satisfied.

Other attempts to combine model checking with deductive verification are presented in [JS94, RSS95], where the result of model checking is accepted as an axiom by the theorem prover. To overcome the limit of model checking to only treat finite state systems, [RSS95] attempts to verify the finite parts of complex systems automatically to narrow the state space that needs to be analyzed deductively. The same idea is followed by [YL97, Spr98] using *explicit state model checkers* and generating tableau proofs.

All these approaches decorate the model checking result with pieces of information and annotations that are obviously strictly connected to the modeling

formalism and, therefore, the architecture of the model checker used. Nevertheless, we observe two common issues:

- All the mentioned methods [PZ01], [PPZ01], [Nam01], [HNJ+02], [Nec97], and [TC02] describe various ideas to instrument tools to produce formal proofs of the model checking verdict. Nonetheless, a long run still has to be covered before the contribution of the verification tools can be reused in the certification process, which is the future challenge to be faced;
- [Nam03] arises a legitimate issue by observing that using deductive verification on abstracted models still fails to describe the missing link from the abstract program leading back to the concrete one: justifications for both positive and negative answer of the model checking query, are therefore not always really meaningful for the purposes of the designer.

We finally mention a few examples in literature that have contributed to develop *Compositional verification*, a technique based on breaking up the verification of a system into smaller tasks that involve performing the verification of its components separately, and then combining the proofs. Works that have implemented this idea are [LT88], that presents a compositional correctness proof, [McM99], where theorem proving has been used to prove conditions for sound compositional reasoning, [FMS98] that performs deductive verification on modular systems, [LSW95] that shows a methodology to build a state-based proof which is oriented to the formalization of constraints for concurrent software systems. This procedure aims at reducing the dimension of the verification problem under investigation, by exploiting compositionality and abstraction aspects.

7 Conclusions

Software pervades every aspects of our modern lives. While the software development process has nowadays reached maturity in many aspects, its correctness can never be achieved in its entirety. Fortunately, verification methods supply a substantial help in locating the errors in the designs and in the code.

Regrettably, software verification is still often seen as a heavy burden that slows down the production process. The hard work of the formal methods engineers is to introduce and integrate new software reliability techniques and tools that respond to the market demand. Flexibility and modularity are definitely two of the most important features that successful solutions should present.

In literature, several contributions to software verification have dealt with incompleteness in its various forms, promoting incremental perspectives and modular approaches. An interesting number of works has instead addressed the issue of combining different techniques to resolve diverse aspects of verification in an integrated way.

This thesis has weighed the advantages and drawbacks of model checking and deductive verification techniques, and has suggested a novel approach to perform deductive verification on incompletely specified systems by exploiting the results obtained through model checking.

This chapter presents the final considerations about our procedure of incremental deductive proofs construction. Section 7.1 describes the contribution of our methodology together with its limits, while Section 7.2 discusses some possible future works.

7.1 Contributions and limits

This thesis has analyzed the features of two different formal verification techniques and evaluated the benefits that their use can bring during the design phase. Within this context, we may summarize our contributions as follows:

- We considered the incremental model checking approach presented in [MSG15]; their procedure offers an infrastructure to model incompletely specified systems and reason on them to analyze required properties an their satisfiability. In our work, we exploited their structures and results to automatically feed a procedure of deductive proof generation;

- We considered the approach described in [PZ01, PPZ01], that integrates model checking with the possibility to generate deductive systems that describe the computed intersection automaton. Our idea was to modify this work to proceed in a modular way, by taking into consideration an initially incomplete model that is refined step by step;

- By combining the two mentioned works, this thesis extrapolated an integrated approach that follows the developer from the beginning to the end of the design phase according to the following scenarios:

  - When a complete (or incomplete) model is considered against a *satisfied* requirement, our approach allows to prove why this happens, by deriving the requirement as a theorem of the model;

  - When an incomplete model is considered against a *possibly-satisfied* requirement, our approach builds a deductive proof that is incomplete itself and, to be completed, requires information coming from the model refinement;

  - When an incomplete model is refined by substituting the incomplete modules with suitable replacements (checked against the appropriate constraint), we are able to build *ad hoc* proofs for only the considered new portions of the model.

- We provide a methodology to link results related to sub-parts of the model to the ones related to the whole model. Our procedure uses a data structure that contains information about the hierarchy that relates the computed proofs and the dependencies among each other. Proofs associated to lower level components contribute to complete the ones associated to the higher level modules.

- We validated our approach through the realization of a prototype tool that receives, as inputs, descriptions of a model and a requested property and implements the algorithms to build a deductive proof to be output to the user;

- We tested the feasibility of the technique in a case study that models the functioning of a communication channel system. In the case study, we built a master proof associated to the incomplete initial system and later provided some example refinements to show how the entire flow of the approach work.

Our contribution presents two additional advantages with respect to other system verification solutions: we include the benefits of an incremental methodology and the benefits of a widely used technique such as model checking enriched by a justification of a positive output.

Even though our approach is able to prove properties of a lot of systems, one of its main drawbacks is that, at the current stage, it only supports sequential systems. Deductive proofs are indeed interesting also for concurrent systems. Nevertheless, in some cases, concurrent systems can be expressed as if they were sequential, by simply using variables assignment as they where constant propositions.

Another limitation of our approach is the time requested to compute proofs. Without doubt, a considerable wait is added to the already known complexity of model checking. We believe, nevertheless, that the insight and information gained by the proof completely repays the computational effort. In any case, the output is provided to the designer as a deductive proof to ponder, therefore the "human time" required covers the time overhead we added to traditional verification.

In conclusion, our approach represents an exploration of applicable techniques to the problem of agile verification. As a proof of concept, we re-engineered the procedure suggested in [PZ01] to be inserted in an existing model checking framework for incomplete systems.

7.2 Perspectives for future work

The directions for future improvements are various, from both theoretical and implementative perspectives. We would like to explore:

- the possibility to extend the approach to concurrent systems;
- the chance to consider sets of fairness conditions, in addition to the user specified claim. This would require adding different versions of deductive rules;
- modeling formalisms alternative to Büchi automaton, such as fair transition systems and modal transition systems (that could better grasp the essence of programs states and their evolution, especially in case of concurrency);
- the possibility to handle situations where both the system and the property are incompletely specified: including incomplete properties, would help inferring additional properties and investigate behaviors that we might want to request to systems;
- more complex case studies that could help understand the scalability of the methodology. The thesis has evaluated the presented approach using a simple case study representative of many interesting features. However, it would be interesting to consider different scenarios and apply the procedure to bigger benchmarks;

- a completion of the prototype tool proposed in this thesis. We intend to implement the sub-proofs generation and the resolution of dependencies to supply a complete framework that might be used with bigger models, guiding the user from the beginning to the end of the design process.

Our work was inspired by the idea of widening the range of agile methods of formal verification. A modular technique to perform model checking and together initialize deductive systems can be an example for other original and agile methods that help the integration of verification within the development process.

8 Bibliography

[AY01] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3): 273–303, 2001.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for ctl. In *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pages 388–397. IEEE, 1995.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[Büc90] J.Richard Büchi. On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 425–435. Springer New York, 1990.

[CBFU07] Marsha Chechik, Greg Brunet, Dario Fischbein, and Sebastian Uchitel. Partial behavioural models for requirements and early design. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[CDEG03] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4): 371–408, October 2003.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CGMZ95] Edmund M Clarke, Orna Grumberg, Kenneth L McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd*

annual ACM/IEEE Design Automation Conference, pages 427–432. ACM, 1995.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[cli] Cliche command-line shell. https://code.google.com/p/cliche/.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1): 36–72, January 1993.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *FORMAL METHODS IN SYSTEM DESIGN*, pages 275–288, 1992.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4): 626–643, December 1996.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 437–440, London, UK, UK, 1996. Springer-Verlag.

[FMS98] Bernd Finkbeiner, Zohar Manna, and HennyB. Sipma. Deductive verification of modular systems. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 239–275. Springer Berlin Heidelberg, 1998.

[GC03] Arie Gurfinkel and Marsha Chechik. Proof-like counter-examples. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 160–175, 2003.

[GMSS13] Carlo Ghezzi, Claudio Menghi, Amir Molzam Sharifloo, and Paola Spoletini. On requirements verification for model refinements. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 62–71. IEEE, 2013.

[GMSS14] Carlo Ghezzi, Claudio Menghi, Amir Molzam Sharifloo, and Paola Spoletini. On requirement verification for evolving statecharts specifications. *Requirements Engineering*, 19(3): 231–255, 2014.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In Gerard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin Heidelberg, 2001. Tool URL: http://www.lsv.ens-cachan.fr/ gastin/ltl2ba/.

[GPB02] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. pages 3–12, 2002.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3): 231–274, June 1987.

[HJS00] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. pages 155–169. Springer, 2000.

[HK90] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1): 45–59, 1990.

[HNJ+02] Thomas A. Henzinger, George C. Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. Temporal-safety proofs for systems code. In Ed Brinksma and KimGuldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer Berlin Heidelberg, 2002.

[jgr] Barak naveh. jgrapht a free java graph library, 2011.

[JS94] Jeffrey Joyce and Carl Seger. The hol-voss system: Model-checking inside a general-purpose theorem-prover. In JeffreyJ. Joyce and Carl-JohanH. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 185–198. Springer Berlin Heidelberg, 1994.

[Kic] Alexander Kick. Generation of witnesses for global μ-calculus model checking.

[KV04] Orna Kupferman and Moshe Y. Vardi. From complementation to certification, 2004.

[LSW95] Kim G. Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In *In BRICS Notes*, pages 17–40. Springer-Verlag, 1995.

[LT88] Kim G. Larsen and Bent Thomsen. Compositional proofs by partial specification of processes. In Michal P. Chytil, Václav Koubek, and Ladislav Janiga, editors, *Mathematical Foundations of Computer Science 1988*, volume 324 of *Lecture Notes in Computer Science*, pages 414–423. Springer Berlin Heidelberg, 1988.

[McM99] K.L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–237. Springer Berlin Heidelberg, 1999.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5): 1045–1079, 1955.

[MP89] Zohar Manna and Amir Pnueli. Completing the temporal picture. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and SimonettaRonchi Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 534–558. Springer Berlin Heidelberg, 1989.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[MSG15] C. Menghi, P. Spoletini, and C. Ghezzi. Dealing with incompleteness in automata-based model checking. 2015.

[Nam01] Kedar S. Namjoshi. Certifying model checkers. In Gerard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 2–13. Springer Berlin Heidelberg, 2001.

[Nam03] Kedar S. Namjoshi. Lifting temporal proofs through abstractions. In LenoreD. Zuck, PaulC. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2003.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PPZ01] Doron Peled, Amir Pnueli, and Lenore Zuck. From falsification to verification. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 292–304. Springer Berlin Heidelberg, 2001.

[PW97] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5): 243 – 246, 1997.

[PWW98] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2): 183 – 203, 1998. Concurrency Theory.

[PZ86] Amir Pnueli and Lenore Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1): 53–72, 1986.

[PZ01] Doron Peled and Lenore Zuck. From model checking to a temporal proof. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 1–14, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, jun 1995. Springer-Verlag.

[SG04] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for ctl. In *In TACAS*, pages 546–560. Springer, 2004.

[Spr98] Christoph Sprenger. A verified model checker for the modal μ-calculus in coq. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of

Lecture Notes in Computer Science, pages 167–183. Springer Berlin Heidelberg, 1998.

[SS13] Amir Molzam Sharifloo and Paola Spoletini. Lover: light-weight formal verification of adaptive systems at run time. In *Formal Aspects of Component Software*, pages 170–187. Springer Berlin Heidelberg, 2013.

[Sti95] Colin Stirling. Local model checking games (extended abstract). In Insup Lee and ScottA. Smolka, editors, *CONCUR '95: Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin Heidelberg, 1995.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[TC02] Li Tan and Rance Cleaveland. Evidence-based model checking. In Ed Brinksma and KimGuldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 455–470. Springer Berlin Heidelberg, 2002.

[UABD+13] Sebastian Uchitel, Dalal Alrajeh, Shoham Ben-David, Victor Braberman, Marsha Chechik, Guido Caso, Nicolas D'ippolito, Dario Fischbein, Diego Garbervetsky, Jeff Kramer, Alessandra Russo, and German Sibay. Supporting incremental behaviour model elaboration. *Comput. Sci.*, 28(4): 279–293, November 2013.

[UBC09] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Transactions on Software Engineering*, 35(3): 384–406, 2009.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[Wah08] Benjamin W. Wah, editor. *Wiley Encyclopedia of Computer Science and Engineering.* John Wiley & Sons, Inc., 2008.

[Wol85] Pierre Wolper. The tableau method for temporal logic: An overview. 1985.

[YL97] Shenwei Yu and Zhaohui Luo. Implementing a model checker for lego. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 442–458. Springer, 1997.