

UNIVERSIDADE FEDERAL FLUMINENSE

ERICK SIMAS GRILO

ReLo: a dynamic logic to reason about Reo circuits

NITERÓI

2021

UNIVERSIDADE FEDERAL FLUMINENSE

ERICK SIMAS GRILO

ReLo: a dynamic logic to reason about Reo circuits

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Orientadores:
Bruno Lopes

NITERÓI

2021

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

G858r Grilo, Erick Simas
ReLo: a dynamic logic to reason about Reo circuits / Erick
Simas Grilo ; Bruno Lopes Vieira, orientador. Niterói, 2021.
151 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense,
Niterói, 2021.

DOI: <http://dx.doi.org/10.22409/PGC.2021.m.16759421701>

1. Reo. 2. Coq. 3. Lógica Dinâmica. 4. Produção
intelectual. I. Vieira, Bruno Lopes, orientador. II.
Universidade Federal Fluminense. Instituto de Computação.
III. Título.

CDD -

Erick Simas Grilo

ReLo: a dynamic logic to reason about Reo circuits

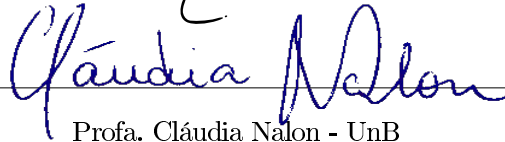
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Aprovada em junho de 2021.

BANCA EXAMINADORA



Prof. Bruno Lopes - Orientador, UFF



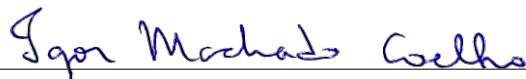
Profa. Cláudia Nalon - UnB



Prof. Edward Hermann Haeusler - PUC-Rio



Prof. Mario Benevides - UFF



Prof. Igor Machado Coelho - UFF

Niterói

2021

Aos que os possibilitam observarmos a distância, apoiando-nos em seus ombros.

Acknowledgments

First of all I thank my family, which has always given unconditional support in many situations along this way. My brother, who were always there for me. I also thank Kara for being the best Labrador dog ever, and for every time it'd check on me when I was working on this project.

To my advisor and friend Bruno Lopes, for the opportunity to work in this project, for every tip, advice and any other kind of help he has offered me, especially considering the 2020 COVID-19 pandemic period this work was conceived.

To Daniel Toledo, my best friend, for everything in the past 16 years. Especially for the help in the Byzantine Fault example explored in this work.

To all whom have helped me in any way in this project, namely the Coq-club mailing list members. I also thank the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for supporting this work.

Resumo

Sistemas críticos requerem alta confiabilidade e estão presentes nos mais variados domínios. Em suma, são sistemas cuja falha em atender requisitos específicos podem levar a perdas financeiras e até mesmo de vidas. Técnicas padrão de engenharia de *software* não são o suficiente para garantir a ausência de falhas inaceitáveis e/ou que requisitos críticos sejam satisfeitos.

Reo é uma linguagem de modelagem baseada em componentes cujo objetivo é prover um arcabouço para a construção de software baseada em sistemas já existentes, abordagem esta que costuma ser usada em uma variedade de domínios. As semânticas formais de Reo permitem que sistemas baseados em modelos Reo possam ter suas propriedades desejadas verificadas. Estas verificações podem ser feitas no sistema como um todo ou apenas considerando parte dele.

As abordagens existentes para raciocinar sobre modelos Reo utilizando lógicas requerem a conversão de uma semântica formal para um arcabouço lógico. O trabalho de [22] modela conectores Reo diretamente em *Zero-Safe Petri Nets*, uma classe especial de redes de petri que contém apenas dois tipos de lugares (*Zero* e *Safe*), sendo necessário a conversão destas redes em lógica intuicionista temporal linear.

Este trabalho apresenta *ReLo*, uma lógica dinâmica projetada para raciocinar sobre modelos Reo. *ReLo* é uma semântica formal para Reo acompanhada de provas de corretude e completude, um procedimento de provas sintáticas baseado em *tableaux*, a prova de corretude e completude deste sistema, e uma implementação de *ReLo* no assistente de provas Coq. A implementação no Coq especificamente permite a modelagem e o raciocínio sobre modelos Reo de forma automatizada através de *ReLo* em um ambiente computacional.

Abstract

Critical systems require high reliability and are present in many domains. They are systems which failure may result in financial damage or even loss of lives. Standard techniques of software engineering are not enough to ensure the absence of unacceptable failures and/or that critical requirements are fulfilled.

Reo is a component-based modelling language that aims to provide a framework to build software based on existing pieces of software, which has been used in a wide variety of domains. Its formal semantics provides grounds to certify that systems based on Reo models satisfy specific requirements (i.e., absence of deadlocks). It enables the modelling of systems that require guarantees regarding specific properties that they must meet. These guarantees may be obtained by formal verification of the system (or its required parts).

Current logical approaches for reasoning over Reo require the conversion of a formal semantics into a logical framework. *ReLo* is a dynamic logic that naturally subsumes Reo's semantics. It provides a means to reason over Reo circuits.

This work proposes *ReLo*, a dynamic logic tailored to reason over Reo models. *ReLo* serves as a Reo formal semantics with soundness and completeness proofs, a tableaux-based deductive system followed by its soundness and completeness proofs, and an implementation of *ReLo* in the Coq proof assistant. The Coq implementation provides means to the modelling and verification of *ReLo* models in a computational environment.

List of Figures

| | | |
|-----|--|----|
| 3.1 | A screenshot of CoqIde | 18 |
| 4.1 | Canonical Reo connectors | 23 |
| 4.2 | Modelling of the Sequencer in Reo | 24 |
| 4.3 | A Reo model for smart traffic lights and a controller of a crossroad | 25 |
| 4.4 | A model of the replica in the dBFT algorithm with standard Reo connectors | 26 |
| 5.1 | Examples of Reo models | 36 |

List of Tables

| | | |
|-----|---|-----|
| 2.1 | Classical Propositional Logic connectives | 7 |
| 4.1 | Basic Reo channels and their behaviour | 24 |
| 4.2 | Basic Reo channels and their respective constraint automata | 32 |
| 6.1 | Summarization of <i>applyRule</i> per tableau rule — propositional Rules. | 114 |
| 6.2 | Summarization of <i>applyRule</i> per tableau rule — modal rules. | 119 |
| 6.3 | Summarization of <i>applyRule</i> per tableau rule — iteration rules. | 120 |

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Introduction | 1 |
| 2 | Related Logic Formalisms | 6 |
| 2.1 | Classical Propositional logic | 6 |
| 2.2 | Modal Logic | 8 |
| 2.3 | Propositional Dynamic Logic | 9 |
| 2.4 | Calculus of Inductive Constructions | 12 |
| 3 | Coq | 16 |
| 3.1 | Proof Assistants | 16 |
| 3.2 | Coq | 17 |
| 4 | Reo | 22 |
| 4.1 | The modelling language | 22 |
| 4.2 | Constraint Automata | 27 |
| 5 | A dynamical logic to reason about Reo circuits | 34 |
| 5.1 | A <i>ReLo</i> Primer | 34 |
| 5.1.1 | Semantic notion of <i>ReLo</i> | 43 |
| 5.1.2 | Axiomatic System | 45 |
| 5.2 | Soundness | 46 |
| 5.3 | Completeness | 47 |

| | | |
|----------|--|------------|
| 5.4 | A Tableau for <i>ReLo</i> | 56 |
| 5.4.1 | Tableau Usage Examples | 60 |
| 5.4.2 | Termination | 63 |
| 5.4.3 | Soundness | 67 |
| 6 | A <i>ReLo</i> Implementation in Coq | 71 |
| 6.1 | Core <i>ReLo</i> definitions | 71 |
| 6.2 | Model Verification | 86 |
| 6.3 | Model Construction | 93 |
| 6.4 | A tableau for <i>ReLo</i> in Coq | 103 |
| 7 | Usage Examples | 121 |
| 7.1 | Sequencing Entities' communication in <i>ReLo</i> | 121 |
| 7.2 | Modelling Smart Cities entities interaction in <i>ReLo</i> | 123 |
| 7.3 | Byzantine Consensus | 124 |
| 7.4 | <i>ReLo</i> Tableau proofs | 127 |
| 8 | Conclusions and Further Work | 135 |
| | References | 136 |

Chapter 1

Introduction

1.1 Introduction

In software development, service-oriented computing [67] and model-driven development [9] are examples of techniques that take advantages of software models. The first technique advocates computing based on preexisting systems (services) as described by Service-Oriented Architecture (SOA), while the latter is a development technique which considers the implementation of a system based on a model. A model is an abstraction of a system (or some particular portion of it) in a specific language, which will be used as a specification basis for the system's implementation. It can be specified in languages such as Unified Modeling Language (UML) or formal specification languages like B [1] or Alloy [42]. Researchers also have applied approaches such as formal methods in software development to formalize and assure that certain (critical) systems have some required properties [45, 65].

Reo [3] is a prominent modelling language, enabling coordination of communication between interconnected systems externally from the model. Reo models are compositionally built from base connectors, where each connector in Reo stands for a specific communication pattern. Reo has proven to be successful in modeling the organization of concurrent systems' interaction, being used in a variety of applications, from process modeling to Web-Services integration [6] and even in the construction of frameworks to verify specifications in Reo [49, 78].

Standard software engineering techniques are not designed to deal with fault non-tolerant systems, namely critical systems. In many domains, such systems need a way to ensure their safety to guarantee that they indeed meet the required reliability. Formal systems compose a theoretical and implemented background able to model and reason

about systems, ensuring (mathematically) that requirements are fulfilled and that systems behave as expected.

Proof assistants [58], like Coq [30] and Isabelle [64], lead to the possibility of automatizing the verification of such systems and provide certified code. They are computational tools that implement logic systems. Their design is tailored to automatize many (when possible) steps of proofs. Some of them have a theoretical background that leads to see proofs as programs and programs as proofs.

Coordination models are models used mainly to describe concurrent and distributed computational systems. The purpose of such models is to enable software engineering based on heterogeneous software components, providing means on how these components interact with each other. Among other advantages, this enables fast deployment of new systems, by reusing already existing software that has been tested and is in production environment, which also reduces development costs. Building software following this idea requires a way to coordinate how their components will interact with each other.

The development of systems based on SOA employing model-driven development has been proved to be a valuable approach [68]. The formal verification of models that the development process is based upon enables the detection of errors that could appear only in posterior phases of software development, or even when the software reaches a productive environment, thus avoiding unplanned costs and even greater losses.

Reo's ability to model communication between software interfaces has also attracted researches over how one can formally verify Reo circuits, resulting in many different formal semantics [43] like automata-based models [4, 10, 50], coalgebraic models [3], Intuitionistic Logic with Petri Nets [22] (to name a few), and some of their implementations [49, 50, 56, 63, 77, 79, 80]. However, as far as the author is concerned, there is no logic to specifically reason about Reo models naturally, where the usage of logic-based approaches requires conversion between different formal semantics.

The fact that Reo can be used to model many real-world situations has attracted attention from researchers all around the world, resulting in a great effort directed in formalizing means to verify properties of Reo models [43, 44, 47, 49, 57, 61, 72]. Reo's formal studies also resulted in the proposal of many formal semantics for this modelling language [43], varying from operational semantics to coloring and coalgebraic models.

One of the most known formal semantics for Reo consists of Constraint Automata [11], an operational semantic in which Reo connectors are modelled as automata for *TDS*-

languages [7]. It enables reasoning over data flow of Reo connectors and when they happened. Constraint Automata have been extended to some variants which aim to enrich the reasoning process by capturing properties like timing of the data flows or possible actions over the data, respectively as Timed Constraint Automata [49] and Action Constraint Automata [48]. Some of them are briefly discussed below, along with other formal semantics for Reo.

The approach presented by Klein et al. [44] provides a platform to reason about Reo models using Vereofy,¹ a model checker for component-based systems, while Pourvatan et al. [72] propose an approach to reason about Reo models by means of symbolic execution of Constraint Automata. Kokash & Arbab [47] formally verify Long-Running Transactions (LRTs) modelled as Reo connectors using Vereofy, enabling expressing properties of these connectors in logics such as Linear Temporal Logic (LTL) or a variant of Computation Tree Logic (CTL) named Alternating-time Stream Logic (ASL). Kokash et al. [49] use mCRL2 model checker to encode Reo's semantics in Constraint Automata and other automata-based semantics, encoding their behaviour as mCRL2 processes and enabling the expression of properties regarding deadlocks and data constraints which depend upon time. Mouzavi et al. [61] propose an approach based on Maude to model checking Reo models, encoding Reo's operational semantics of the connectors. Li et al. [57] propose a real-time extension to Reo, implementing new channels and relying on Stochastic Timed Automata for Reo (STA) as its formal semantics, also providing a translation of STA to PRISM² for model checking. UPPAAL³ model checker has also been employed in the verification of Reo connectors employing the usage of Timed Constraint Automata [5] to build the corresponding UPPAAL model, and in the simulation of Hybrid Reo Connectors [8].

Proof assistants have been used to reason about Reo connectors [55, 56, 63, 79, 80]. The approaches adopted by Li et al. [55, 79] are among the ones that employ Coq to verify Reo models formally. The first work formalizes four of the Reo canonical connectors (Sync, FIFO1, SyncDrain and LossySync) along with an LTL-based language defined as an inductive type in Coq. The latter proposes the formalization of five Reo canonical channels: Sync, SyncDrain, FIFO1, Asyncdrain, and LossySync. They are modelled as logical propositions in Coq, where their behavior are defined as conjunctions regarding data and time constraints on streams denoting input and output of the automaton. Both

¹<http://www.vereofy.de>

²<https://www.prismmodelchecker.org>

³<http://www.uppaal.org/>

formalizations implement the notion of Timed Data Streams as it is the first formalization of semantics of Reo connectors [43].

The implementation proposed by Li et al. [55] enables the verification of timed properties of connectors: such properties may be proven considering the data flow a connector takes as input. The formalized LTL-based language enables bounded model checking on these connectors. However, it lacks any automatic composition operation for formalized connectors. Therefore, the composition of Reo channels need to be manually written by the user. The approach employed by Li et al. [79] implements the composition of Reo connectors employing logical conjunction of connectors' behaviour, denoted by their respective TDS.

When restricting the works considering logics and Reo, as far as the author knows there is only the work by [22] which focuses on formalizing the semantics of nine Reo connectors (Sync, LossySync, FIFO1, SyncDrain, AsyncDrain, Filter, Transform, Merger, and Replicator) in terms of zero-safe Petri nets [17], a special class of Petri-nets with two types of places: zero and stable places. This encoding is then converted to terms in Intuitionistic Temporal Linear Logic, enabling reasoning about Reo connectors in this logic.

ReLo [37] is a dynamic logic tailored to reason about Reo models, with a sound and complete axiomatization and contains a tableau-based proof procedure to syntactically reason over *ReLo* formulae. Based on Kripke frames, *ReLo* provides a framework in which one can directly formalize properties in its language, and check whether a Reo circuit as a *ReLo* model satisfies such properties. Formulas in *ReLo* are composed of symbols and indexed modalities, similar to other dynamic logics [39] to model program execution, and may be combined with logical connectives to compose other formulae. A formula $[t, \pi]\varphi$ in *ReLo* captures a Reo model as a *ReLo* program π , its data flow t and a property φ , stating that “ φ holds” in every program state reached from the current state, by executing π with t .

The adopted approach here proposed is threefold. This work presents (i) *ReLo* as a logical framework to model and reason about Reo circuits directly in a logic's language, in which the reasoning of data flows may be performed employing techniques like model checking, (ii) a prototypical implementation of this framework in Coq proof assistant, enabling the verification of properties of Reo programs in *ReLo* within a computerized environment, and (iii) the definition of a tableau-based syntactic proof system for this logic, followed by the implementation of core definitions of this tableau, enabling the

syntactic reasoning of *ReLo* in Coq.

This work is structured as follows. Chapter 2 discusses briefly some related logic formalisms with the one hereby proposed. Chapter 3 gives a glance at proof assistants, showing core concepts and keywords of Coq proof assistant, while Chapter 4 introduces Reo modelling language, along with some examples and a brief introduction to two of Reo's formal semantics: Timed Data Streams and Constraint Automata. Chapter 5 discuss *ReLo*'s main aspects, from its core definitions (such as language, models, transitions firing) and soundness and completeness proofs, to the introduction of a *ReLo* Tableau that enables syntactic reasoning. Chapter 6 discuss thoroughly a *ReLo* formalization in Coq proof assistant of the concepts defined in Chapter 6. Chapter 7 introduces some usage examples of *ReLo* and its Coq implementation. Finally, Chapter 8 closes the work by discussing the obtained results, and assessing possible future work.

Chapter 2

Related Logic Formalisms

The present work proposes a logic-based approach to reason over Reo models in Coq. In this chapter, the main aspects of the underlying logic theory present in Coq are recovered, with a focus on the logic formalisms that Coq employs. It also provides an overview of logical frameworks upon which *ReLo* is based.

2.1 Classical Propositional logic

Propositional Logic is a mathematical model which enables the reasoning of logical sentences (propositions) which contain truth values (such as true or false). Propositions can be combined to produce more complex propositions out of simpler ones [2]. As part of a branch of logic that studies methods to reason about relationship and/or how to compose propositions out of other propositions, Classical Propositional Logic is a logical system that provides the necessary apparatus to deal with such study.

Being a relatively simple mathematical model, Classical Propositional Logic allows reasoning over statements such as “Charlie is a nice boy”, represented in the logic as a propositional symbol α . Therefore, $\alpha \equiv$ “Charlie is a nice boy” denotes the attribution to α the proposition mentioned the same way variables are used to represent numbers in mathematics.

Propositions in this logic can be separated into two structural types: atomic propositions, propositions with no logical connectives, and molecular propositions, which are propositions built from other propositions using logical connectives. Logical connectives can be found in the spoken language as conjunctions such as “or”, “and”, and the negation adverb “not”. Such connectives lead to the possibility of reasoning about connected

molecular propositions.

To formally reason about these propositions, a formal system is defined to translate written sentences into propositional symbols as shown above (these are called well-formed formulae, which in the propositional calculus are atomic or molecular formulae) and inference rules, which are rules used to reason over these propositions.

In Classical Propositional Logic, both atomic and molecular propositions may denote only two truth-values: true and false. At any moment, a given proposition may either be true or false, not being able to be both at the same time.

Formally, the language of Propositional Logic can be described as follows.

Definition 2.1.1 (Language of Classical Propositional Logic). *The language of Propositional Logic can be described by*

an enumerable set Φ of propositional symbols;

a set of connectives which are interpreted as operator symbols, composing molecular statements out of other molecular or atomic propositions, namely the connectives \wedge (and), \vee (or), \rightarrow (implies), \leftrightarrow (biconditional) and \neg (not).

In what follows we discuss the semantics of Classical Propositional Logic as in [32]. It considers a space of truth values $Tv = \{t, f\}$ which can be assigned to propositions: t denoting truth and f , falsehood. The negation \neg is a unary connective which semantically can be seen as a map $\neg: Tv \rightarrow Tv$, $\neg(t) = f$ and $\neg(f) = t$. The semantics of binary connective ($\wedge, \vee, \rightarrow$, and \leftrightarrow) propositions that follows from Definition 2.1.1 are shown in Table 2.1, where φ and ψ are formulae.

| φ | ψ | \wedge | \vee | \rightarrow | \leftrightarrow |
|-----------|--------|----------|--------|---------------|-------------------|
| t | t | t | t | t | t |
| t | f | f | t | f | f |
| f | t | f | t | t | f |
| f | f | f | f | t | t |

Table 2.1: Classical Propositional Logic connectives

Classical Propositional Logic has an inference rule as follows, called “Modus ponens”. Any inference in which any well-formed formula of its language is substituted uniformly for the schematic letters in this rule is an instance of it.

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$$

A well formed formula in Classical Propositional Logic is expressed by employing the language in Definition 2.1.1 as in the following grammar, with $p \in \Phi$: $p \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid \neg\varphi$.

2.2 Modal Logic

Modal logics are defined as logics in which the notion of modality is added to formulae, employing the usage of terms “necessarily” (\Box) and “possibly” (\Diamond) to denote the truth of a statement [19, 33]. Although the term “Modal Logic” can be used to describe a family of logics, considering other modalities of other logics [33], we will focus on the two modalities presented. The structure of the language of modal logics is the one presented in Definition 2.1.1 with the modality operators \Box and \Diamond added as follows.

Definition 2.2.1 (Language of Modal Logics).

an enumerable set Φ of propositional symbols;

$p \in \Phi$: $p \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid \neg\varphi \mid \Box\varphi \mid \Diamond\varphi$;

The notion of truth in modal logics depends on whether a formula φ has a modality, and if the modality is either \Box or \Diamond . A formula $\Box\varphi$ denotes that “it is necessary that φ holds in every possible world”, while $\Diamond\varphi$ stands for “it is possible that φ holds at some possible world”. Worlds can be seen as “places” where the logical sentences hold, and they can model a variety of situations, like the current state of the execution of some machinery.

Notions of truth in modal logics rely on a structure called Kripke frames, in honor of Saul Kripke¹. A Kripke frame \mathcal{F} is defined as a pair $\mathcal{F} = \langle W, R \rangle$, where W is a set (of possible worlds), and $R \subseteq W \times W$ a relation over W denoting how the worlds relate with each other (i.e., how can one world reach another world). Then, a model in modal logics is $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$, where $V: W \rightarrow 2^\Phi$ is the valuation function: for each world $w \in W$, $V(w)$ is the set of all atomic propositional symbols that hold in w .

The semantic notion related to formulae in modal logic is stated in Definition 2.2.2. Let $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ a model in modal logic, and p, φ_1 and φ_2 be propositional formula. The

¹<https://www.gc.cuny.edu/faculty/core-bios/saul-kripke>

notion of satisfaction of a formula φ in \mathcal{M} at a state $w \in W$ of the model, denoted by $\mathcal{M}, w \Vdash p$ is defined as follows.

Definition 2.2.2 (Semantic notion of standard Modal Logic).

- $\mathcal{M}, w \Vdash p$ iff $p \in V(w)$
- $\mathcal{M}, w \Vdash \neg\varphi$ iff $\mathcal{M}, s \not\Vdash \varphi$
- $\mathcal{M}, w \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, w \Vdash \varphi_1$ and $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \vee \varphi_2$ iff $\mathcal{M}, w \Vdash \varphi_1$ or $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \rightarrow \varphi_2$ iff $\mathcal{M}, w \not\Vdash \varphi_1$ or $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \leftrightarrow \varphi_2$ iff $\mathcal{M}, w \not\Vdash \varphi_1$ or $\mathcal{M}, s \not\Vdash \varphi_2$ and $\mathcal{M}, w \Vdash \varphi_1$ or $\mathcal{M}, s \not\Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \Diamond\varphi$ if there exists a state $v \in W$, wRv and $\mathcal{M}, v \Vdash \varphi$
- $\mathcal{M}, w \Vdash \Box\varphi$ if for all states $v \in W$ such that wRv and $\mathcal{M}, v \Vdash \varphi$

We denote by $\mathcal{M} \Vdash \varphi$ if φ is satisfied in all states of \mathcal{M} . By $\Vdash \varphi$ we denote that φ is valid in any state of any model.

2.3 Propositional Dynamic Logic

Propositional Dynamic Logic (PDL) is a Dynamic Logic tailored to reason about programs. It describes how programs interact between themselves and the propositions which are not bounded to a specific domain of computation [39]. Intuitively, it can be understood as a special case of a modal logic tailored to reason about programs.

There are numerous variants of PDL which focus on employing PDL to reason over specific domains, like PDL with converse [28] or PDL without iteration [38], but we will focus on regular PDL. The syntax of PDL considers elements from Classical Logic, Modal logic, and the algebra of regular expressions (to define iteration of programs). PDL's syntactic elements must be of one of the following types: formulas (or atomic propositions) φ, ψ, \dots , programs α, β, \dots , and modalities \Box and \Diamond indexed by programs π as $[\pi]$ and $\langle \pi \rangle$. Atomic propositions are symbols p, q, \dots the same way they are defined in Classical or Modal logic. PDL's syntax is formally introduced by Definition 2.3.1.

Definition 2.3.1 (Syntax of PDL).

- an enumerable set Φ of propositional symbols;
- $p \in \Phi$: $p \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid \neg\varphi \mid [\pi]\varphi \mid \langle\pi\rangle\varphi$;
- a program π may be an atomic program γ or the result of one of the following operations over one (or more) programs α, β :
 - $\alpha; \beta$ as the sequential composition operator
 - $\alpha \cup \beta$ as the nondeterministic choice operator
 - α^* as the iteration operator
 - $\varphi?$ as the test operator

Intuitively, formulas in PDL can be interpreted as the following: $[\pi]\varphi$ denotes “It is necessary that after executing π , φ holds”. The execution of PDL programs can be interpreted as follows.

- $[\pi]\varphi$: “It is necessary that after executing π , φ holds”
- $\alpha; \beta$: “Execute α , followed by the execution of β ”
- $\alpha \cup \beta$: “Either α or β will be nondeterministically selected to be executed”
- α^* : “ α will be executed zero or a finite number of times ”
- $\varphi?$: “if φ is true, then proceed with the execution, otherwise abort”

The semantics of PDL comes from modal logics [39]. Execution of programs π in PDL are seen as relations R_π over a set of states W , $R_\pi \subseteq W \times W$, which describes the program’s execution. Notation $uR_\pi v$ means that $(u, v) \in R_\pi$. The relation of the iteration of a program α^* R_{α^*} is the reflexive transitive closure of R_α as R_α^* .

PDL also employs Kripke frames $\mathcal{F} = \langle S, R_\pi \rangle$, where S is a set of states and R_π is an indexed binary relation over a set of worlds in PDL for each atomic program π . A PDL model is a model $\mathcal{M} = \langle \mathcal{F}, V \rangle$, V as the valuation function which addresses for states $s \in S$ each formula which is valid on them, $V \subseteq S \times 2^\Phi$.

The relation R for compound programs is as follows.

- $uR_{(\alpha; \beta)}v$ iff there exists a world w such that $uR_\alpha w$ and $wR_\beta v$
- $uR_{(\alpha \cup \beta)}v$ iff $uR_\alpha v$ or $uR_\beta v$

- $uR_{(\varphi?)}v$ iff $u = v$ and $\varphi \in V(v)$

Let us consider Algorithm 1 as a PDL modelling example. We will denote each atomic statement as a different symbol. Consider the following symbolization, where φ and ψ are the tests respectively used in the “while” and “if” statements in the algorithm, and α, β , and γ are the atomic program statements.

Algorithm 1: An imperative program

```

1 while  $a \neq 100$  do
2    $a = a + 1$ ;
3   if  $x < 100$  then
4     printf("a's value is %d", a);
5   else
6      $y = 100 - x$ ;
7   end if
8 end while

```

- “ $a \neq 100$ ” $\rightarrow \varphi$
- “ $a = a + 1$,” $\rightarrow \alpha$
- “ $x < 100$ ” $\rightarrow \psi$
- “printf(“a’s value is %d”, a);” $\rightarrow \beta$
- “ $y = 100 - x$,” $\rightarrow \gamma$

Therefore, Algorithm 1 is modelled in PDL as follows. The “if” code block can be formalized as $\pi' = ((\psi?; \beta) \cup (\neg\psi?; \gamma))$, where the nondeterministic choice operator \cup models both outcomes of the “if” condition: if y holds, then β is executed, otherwise γ is executed. The “while” statement can be modelled as $\pi = ((\varphi?; \alpha; \pi') \cup \neg x?)$, where $(\varphi?; \alpha; \pi')$ models the execution of the “while” code block (lines 2-7), and $\neg x$ models the case the program execution does not enter in the “while” code block, where the program execution would begin from line 8.

Similar to Definition 2.2.2, Definition 2.3.2 introduces the semantic notion of formulae in modal logic in Definition 2.2.2. Let $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ a model in modal logic, and p, φ_1 and φ_2 be propositional formula. The notion of satisfaction of a formula p in \mathcal{M} at a state $s \in S$ of the model, denoted by $\mathcal{M}, s \models p$ is defined as follows.

Definition 2.3.2 (Semantic notion of PDL).

- $\mathcal{M}, s \Vdash p$ iff $p \in V(s)$
- $\mathcal{M}, s \Vdash \neg\varphi$ iff $\mathcal{M}, s \not\Vdash \varphi$
- $\mathcal{M}, s \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \Vdash \varphi_1$ and $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \vee \varphi_2$ iff $\mathcal{M}, w \Vdash \varphi_1$ or $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \rightarrow \varphi_2$ iff $\mathcal{M}, w \not\Vdash \varphi_1$ or $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, w \Vdash \varphi_1 \leftrightarrow \varphi_2$ iff $\mathcal{M}, w \not\Vdash \varphi_1$ or $\mathcal{M}, s \Vdash \varphi_2$ and $\mathcal{M}, w \Vdash \varphi_1$ or $\mathcal{M}, s \not\Vdash \varphi_2$
- $\mathcal{M}, s \Vdash \langle \pi \rangle \varphi$ if there exists a state $w \in S$, $sR_\pi w$, and $\mathcal{M}, s \Vdash \varphi$
- $\mathcal{M}, s \Vdash [\pi] \varphi$ if for all states $v \in W$ such that wRv (i.e., $(w, v) \in R$), $\mathcal{M}, v \Vdash \varphi$

We denote by $\mathcal{M} \Vdash \varphi$ if φ is satisfied in all states of \mathcal{M} . By $\Vdash \varphi$ we denote that φ is valid in any state of any model.

2.4 Calculus of Inductive Constructions

The Calculus of Constructions (CoC) [24] is conceived as a formalism for constructive proofs in natural deduction style, where every proof is a λ -expression typed with propositions of the implemented logic. Calculus of Constructions is based on Intuitionistic Type Theory (as proposed by Martin-Löf [59]): by removing types the result is a pure λ -expression with its associated algorithm. The Calculus of Constructions therefore leads to a high-level functional programming language that lets their users formalize definitions and prove properties about them in the same computational environment [24].

The idea behind the Calculus of Constructions is a formalism that relies heavily on the Curry-Howard Correspondence among proposition and types. Calculus of Constructions provides a powerful language to formalize constructions as a notion of a high-level fully functional programming language containing enough expressibility to allow the specification of complex algorithms, as well as the notion of data types as present in programming languages.

The Calculus of Constructions was conceived to show how powerful the Curry-Howard Correspondence is to Computer Science. In the Calculus of Constructions, every term is a λ -expression. All expressions in CoC are typed: there are types for functions, proofs, atomic types, and types for types themselves. Every object formalized in CoC must

belong to a type. Quantifiers such as the existential quantifier \exists and \forall are formalized with respect to a type and has form “exists a of type P ” and “for all a of type P ”, respectively. Expressions of type x of type P are written as $x : P$ and can be informally read as “ x belongs to P ”.

Object types in the Calculus of Constructions hold logical propositions, individual terms denoting data types, and function types. Therefore all terms in the Calculus of Constructions are formalized within the same context, where both data types and function types can be thought of as proofs of their types. As an example, the λ -abstraction $(\lambda x : M)N$ can be interpreted as a proof for $[x : M]P$ when M is a proof of P considering N as a hypothesis.

Calculus of Constructions was then extended to a variant that enables the formalization of (co-)inductive definitions. This extension was provided in 1989 as Calculus of Inductive Constructions (CiC) [26], a result of research on the extraction of programs from proofs in the Calculus of Constructions. An important aspect for this extension relies on the idea that inductively defined propositions and data types play a core role in any application [71]. Calculus of Inductive Constructions is the current logical formalism behind the Coq system. This extension enables inductively defined data types as well as principles such as proofs by induction and recursively defined functions.

Sorts in CiC have types and there is a well-founded hierarchy of types. Calculus of Inductive Constructions introduces three base sorts:

Prop is the sort of logical propositions. Let P be a logical proposition. Therefore P stands for the type of proofs of P . An element $p \in P$ is an evidence that P is provable (namely, p is a proof of P).

Set is the set of small data types, such as booleans, natural numbers, operations and functions over them.

Type is the type of all types. **Type** also contains all data types defined by **Set** and their operation, as well as larger types and operations over them. **Type** is defined because assuming **Set** is of type **Set** leads to an inconsistency. CiC provides an infinite well-founded hierarchy of sorts where **Set** and **Type** (**i**) belongs to a **Type** (**j**), where **i** < **j**.

Terms in CiC may be constructed from sorts, variables, constants, functions and their applications. They are syntactically built from the following rules:

1. the types **Prop**, **Set** and **Type** (the well-founded infinite hierarchy is hereby implicit) are terms.
2. variables and constants are terms,
3. if x is a variable, with T and U terms, then $\forall x: T, U$ is also a term.
4. if x is a variable, with T and u terms, then $\lambda x: T. u$ denoting a function that maps elements from T to u is also a term.
5. if t and u are terms, then (tu) is also a term. This term denotes the application of t on u .

CoC is conceived with an objective which is to provide in the same environment means to formalize both proofs and programs. The inference rules of CoC are defined as follows. Let Γ a context (set of logical types, propositions) and Δ a logical type. The symbol \cong denotes the congruence over propositions, contexts and terms by means of β -conversion.

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta \cong \Delta}$$

which is the identity for (logical) types.

$$\frac{\Gamma \vdash M: N}{\Gamma \vdash M \cong M}$$

as the identity for terms, where M is a well typed term of type N .

$$\frac{\Gamma \vdash M \cong N}{\Gamma \vdash N \cong M}$$

as the rule that states that the congruence over propositions is symmetric.

$$\frac{\Gamma \vdash M \cong N \quad \Gamma \vdash N \cong P}{\Gamma \vdash M \cong P}$$

denoting the transitivity of congruence over propositions, where M, N and P are terms.

$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma[x: P_1] \vdash M_1 \cong M_2}{\Gamma \vdash [x: P_1]M_1 \cong [x: P_2]M_2}$$

as the conversion rules between types P_1 and P_2 .

$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma[x: P_1] \vdash M_1 \cong M_2 \quad \Gamma[x: P_1] \vdash M_1: N_1}{\Gamma \vdash (\lambda x: P_1)M_1 \cong (\lambda x: P_2)M_2}$$

$$\frac{\Gamma \vdash (MN : P) \quad \Gamma \vdash M \cong M_1 \quad \Gamma \vdash N \cong N_1}{\Gamma \vdash (MN) \cong (M_1 N_1)}$$

as the conversion between λ -applications, with M , M_1 , N and N_1 are well typed terms of type P .

$$\frac{\Gamma[x : A] \vdash M : P \quad \Gamma \vdash N : A}{\Gamma \vdash ((\lambda x : A)MN) \cong [N/x]M'}$$

denoting the compositionally of λ -terms by means of β -reduction.

$$\frac{\Gamma \vdash M : P \quad \Gamma \vdash P \cong Q}{\Gamma \vdash M : Q}$$

as the type conversion rule, P and Q types and M a well formed term of type P

For further details regarding CoC, the work presented in [24] provide a full overview on the rules and the general aspects of this system.

Chapter 3

Coq

This chapter briefly introduces proof assistants and discusses some of the main aspects of the Coq Proof Assistant, focusing on the portion of the system employed in this work. Coq is a widely used proof assistant in many projects around the world [12, 15, 18, 35, 36, 53, 55, 57]. Before introducing Coq, details on proof assistants regarding their objectives with some background on why they became important tools in software development are briefly discussed.

3.1 Proof Assistants

Proof assistants are systems that support the process of formalizing objects like ideas and models, and properties (consisting of what one would like to prove) regarding these objects and the process of proving properties about formalized properties [34]. Based on some logical formalism, they provide an environment that provides tools to structure and process proofs.

The advent of such systems has led to a new way of proving theorems, which may lead to the reduction of the human effort involved in the steps of the process, and the human errors that could be introduced in a proof (such as typos and steps erroneously taken). By using a proof assistant, such errors are easily avoided, making the task of building up proof easier to perform. They also enable one to verify that proof has been performed in a machine environment, which may increase the trust of a proof result [35].

Their popularity began to rise in the mid-'90s when at the same time Intel faced the infamous "Pentium Bug", which resulted in the recall of millions of buggy chips and a loss of approximately USD 475 millions [23]. Although the application of proofs of correctness

of programs has been used since in the early days of computer science, it was ignored by the software industry until this time as it was dubbed “impractical” [40] by people who judged the required mathematical skills as “impossibly difficult”, followed by claims that its usage extends software development life-cycle [46].

In the past thirty years, the usage of such systems has transcended the academic environment, making its way to the software development industry in a variety of areas, to ensure properties about software or certain programs/functions: among many other initiatives, SiFive¹ is a startup based in San Mateo, California, which aims to use Coq to certify processors and other core pieces of software while Bella et. al. [13] uses Isabelle to certify an electronic payment protocol.

Unlike Automated Theorem Provers [32, 58], Proof Assistants relies on heavy user manipulation to build proofs about theorems, although some of them let their users to automatize some, if not all steps, of proofs [30], and even to rehash or build brand new proof tactics [54].

3.2 Coq

Coq [30] is a proof assistant based on an implementation of Calculus of Inductive Constructions [26, 69], a type theory based on Calculus of Constructions [24, 25]. Its objective is to provide tools to write both functional programs and proofs in higher-order logic using only one programming language named Gallina [70]. Terms of Gallina can represent programs as well as properties of these programs and proofs of these properties.

Coq offers a system that lets one develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct concerning their specification, all within a single environment that also lets their users define their proof tactics [29]. Coq also contains specific software development formalisms which enable features like the modularization of code, and type classes, which can easily relate to programming constructs that some functional languages have, such as Haskell.

There are also other proof assistants with similar objectives as Coq, but with different logical foundations, such as HOL systems, a family of interactive theorem provers based on Church’s higher-order logic including Isabelle/HOL [64], HOL4 [74] and PVS [66]. As a proof assistant, Coq has been used for many purposes: to name a few, Gonthier [35] presents a machine-checked proof of the four-color theorem, while Leroy et al. certify a

¹<https://www.sifive.com/>

compiler in Coq [53], among others [18, 57]. Figure. 3.1 shows an overview of CoqIde for Coq version 8.13.1.

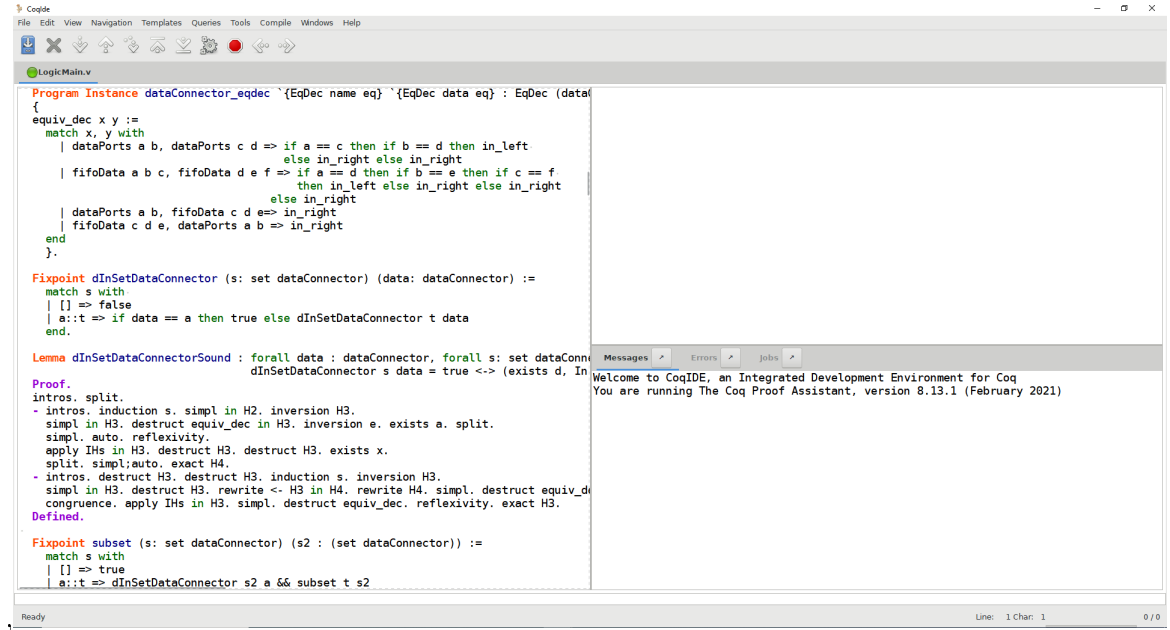


Figure 3.1: A screenshot of CoqIde

Another key property of Coq is that its language contains dependent types, which can be intuitively described as types that types depend upon values. An example of a dependent type is a list of elements where its type includes an expression denoting its size. This leads to the possibility of statically verifying the absence of out-of-bounds access. In short, dependent types may enable the possibility to enrich property verification by expressing correctness properties in the type's definition [21].

Coq offers a centralized environment to write programs, algorithms, and prove properties regarding these objects, based on the logical formalism briefly introduced in Section 2.4. All expressions formalized in Coq are named terms, and all terms have a type. Hence, every object handled in Coq is typed. There are types for propositions, programs (or functions), data types (natural numbers, booleans, lists, pairs, and many others). The types of types are called **sort**. All sorts have a type, and there is an infinite well-founded typing hierarchy of sorts, whose base sorts are Prop, Set, and Type, to avoid inconsistencies [25].

Coq's built-in language allows one to work both with program definitions and with the proof process. In what follows some of its keywords are briefly introduced. For more insights regarding these keywords, we suggest the reader refer to the system's reference

manual².

Lemma *id* : *Prop* denotes the binding of the type of a proposition to the variable *id*, enabling the interactive proving of *id* by employing Coq tactics.

Qed/Defined. **Qed** defines the proof term in Coq as an opaque term (a term which can be unfolded in tactic applications), whereas the usage of **Defined** closes the proof term as a transparent term, enabling it to be unfolded in posterior programs and proofs.

Inductive *ident* : *type* := { | *ident* : *type* } defines an inductive type whose constructors are defined by each { | *ident* : *type* } clause. The type of *ident* is *type* (which can be omitted as Coq’s type checker is capable of deducting the term’s type from its constructors). Inductive definitions are closed by types (i.e., their constructors have the same type of the definition) and they can be parametrized.

Definition *id* lets their users bind functions, theorems, (co-)inductive definitions and the evaluation of an expression (basically any well-typed term) to a variable named *id*.

Record *ident* : *sort?* := *ident?* { *ident binders* : *type* } defines a macro which constructs records as in many programming languages, similar to C’s “struct” keyword. The first identifier *ident* is the name of the defined record. The keyword *sort?* is the record’s type, and *ident?* is an identifier which defines a constructor function for the record. Both *sort?* and *ident?* may be omitted, in which Coq will try to guess the resulting record type, and instances of the record must be defined by using the regular syntax (writing each field and its corresponding value). A record may have one or more fields (denoted by the *ident* within the curly brackets) separated by “;”.

Fixpoint *param* {*struct id*} is the command that allows the definition of functions by pattern-matching over an inductive structure which is one of the **param** provided, defining recursive functions in Coq. These definitions need to meet syntactical criteria on an argument called decreasing argument. Thus, the idea of the criteria is to have a structure that tells Coq such definitions always terminate. The decreasing argument can be specified by using *struct id* or automatically guessed by Coq.

²<https://coq.inria.fr/doc/index.html>

Instance *id class_id binders : type := { id := term }* declares a class instance identified by *id*, with non-obligatory parameters *binders* and the fields declared within the scope of *{ id := term }*. Coq then generates obligations that must be proven correct in order for the term to be defined, which may be proven using Coq’s proof apparatus.

Extraction *id* enables the extraction of definition *id* to either Haskell, OCaml or Scheme. Variants like *Extraction “file.v” destFile* extracts all definitions within the file *file.v* (where *.v* is Coq’s source code file extension) to the specified target language in a file named *destFile*. This language can be set with the command **Extraction Language lang**, where *lang* is either **Scheme**, **Haskell** or **OCaml** (the default extraction language).

Program Instance *id : Type := term* uses the Program [75] tactic to bind the term defined in *term* to the variable *id* by formalizing it as if one were programming in a regular functional language, but at the same time using Coq’s proof apparatus to support in proving automatically that the function definition indeed meets its requirements. Its combination with keyword *Instance* provides tools that ease the task of dealing with obligations generated by **Instance**, by defining the **Obligation Tactic** which Program will use by default.

Coq comes packed with a core set of **tactics** and some powerful tactics defined in some modules (which require importing such modules) that eases process of proving propositions. These tactics can also be extended by a user to adapt basic tactics to a certain domain. A tactic can be defined as an intermediate between the user and Coq’s language to deal with individual parts of a proof, enabling the possibility to adapt or automate a given part of a proof [29].

A simple yet representative usage example is introduced as follows. One can use Coq to obtain certified code in other languages, such as Haskell or Scheme. Suppose one would like to formalize weekdays and then reason about the next day. This can be achieved by formalizing *weekdays* as an inductive type with its constructors denoting days of the week.

```
Inductive weekdays :=
  | monday | tuesday | wednesday | thursday | friday | saturday | sunday.
```

Then *nextDay* is a function that takes a weekday and returns the next day according to the current calendar.

```
Definition nextDay (day : weekdays) : weekdays :=
```



```

    match day with
    | monday  $\Rightarrow$  tuesday | tuesday  $\Rightarrow$  wednesday | wednesday  $\Rightarrow$  thursday
    | thursday  $\Rightarrow$  friday | friday  $\Rightarrow$  saturday | saturday  $\Rightarrow$  sunday
    | sunday  $\Rightarrow$  monday
    end.

```

Properties about *nextDay* can then be formalized, proved and the specified algorithm can be extracted to the aforementioned target languages.

Lemma *nextDayMonday* : \forall day: weekdays, *nextDay* day = monday \leftrightarrow day = sunday.

Proof.

split.

- intros. destruct day. all: inversion H. reflexivity.

- intros. rewrite H. reflexivity.

Defined.

The following Coq code illustrates the example of instantiating a typeclass in Coq, and to which extent we use it to define equality relations provided by class *EqDec*.

```

Program Instance nat_eqDec : EqDec (nat) eq :=
{ equiv_dec := fix rec x y :=
  match x,y with
  | 0, 0  $\Rightarrow$  in_left
  | Datatypes.S n, Datatypes.S m  $\Rightarrow$  if rec n m then in_left else in_right
  | 0, Datatypes.S n | Datatypes.S n, 0  $\Rightarrow$  in_right
  end
}.

```

This defines *nat_eqDec* as an equality relation for natural numbers described by *nat*. The usage of **Instance** introduces some goals generated by each of the patterns in the pattern matching defined in *equiv_dec*, which are automatically solved by **Program**'s usage.

The extraction of these definitions may be done with the command **Extraction Language Scheme**, given that the desired target extraction language is Scheme. By formalizing these definitions within a module named *example*, the command **Extraction example usageEx** generates a .scm file named *usageEx*, containing all the aforementioned definitions as Scheme code.

Chapter 4

Reo

In this chapter, a succinct overview of Reo [3, 4] is presented, considering its main characteristics with two usage examples. We also briefly introduce the main aspects of one popular formal semantics for Reo, Constraint Automata as proposed by Baier et al. [11]. We also introduce two usage examples of Reo, which will later be discussed in Chapter 6, bound to show how one can formalize and certify Reo circuits employing *ReLo*.

4.1 The modelling language

Reo plays a central role in integrating software components, especially considering Component-Based Software Engineering, where it is expected that software components are independent of each other, being more adapted to the environment they were conceived for. In recent times, software development has shifted from building large, single instances of a system to building systems by reuse of already existing pieces of software, where the full application (system) itself is generated through the orchestrated interaction of these software components, where Reo may be adequate in orchestrating such interaction.

As a coordination model, Reo focuses on connectors, their composition, and how they behave, not focusing on particular details regarding the entities that are connected, communicate, and interact through those connectors. Connected entities may be modules of sequential code, objects, agents, processes, web services, and any other software component where its integration with other software can be used to build a system [3]. Such entities are defined as component instances in Reo.

Component instances are defined as a non-empty set P that denotes a set of entities involved in an instance (process, services, actors, usually denoted by capital letters) and a predefined set of I/O operations associated with each of those entities, where they only interact with each other by the channel that connects these instances. A software component is a software implementation that may execute in physical or logical devices.

Therefore, software components are abstract entities that describe the behavior of its instances.

Channels in Reo are defined as a point-to-point link between two distinct nodes, where each channel has its unique predefined behavior. Each channel in Reo has exactly two ends, which can be of the following types: the source end, which accepts data into the channel, and the sink end, which dispenses data out of the channel. Channels are used to compose more complex connectors, being possible to combine user-defined channels amongst themselves and with the canonical connectors provided by Baier et al. [11]. Figure 4.1 shows the basic set of connectors as presented by Kokash et al. [49]. Table 4.1 relates the canonical Reo connectors with their behavior.

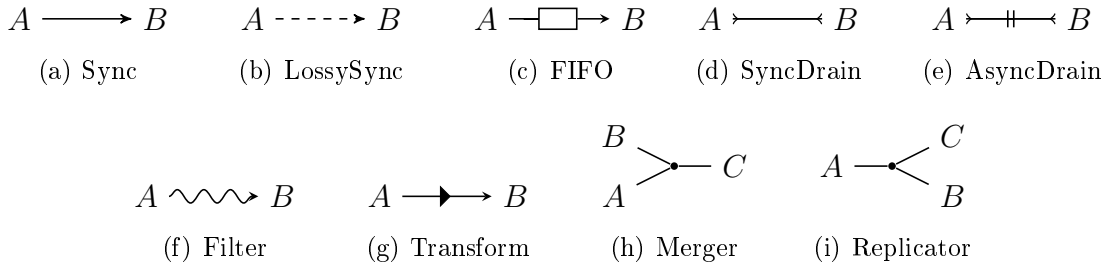


Figure 4.1: Canonical Reo connectors

A node in Reo is defined as a logical structure denoting how channel ends are linked to each other in Reo connectors. Nodes composing channel ends in Reo can be either source nodes, sink nodes, or mixed nodes. Source nodes are nodes that accept data into the channel, i.e., nodes that serve as a gateway to data flow into the channel, while sink nodes are nodes where data flows out of the channel and mixed nodes are nodes that act both as source nodes and sink nodes.

Channel ends can be used by any entity to send/receive data, given that the entity belongs to an instance that knows these ends. In other words, entities may use channels only if the instance they belong to is connected to one of the channel ends, enabling either sending or receiving data (depending on the kind of channel end the entity has access to).

The bound between a software instance and a channel end is a logical connection that does not rely on properties such as the location of the involved entities. Channels in Reo have the sole objective to enable the data exchange following the behaviour of the connectors composing the channel, utilizing I/O operations predefined for each entity in an instance. A channel can be known by zero or more instances at a time, but its ends can be used by at most one entity at the same time.

Figure 4.2 introduces a Reo connector known as Sequencer¹. It models the data flow

¹<http://reo.project.cwi.nl/v2/#examples-of-complex-connectors>

| Connector | Reo | Behaviour |
|------------|---|---|
| Sync | $A \longrightarrow B$ | Data flows Synchronously from A to B . |
| LossySync | $A \dashrightarrow B$ | Data either flows Synchronously from A to B , or it is lost in its way from A to B due so some communication failure (i.e., link failure between the interfaces). |
| FIFO | $A \rightarrow \square \rightarrow B$ | Data flows from A to B in a buffer-fashioned manner. It first leaves A , then it is stored in an intermediate place before reaching B . |
| SyncDrain | $A \multimap B$ | The data flow of A and B must be synchronized. |
| AsyncDrain | $A \multimap \# \multimap B$ | The data flow of A and B must not be synchronized. |
| Filter | $A \rightsquigarrow B$ | A data item d will successfully be transmitted from A to B if it satisfies a logical predicate P . |
| Transform | $A \rightarrow \blacktriangleright \rightarrow B$ | A data item d will be transformed by a transformation function $f: data \rightarrow data$ before reaching B . |
| Merger | $\begin{array}{c} B \\ A \end{array} \rightarrow C$ | The data from ports A and B are transmitted to C similar to the functioning of a demultiplex |
| Replicator | $A \rightarrow \begin{array}{c} C \\ B \end{array}$ | Data from A is simultaneously replicated to B and C , similar to a multiplex. |

Table 4.1: Basic Reo channels and their behaviour

between three entities sequentially. The data flows from X to the first FIFO connector (a buffer), which will be sequentially transmitted to port names A , B , and C . The Sequencer can be used to model scenarios where processes sequentially interact between themselves.

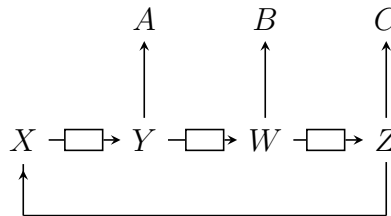


Figure 4.2: Modelling of the Sequencer in Reo

Figure 4.3 models a simplification of a scenario containing two Smart traffic lights A and B in a crossroad. Their default functioning follows a timed schedule: while one of them is green, the other is red. In addition to this timed behaviour, a controlling station

has a sensor (i.e., a camera, denoted by the upper dot) that monitors the crossroad and identifies whether there is heavy traffic waiting for the green light on one of the traffic lights [36].

Intuitively, the circuit controls the effective time a traffic light may be green or red depending on the number of cars waiting to pass. This may be done by verifying which data item is coming from both the timer and the sensor, and when is these data incoming. The circuit filters this data, to mutually exclude one of the traffic lights. The destination node (denoted by the leftmost dot in Figure 4.3) will receive the data item (0 or 1) and, based on this item decide which traffic light gains the priority to go green.

The data incoming from the uppermost dot denotes a property which the sensor has detected (i.e., many cars waiting for the traffic light to be green), while d is a data item denoting that the semaphores will alternate between open and closed, enabling the interchange of which traffic light will be either open or closed (the interchange between d and $!d$ forced by the circuit renders unable the scenario where one of the traffic lights is always open).

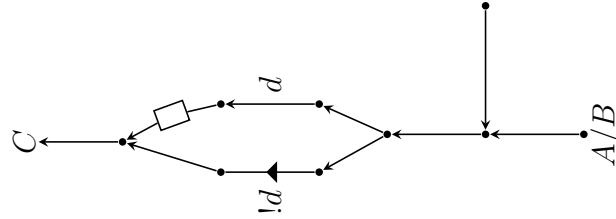


Figure 4.3: A Reo model for smart traffic lights and a controller of a crossroad

Reo can also be used to model a consensus algorithm that deals with Byzantine fault [52] to achieve consensus under uncertainty, which may be caused by different factors (varying from network issues to faulty agents). Intuitively, these machines can lead to “bad decisions” to be taken in a consensus-based distributed system. Among the algorithms conceived to address this problem, the delegated Byzantine Fault Tolerance 2.0 (dBFT) [41] was employed for the Neo Blockchain to achieve consensus globally in the system. It takes quorum-based decisions in an environment with $3f + 1$ machines, where at most f machines can present Byzantine fault-like behavior. A machine in dBFT is also called a replica.

The dBFT 2.0 is an algorithm split into three steps: I) One of the replicas will be chosen as the primary, being responsible to propose an information pack, II) the remainder of the replicas (known as backups) will validate such information, and III) finally commit to that information, as soon as $2f + 1$ valid responses are available. After commitment, the replica is unable to revert its own decision, so the logic ensures that each local commit

decision, once made by an honest replica, will represent a global commit after passing the uncertain quorum $2f + 1$ threshold. If enough time passes without changing state (assuming that the primary system may have failed), then a condition is triggered that will reset the whole process and restart from the next primary. This process is known as view change.

Figure 4.4 models the communication structure of a replica in dBFT in Reo. Node A' denotes the circuit data input, which will be filtered to the primary node P or backup node B in a round-robin scheme as follows:

- If the data flows from A to node P, after a preset time the data flow from P to the request node R. From R, there are two accessible nodes: one that will timeout after a preset time and will put the machine in the change view node V' (by flowing data from R to V'); and one that will move the system to the commit node CA' if the test condition for the validity of the block passes (by flowing data from R to C').
- If the data flow from A to node B, after a preset time it will move to the change view node VR'. Node R' represents other replicas in the network, where its data will be the requests released by the other consensus replicas, which arrive in R' with a bounded nondeterministic delay. All filters in the model that have their sink node one of the V' nodes denote the preset time the replica will wait to change its state to change view (V').

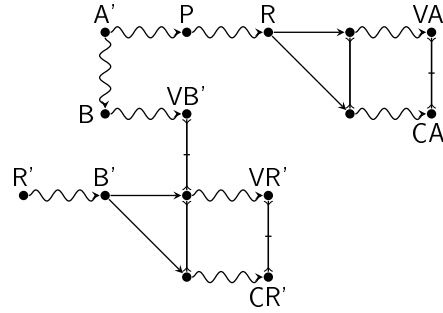


Figure 4.4: A model of the replica in the dBFT algorithm with standard Reo connectors

As a consensus replica, A' and R' interface with the remainder of the network as input nodes, where A' receives data from the network and R' as the input coming from the node R of other replicas from within the network. Nodes starting with V' and C' are output nodes, where they will broadcast data to the other replicas within the network considering whether they will commit (C') or change view (V').

From the perspective of dBFT 2.0, the filters with sink nodes B, VR', VB', R, and VA' are temporizers which control the data flow in the model. The ones with sink nodes

VR' VB' and VA' especially denotes the temporizers which controls the time out which will lead the current replica to set its state to "Change view", indicating that some failure happened and it will not go to the commit nodes CA' and CR'.

In short, Reo circuits may be understood as data flowing from different interfaces (i.e., port names connected to a node), where the connector itself models the communication pattern between two of these interfaces. A *ReLo* program is composed of one or more Reo connectors as introduced in Figure 4.1.

4.2 Constraint Automata

Constraint Automata [4] are defined as the most basic operational models for Reo, among many other formal semantics for Reo [43]. They have been extended to other formal semantics for Reo that aim to capture different sets of properties, like actions over data (Action Constraint Automata [48]) or timing properties (Timed Constraint Automata [5]).

Constraint Automata compose a basis to model and verify the specification of such coordination mechanisms by the usage of formal methods (e.g., model checking against temporal-logic specifications [62]). By using Constraint Automata as formal semantics for Reo, automata states depict the possible configurations of a channel (e.g., the data within a connector at a given time), while transitions of the automaton denote how data in the connector flow and how it changes the configuration of the automaton. In what follows, we recover and briefly discuss the main concepts from Baier et al. [11] on Constraint Automata. They are formally introduced by Definition 4.2.1.

Definition 4.2.1 (Constraint Automata). *A Constraint Automaton (CA) is a tuple $\mathcal{A} = (Q, \text{Names}, \rightarrow, Q_0)$ where*

Q is a finite set of states, configurations of \mathcal{A}

Names is a finite set of names,

$\rightarrow: Q \times 2^{\text{Names}} \times DC \times Q$ is the transition relation with DC a set of (propositional) Data Constraints, and

$Q_0 \subseteq Q$ is the set of initial states.

Constraint automata are seen as Timed Data Stream (TDS) acceptors. To understand how constraint automata relate to Timed Data Streams, we recover the main definitions from Baier et al. [11] regarding Timed Data Streams and Constraint Automata.

Let A be any set. Streams are defined as a set A^ω containing all infinite sequences over A . Therefore, $A^\omega = \{\alpha \mid \alpha: \{0, 1, 2, \dots\} \rightarrow A\}$. Individual streams are described as $\alpha =$

$\alpha(0), \alpha(1), \alpha(2), \dots$ and the derivative of a stream α is denoted as the stream initiating in the next value, namely $\alpha' = \alpha(1), \alpha(2), \alpha(3), \dots$. $\alpha^{(i)}$ denotes the i -th derivative, where $\alpha'(k) = \alpha(k+1)$ and $\alpha^{(i)}(k) = \alpha(i+k), \forall i, \forall k > 0$. Hence, TDS are composed by a stream $\alpha \in Data^\omega$, $Data$ a non-empty finite set and a time stream $a \in \mathbb{R}_+^\omega$ as a stream of increasing positive real numbers.

The behavior of Reo channels are modeled as Timed Data Streams (TDS) [7]. *TDS* models are one of the first formalizations of co-algebraic semantics of Reo connectors [43]. They introduce the notion of a channel node's behavior being a relation $R \subseteq TDS \times TDS$. Informally, *TDS* are composed of two streams, one denoting the data items that will flow through a given port and the other one denoting the time instant that the port observes this data flow. TDS are used to model how data flows through a Reo connector by discriminating the flow through the relation R . TDS are formally presented by Definition 4.2.2.

Definition 4.2.2 (Timed Data Streams).

A *Timed Data Stream* is defined as a pair of functions (α, a) as follows.

$$TDS = \{(\alpha, a) \in Data^\omega \times \mathbb{R}_+^\omega : \forall k \geq 0: a(k) \leq a(k+1) \text{ and } \lim_{k \rightarrow \infty} a(k) = \infty\}$$

Hence, TDS are composed by a stream $\alpha \in Data^\omega$ with $Data$ as a non-empty finite set and a time stream $a \in \mathbb{R}_+^\omega$, a stream of increasing positive real numbers. A TDS can be intuitively seen as a “controller” which denotes, for each data item $\alpha(k)$, the moment $a(k)$ it is flowing.

In order to formalize the concept of input/output behavior of Constraint Automata by means of TDS, a set of names \mathcal{Names} is used, where \mathcal{Names} consists of a finite set of names A_1, A_2, \dots, A_n used to identify the input/output ports that connect different components or a whole system within the environment it is inserted. For each port $A_i \in \mathcal{Names}$, a TDS is defined. Intuitively, each TDS depicts the behavior of how data flow in a port denoted by a port name $A \in \mathcal{Names}$. Definition 4.2.3 formalizes the notion of $TDS^{\mathcal{Names}}$ as the set of all TDS-tuples containing one TDS for each port.

Definition 4.2.3 ($TDS^{\mathcal{Names}}$).

$$TDS^{\mathcal{Names}} = \{((\alpha_1, a_1), (\alpha_2, a_2), \dots, (\alpha_n, a_n)) : (\alpha_i, a_i) \in TDS, i = 1, 2, \dots, n, \\ \text{with } n = |\mathcal{Names}|. \}$$

With the notion of TDS and ports as defined, a Data Assignment denotes which data element is in each port that belongs to a non-empty subset of ports $N \subseteq \mathcal{Names}$. Hence, a Data Assignment for a port is defined as a function $\delta: N \rightarrow Data$, and Definition 4.2.4 presents the notation's definition.

Definition 4.2.4 (Data Assignment). *A Data Assignment is defined as a notation for functions $\delta: N \rightarrow \text{Data}$ as $\delta = [A \rightarrow \delta_a]: A \in N$*

By defining $\theta = ((\alpha_1, a_1), (\alpha_2, a_2), \dots, (\alpha_n, a_n)): (\alpha_i, a_i) \in TDS^{\mathcal{N}_{\text{ames}}}$, $\theta.time$ is defined in Definition 4.2.5 as the time stream obtained by merging all timed streams a_1, a_2, \dots, a_n increasingly. At each iteration, $\theta.time$'s value is recalculated as the minimum time value obtained by such merging, considering θ' as the derivative of θ .

Definition 4.2.5 ($\theta.time(k)$). *The merging of time streams in increasing order denotes $\theta.time(k)$ as*

$$\theta.time(0) = \min\{a_i(0): i = 1, 2, \dots, n\},$$

$$\theta.time(k) = \min\{a_i(k): a_i(k) > \theta.time(k-1), i = 1, 2, \dots, n, k = 1, 2, \dots\}.$$

With $\theta.time(k)$ as the k -th minimum time in where data starts to flow in a port, the next definition captures the idea of selecting all ports that are in $\theta.time(k)$, $\theta.N = \theta.N(0), \theta.N(1), \theta.N(2), \dots$, as a stream over $2^{\mathcal{N}_{\text{ames}}}$ as follows.

Definition 4.2.6 ($\theta.N(k)$). *$\theta.N(k)$ denotes all ports that contains data in time instant $\theta.time(k)$:*

$$\theta.N(k) = \{A_i \in \mathcal{N}_{\text{ames}}: a_i(l) = \theta.time(k) \text{ for some } l \in \{0, 1, 2, \dots\}, i = 1, 2, \dots, n\}.$$

Considering θ , $\theta.time(k)$ and $\theta.N(k)$, the derivative of θ is written θ' as the TDS-tuple that is obtained by calculating the derivatives of all TDS (α_i, a_i) with its associated port $A_i \in \theta.N(k)$. As an example, let $\theta = ((\alpha_0, a_0), (\alpha_1, a_1), (\alpha_2, a_2))$ and $k = 0$. If $\theta.N(0) = \{A_0\}$, $\theta' = (\alpha'_0, a'_0), (\alpha_1, a_1), (\alpha_2, a_2)$.

Following the same idea presented in Definition 4.2.6, the concept of a stream over the data flow in ports in $\theta.time$ is defined as $\theta.\delta = \theta.\delta(0), \theta.\delta(1), \theta.\delta(2), \dots$ as a stream over the set of data assignments for each port $A_i \in \theta.N$. Intuitively, $\theta.\delta(k)$ holds all observed data flow at time instant $\theta.time(k)$ and is defined as follows.

Definition 4.2.7 ($\theta.\delta(k)$). *The stream $\theta.\delta(k)$ over the set of Data Assignments is defined as $\theta.\delta(k) = [A_i \rightarrow \alpha_i(l_i): A_i \in \theta.N(k)]$*

where $l_i \in [0, 1, 2, \dots]$ is the unique index with $a_i(l_i) = \theta.time(k)$.

A TDS language (for $\mathcal{N}_{\text{ames}}$) denotes any subset of $TDS^{\mathcal{N}_{\text{ames}}}$ where TDS languages are also used as a formalism to describe the possible data flow a coordination model (namely, data flow of a Reo circuit).

Constraint Automata uses a finite set \mathcal{Names} denoting Reo port names, where it can be a set of form $\{A_1, A_2, \dots, A_n\}$, and the i -th port stands for a I/O port of a Reo connector or component. As depicted by Definition 4.2.1, transitions of Constraint Automata are labeled with pairs containing a non-empty subset $N \subseteq \mathcal{Names}$ and a data constraint g . Data constraints are seen as a representation on data assignments in the sense of denoting which data item may be observed at a given port, being propositional formulae built from atomic propositions such as $d_A = d$, meaning “at port A the data item observed must be d ”, with $A \in \mathcal{Names}$ and $d \in Data$. Hence, Definition 4.2.8 formally introduces the grammar to describe data constraints.

Definition 4.2.8 (Data constraints). *A data constraint (DC) g is formally defined by the following grammar:*

$$g ::= true \mid d_A = d \mid g_1 \vee g_2 \mid \neg g.$$

We follow the same notation presented by Baier et al. [11], where a transition is denoted by $q \xrightarrow{N,g} p$ rather than $(q, N, g, p) \in \rightarrow$, $q, p \in Q$ are states of the automaton, g a data constraint which must be satisfied to enable the transition, and $N \subseteq \mathcal{Names}$. For a transition to be fired, it is required that $N \neq \emptyset$ and g is satisfiable by $\theta.\delta(k)$ at the k -th iteration of a run (i.e., $\theta.\delta(k) \models g$, where \models stands for the satisfaction relation for classical propositional logic).

The intuitive meaning of Constraint Automata as formal semantics for Reo models can be understood by interpreting the states as the configuration of the connector and the transitions as how the connector’s behavior can change in a single step. Hence, $q_0 \xrightarrow{N,g} q_1$ means that, in order for the automaton to change its configuration from q_0 to q_1 , it must have data flow only in ports $A \in N$, where such data flow must satisfy the data constraint denoted by g , while in the other ports $N \setminus \mathcal{Names}$ there must not have any data flow (i.e., $N \setminus \mathcal{Names} \not\subseteq \theta.N(k)$).

Hence, the idea of Constraint Automata being TDS acceptors can be interpreted as follows. Given an input TDS-tuple $\theta \in TDS^{\mathcal{Names}}$ as input to a Constraint Automaton \mathcal{A} , the automaton tries to figure whether θ denotes a possible data flow of \mathcal{A} the same way a finite automaton would get as input a finite word and decides whether it describes an accepting run. Nevertheless, since Constraint Automata does not have final states as a criterion for acceptance, all accepting runs are infinite runs if θ is infinite.

Formally, an accepting run in a Constraint Automaton is defined as follows.

Definition 4.2.9 (Accepting runs on Constraint Automata).

Given a TDS-tuple $\theta \in TDS^{\mathcal{Names}}$ as input, an accepting infinite run on a Constraint Automaton \mathcal{A} is denoted by the greatest set of streams $q = q_0, q_1, q_2, \dots$ over Q where:

1. *There exists a transition $q_0 \xrightarrow{N,g} q_1$;*
2. $N = \theta.N(0)$;
3. $\theta.\delta(0) \models g$;
4. q' (an infinite stream initiating from the resulting state obtained from (I)) stands for an infinite q_1 -run on θ' in \mathcal{A} ;

Therefore, Definition 4.2.9 respectively states the following: it is necessary to have at least one transition that can be fired from the actual state in the run, the other ports other than the ones involved in a firing transition contains data, the data on those ports must satisfy g , and that these conditions may hold for the remainder of θ , denoted by its derivative θ' . Such conditions establish the notion of accepting runs on Constraint Automata. Alternatively, Definition 4.2.10 formally introduces the notion of rejecting runs on Constraint Automata.

Definition 4.2.10 (Rejecting runs on Constraint Automata).

Given a TDS-tuple $\theta \in TDS^{\mathcal{N}ames}$ as input, a rejecting run a Constraint Automaton \mathcal{A} is denoted by a finite sequence $q = q_0, q_1, q_2, \dots, q_n$ over Q where:

1. from q_0 there is no transition $q_0 \xrightarrow{N,g} q_1$ with $N = \theta.N(0)$ and $\theta.\delta(0) \models g$;
2. from q_n there is a transition $q_0 \xrightarrow{N,g} q_1$ with $N = \theta.N(0)$ and $\theta.\delta(0) \models g$, but q_1, q_2, \dots, q_n denotes a rejecting run in \mathcal{A} .

Subsuming Definitions 4.2.9 and 4.2.10, an accepting run for θ in an automaton \mathcal{A} is an infinite run which satisfies Definition 4.2.9 starting at some initial state $q_0 \in Q_0$, while a rejecting run is a run for θ in \mathcal{A} where at some point k there is no transition to be fired.

Arbab [3] provides the canonical set of Reo connectors that may be used to compose more complex channels. Because Constraint Automata are a theory that provides a formal semantics for Reo connectors, a constraint automaton for each canonical connector (depicted in Figure 4.1) is also provided, each following its respective channel's behavior. Table 4.2 summarizes basic channels provided by Arbab and their corresponding Constraint Automata. The label depicted above the edges between $\{\}$ are the ports that can "observe" data for the transition to be fired, while the label below it stands for (possible) data constraints upon observed data. The absence of this second label means that there are no data constraints for this transition.

The idea of compositionally building out more complex Reo connectors out of canonical ones is to join source nodes in Reo with other nodes (sink, source or mixed) by the

| Channel | Reo | Constraint automaton |
|------------|--|----------------------|
| Sync | $A \longrightarrow B$ | |
| LossySync | $A \dashrightarrow B$ | |
| FIFO | $A \boxed{\longrightarrow} B$ | |
| SyncDrain | $A \dashrightarrow B$ | |
| AsyncDrain | $A \dashrightarrow B$ | |
| Filter | $A \rightsquigarrow B$ | |
| Transform | $A \rightarrow B$ | |
| Merger | $\begin{matrix} B \\ \searrow \\ A \end{matrix} \rightarrow C$ | |
| Replicator | $A \rightarrow \begin{matrix} C \\ \searrow \\ B \end{matrix}$ | |

Table 4.2: Basic Reo channels and their respective constraint automata

usage of a product construction between automata. Thus, the natural join of two languages L_1 and L_2 , respectively the languages of Constraint Automata \mathcal{A}_1 and \mathcal{A}_2 is done by composing the product automata of \mathcal{A}_1 and \mathcal{A}_2 as a product operation. This natural join is analogue to the operation defined for relational databases [11]. Definition 4.2.11 summarizes such operation.

Definition 4.2.11 (Product Automata). Let $A_1 = (Q_1, \mathcal{N}_{\text{ames}_1}, \rightarrow_1, Q_{0,1})$ and $A_2 =$

$(Q_2, \mathcal{Names}_2, \rightarrow_2, Q_{0,2})$ two Constraint Automata. the Product Automaton $A_1 \bowtie A_2$ is formally defined as $A_1 \bowtie A_2 = (Q_1 \times Q_2, \mathcal{Names}_1 \cup \mathcal{Names}_2, \rightarrow, Q_{0,1} \times Q_{0,2})$, where \rightarrow is the resulting transition relation, defined as follows.

$$\begin{aligned}
1. \quad & \frac{q_1 \xrightarrow{N_1, g_1} p_1, q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap \mathcal{Names}_2 = N_2 \cap \mathcal{Names}_1}{(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2)} \\
2. \quad & \frac{q_1 \xrightarrow{N, g} p_1, N \cap \mathcal{Names}_2 = \emptyset}{(q_1, q_2) \xrightarrow{N, g} (p_1, q_2)} \\
3. \quad & \frac{q_2 \xrightarrow{N, g} p_2, N \cap \mathcal{Names}_1 = \emptyset}{(q_1, q_2) \xrightarrow{N, g} (q_1, p_2)}
\end{aligned}$$

Intuitively, the rules for constructing the resulting product automaton's transitions as the natural join of languages of both automata is expressed as follows. Let \mathcal{A}_1 and \mathcal{A}_2 constraint automata. The product of \mathcal{A}_1 with \mathcal{A}_2 generates a product automaton which joins transitions from both automata where its behavior affect equally both automata (rule (i)), or are disjoint transitions (rules (ii) and (iii)).

Chapter 5

A dynamical logic to reason about Reo circuits

In this Section, we introduce *ReLo* [37] as a dynamic logic tailored to reason about Reo models. *ReLo* introduces a framework to enable the natural modeling of Reo connectors in a logic, enabling the usage of known techniques to model verification in modal logics. We will discuss its foundations, definitions, and as well proofs regarding soundness and completeness, and a syntactic proof system built for *ReLo*. Besides presenting the main concepts of the logic, we also introduce other logic-specific definitions, such as the firing of transition and program behaviour.

5.1 A *ReLo* Primer

ReLo was tailored to subsume Reo models' behaviour naturally in a logic, without needing any mechanism to convert a Reo model denoted by one of its formal semantics to some logical framework. Each basic Reo connector is modelled in the logic's language, which is defined as follows.

Definition 5.1.1 (*ReLo*'s language). *The language of ReLo consists of the following:*

- An enumerable set of propositions Φ .
- Reo channels as denoted by Figure 4.1
- A set of port names \mathcal{N}
- A sequence $Seq_{\Pi} = \{\epsilon, s_1, s_2, \dots\}$ of data flows in ports of a *ReLo* program Π (defined below). We define $s_i \leq s_j$ if s_i is a proper (i.e., s_j contains all of s_i 's data). Each sequence s_i denotes the data flow of the Reo program Π

- *Program composition symbol* : \odot
- *A sequence t of data markings of ports p with data values $\{0,1\}$, which denotes that p contains a data item. A BNF describing t is defined as follows:*

$$\begin{aligned} \langle t \rangle & ::= \langle t \rangle \langle \text{portName} \rangle \langle \text{data} \rangle \mid \langle \text{portName} \rangle \langle \text{data} \rangle \\ & \mid \langle \text{portName} \rangle ::= p \in \mathcal{N} \\ & \mid \langle \text{data} \rangle ::= 0 \mid 1 \end{aligned}$$

- *Empty sequence* ϵ
- *Iteration operator* *

The following is a simple yet intuitive example of the structure of data flows in *ReLo*. Let the sequence t be $t = \{A1, B1C\}$. It states that the port name A have the data item 1 in its current data flow, while there is a data item 1 in the FIFO between B and C .

A *ReLo* program is defined as any Reo model built from the composition of Reo channels π_i , which are defined in *ReLo* as $\Pi = (f, b)$, $\Pi = \pi_1 \odot \pi_2 \odot \dots \odot \pi_n$, and $\pi_i = (f_i, b_i)$. The set f is the set of connectors p of the model where data flows in and out of the channel (the connector has at least a source node and a sink node), namely Sync, LossySync, FIFO, Filter, Transform, Merger and Replicator. The set b is the set of blocking channels (channels without sink nodes whose inability to fire prevents the remainder of the connectors related to their port names from fire), namely SyncDrain and AsyncDrain.

Definition 5.1.2 (Logical formula in *ReLo*).

We define formulae in *ReLo* as follows: $\phi = p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \langle t, \pi \rangle \phi$, such that $p \in \Phi$. We use the standard abbreviations $\top \equiv \neg\perp$, $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$ and $[t, \pi]\varphi \equiv \neg\langle t, \pi \rangle \neg\varphi$, where π is some Reo program and t a data sequence.

Before the processing of programs in *ReLo* can begin, the expected behavior of the connector's data flow must be preserved. The connectors in Figure 5.1 are examples of compound Reo connectors. For *ReLo* they can be constructed as Π , and the order they are composed must not affect the final behaviour of the model, which must be always the same.

The model SyncFIFO is composed by a FIFO and a Sync connector in which the data leaving the FIFO is sent to C from B synchronously. Suppose that there is data in the FIFO and in port B ($t = \{A1B, B0\}$). If the FIFO from A to B is processed first than the Sync between B and C , the data flow in B will be overwritten before it is sent to C ,

which is not the correct behaviour. The Sync from B to C must fire prior to the FIFO from A to B .

Another example is denoted by the model Sync2Drain. Suppose there is data only in port name A ($t = \{A1\}$). If the Sync from B to A is evaluated first than the SyncDrain between B and C , the restriction imposed by the fact that the condition required for the SyncDrain to fire was not met (as C 's data flow differs from B 's at this moment) is not considered, and data will wrongly flow from B to A . The SyncDrain must be first evaluated before all flows as they may block the flow from data of its ports to other channels.

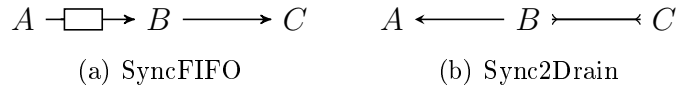


Figure 5.1: Examples of Reo models

Therefore, the next definition maps each canonical connector that compose a Reo model to a *ReLo* program as Definition 5.1.3. This will be further explored in Definition 5.1.4, an auxiliary function which interprets a Reo program $\Pi = (f, b)$ and dismembers it in a ordered sequence of connectors to be evaluated s , which will later be used to process an input t for π .

Definition 5.1.3 (*parse* base cases). *Each canonical Reo connector is mapped to a ReLo program in parse as follows:*

- $A \longrightarrow B$ to $A \rightarrow B$
- $A \dashrightarrow B$ to $(A, A \rightarrow B)$
- $A \text{ --- } \boxed{} \text{ --- } B$ to $fifo(A, B)$
- $A \text{ --- } \text{---} B$ to $SBlock(A, B)$
- $A \text{ --- } \# \text{ --- } B$ to $ABlock(A, B)$
- $A \longrightarrow \longrightarrow B$ to $Transform(f, A, B)$, $f: Data \rightarrow Data$ is a transformation function.
- $A \rightsquigarrow B$ to $Filter(P, A, B)$, P is a logical predicate over the data item in A .
- $\begin{array}{c} B \\ \diagdown \quad \diagup \\ A \end{array} \longrightarrow C$ to $(A \rightarrow C, B \rightarrow C)$
- $\begin{array}{c} C \\ \diagup \quad \diagdown \\ A \end{array} \longrightarrow B$ to $(A \rightarrow B, A \rightarrow C)$

Considering that each Reo program Π is the composition of programs $\pi_1 \odot \pi_2, \odot \dots \odot \pi_n, \pi_i = (f_i, b_i)$ as Reo programs, *parse* is formalized in Definition 5.1.4. The symbol \circ denote the addition of an element to s , the resulting set of *parse*'s processing.

Definition 5.1.4 (*parse* function). *The function that interprets the execution of a ReLo program is defined as follows. We define ε as an abbreviation to denote when there is no f or b in the signature (i.e. the base case when no program is parametrized)*

$$\text{parse}(f, b, s) = \left\{ \begin{array}{l} s, \text{ if } f = b = \varepsilon \\ \text{parse}(f_j, b, s \circ A \rightarrow B), \text{ if } f = A \longrightarrow B \odot f_j \\ \quad s \circ A \rightarrow B, f = A \longrightarrow B \\ \text{parse}(f_j, b, s \circ (A, A \rightarrow B)), \text{ if } f = A \dashrightarrow B \odot f_j \\ \quad s \circ (A, A \rightarrow B), f = A \dashrightarrow B \\ \text{parse}(f_j, b, s) \circ \text{fifo}(A, B), \text{ if } f = A \boxed{\rightarrow} B \odot f_j \\ \quad (s \circ \text{fifo}(A, B)), f = A \boxed{\rightarrow} B \\ SBlock(A, B) \circ \text{parse}(f, b_j, s), \text{ if } b = A \multimap B \odot b_j \\ \quad (SBlock(A, B) \circ s), b = A \multimap B \\ ABlock(A, B) \circ \text{parse}(f, b_j, s), \text{ if } b = A \multimap\!\!\!\multimap B \odot b_j \\ \quad (ABlock(A, B) \circ s), b = A \multimap\!\!\!\multimap B \\ \text{parse}(f_j, b, s \circ \text{Transform}(f, A, B)), \text{ if } f = A \twoheadrightarrow B \odot f_j \\ \quad (\text{Transform}(f, A, B) \circ s), f = A \twoheadrightarrow B \\ \text{parse}(f_j, b, s \circ \text{Filter}(P, A, B)), \text{ if } f = A \rightsquigarrow B \odot f_j \\ \quad (\text{Filter}(P, A, B) \circ s), f = A \rightsquigarrow B \\ \text{parse}(f_j, b_i \odot b_j, s \circ (A \rightarrow C, B \rightarrow C)), \text{ if } f = \begin{array}{c} B \\ \diagup \\ A \end{array} \rightarrow C \odot f_j \\ \quad (s \circ (A \rightarrow C, B \rightarrow C)), f = \begin{array}{c} B \\ \diagup \\ A \end{array} \rightarrow C \\ \text{parse}(f_j, b, s \circ (A \rightarrow B, A \rightarrow C)), \text{ if } f = A \rightarrow \begin{array}{c} C \\ \diagup \\ B \end{array} \odot f_j \\ \quad (s \circ (a \rightarrow b, a \rightarrow c)), f = A \rightarrow \begin{array}{c} C \\ \diagup \\ B \end{array} \end{array} \right. \quad (5.1)$$

We employ *parse* to interpret Reo programs Π as a sequence of occurrences of possible

data flow (where each flow corresponds to the execution of a Reo connector). These data flow may denote data transfer (of the form $a \rightarrow b$), flow “blocks” (i.e., (a)synchronization of data flow) induced by connectors such as SyncDrain and aSyncDrain (the first one requires that data flow synchronously through its ports, while the latter requires that data flow asynchronously through its ports). There are also the notion of a buffer introduced by FIFO connectors, which data flow into a buffer before flowing out of the channel, and merging/replicating data flow between ports, respectively denoted by channels Merger and Replicator.

There are also special data flows, denoting the “transformation” of some data flowing from A to B as $Transform(f, A, B)$ which will apply f with the data in A before it sends $f(D_A)$ (D_A denoting the data item in A) to B , and the filtering of data flow by some predicate as $Filter(P, A, B)$, P as a quantifiable-free predicate over the data item seen in A . Therefore, data will flow to B only if $P(D_A)$ is satisfied.

After processing π with *parse*, the interpretation of the execution of π is given by $go(t, s, acc)$, $go: s \times s \rightarrow s$, where s is a string denoting the processed program π as the one returned by *parse*, and t is the initial data flow of ports of the Reo program π . The parameter *acc* holds all connectors of the Reo circuit that satisfy their respective required conditions for data to flow. In what follows we define $ax \prec t$ as an operator which states that ax is in t , ax a single data of a port and t a structure containing data markup for ports $p \in \mathcal{N}$.

Example 5.1.1 shows how *parse* functions and illustrates why it is necessary. The programs that depict the FIFO connectors from Fig. 4.2 are the last programs to be executed, while the ones that denote “immediate” flow (the Sync channels) come first. This is done to preserve the data when these connectors fire (if eligible). Suppose that there is a data item in the buffer between X and Y and a data item in Y (i.e., $t = X1Y, Y0$). If the data item leaves the buffer first than the data item in Y , the latter will be overwritten and the information is lost.

Example 5.1.1. let π be the Reo program corresponding to the circuit in Fig. 4.2:

$$\pi = X \xrightarrow{\square} Y \odot Y \longrightarrow A \odot Y \xrightarrow{\square} W \odot W \longrightarrow B \odot W \xrightarrow{\square} Z \odot Z \longrightarrow C \odot Z \longrightarrow X$$

$$parse(\pi, \{\}) = \{Y \rightarrow A; W \rightarrow B; Z \rightarrow C; fifo(X, Y); fifo(Y, W); fifo(W, Z)\}$$

The usage of *parse* is required to eliminate problems regarding the execution order of π ’s Reo channels, which could be caused by processing π the way it is inputted (i.e., its connectors can be in any order). Consider, for example, the behavior of SyncDrain and aSyncDrain programs as “blocking” programs as discussed earlier. In a single step, they must be evaluated prior to the flow programs, because if they fail to execute due to

missing requirements, data should not flow from their port names to other connectors. In a nutshell, *parse* organizes the program so this verification can be performed.

Therefore, the interpretation of a π program processed by *parse* is performed by $go(t, s, acc)$, where s is a string containing π as processed by *parse*, t is π 's initial data arrangement, and acc filters the connectors of the *ReLo* program that can be fired if the requirements to do so are met.

The function *go* will check for each of the Reo connectors processed by *parse* satisfies the required condition to fire, following the connectors' behaviour in Table 4.1, which are also formalized as Constraint Automata in Table 4.2.

Definition 5.1.5 (Relation **go** for a single execution step). *We define $go(t, s, acc)$ as follows:*

- $s = \epsilon : fire(t, acc)$
- $s = A \rightarrow B \circ s' :$
 - $go(t, s', acc \circ (A \rightarrow B)), \text{ iff } Ax \prec t, (A \rightarrow B) \not\prec s'$
 - $go(t, s', (acc \circ (A \rightarrow B)) \setminus s'_j) \cup go(t, s', acc), \text{ iff } \begin{cases} Ax \prec t, \\ (A \rightarrow B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
 - $go(t, s', acc), \text{ otherwise}$
- $s = (A, A \rightarrow B) \circ s' :$
 - $go(t, s', acc \circ (A \rightarrow B)) \cup go(t, s', acc \circ (A \rightarrow A)), \text{ iff } Ax \prec t, (A \rightarrow B) \not\prec s'$
 - $go(t, s', (acc \circ (A \rightarrow B)) \setminus s'_j) \cup go(t, s', acc), \text{ iff } \begin{cases} Ax \prec t, \\ (A \rightarrow B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
 - $go(t, s', acc), \text{ otherwise}$
- $s = fifo(A, B) \circ s' :$
 - $go(t, s', acc \circ (Ax B)), \text{ iff } Ax \prec t, fifo(A, B) \not\prec s', (Ax B) \not\prec acc$
 - $go(t, s', acc \circ (Ax B \rightarrow Bx)), \text{ iff } Ax B \prec t, fifo(A, B) \not\prec s'$
 - $go(t, s', (acc \circ (Ax B \rightarrow Bx)) \setminus s'_j) \cup go(t, s', acc), \text{ iff } \begin{cases} Ax B \prec t, \\ fifo(A, B) \not\prec s', \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$
 - $go(t, s', acc), \text{ otherwise}$
- $s = Sblock(A, B) \circ s' :$
 - $go(t, s', acc), \text{ iff } \begin{cases} (Ax \prec t \wedge Bx \prec t) \vee (Ax \not\prec t \wedge Bx \not\prec t) \\ Sblock(A, B) \not\prec s' \end{cases}$
 - $go(t, halt(A, B, s'), acc), \text{ iff } \begin{cases} (Ax \prec t \wedge Bx \not\prec t) \vee (Ax \not\prec t \wedge Bx \prec t) \\ Sblock(A, B) \not\prec s' \end{cases}$

- $s = Ablock(A, b) \circ s' :$

$$- go(t, s', acc), \text{ iff } \begin{cases} (Ax \not\prec t \wedge Bx \prec t) \vee (Ax \prec t \wedge Bx \not\prec t) \vee \\ (Ax \not\prec t \wedge Bx \not\prec t), Ablock(A, B) \not\prec s' \end{cases}$$

$$- go(t, halt(A, B, s'), acc), \text{ iff } \begin{cases} (Ax \prec t \wedge Bx \prec t), \\ Ablock(A, B) \not\prec s' \end{cases}$$

- $s = Transform(f, A, B) \circ s' :$

$$- go(t, s', acc \circ (f(D_A) \rightarrow B)), \text{ iff } \begin{cases} ax \prec t \\ Transform(f, A, B) \not\prec s' \end{cases}$$

$$- go(t, s', (acc \circ (f(D_A) \rightarrow B)) \setminus s'_j) \cup go(t, s', acc), \text{ iff } \begin{cases} Ax \prec t, \\ Transform(f, A, B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$$

$$- go(t, s', acc), \text{ otherwise}$$

- $s = Filter(f, A, B) \circ s' :$

$$- go(t, s', acc \circ (A \rightarrow B)), \text{ iff } \begin{cases} Ax \prec t \\ P(D_A) \text{ holds} \\ Filter(f, A, B) \not\prec s' \end{cases}$$

$$- go(t, s', (acc \circ (A \rightarrow B)) \setminus s'_j) \cup go(t, s', acc), \text{ iff } \begin{cases} Ax \prec t, \\ P(D_A) \text{ holds} \\ Filter(f, A, B) \not\prec s' \\ \exists s'_j \in acc \mid sink(s'_j) = B \end{cases}$$

$$- go(t, s', acc), \text{ otherwise}$$

The existing condition after each return condition of *go* denotes the case where two or more Reo connectors within a circuit have the same sink node. This implies that if both of their respective source nodes have data flowing simultaneously, their sink nodes will have data flowing nondeterministically. Such condition models this scenario, considering when both cases may happen as two nondeterministic “distinct” possible executions. Therefore, the operation $acc \circ (X \rightarrow Y) \setminus s'_j$ removes every interpretation of s' which sink node equals Y , while $go(t, s', acc)$ denotes an execution containing the removed s'_j but not considering

$X \rightarrow Y$. The return condition $s = \epsilon$ denotes that the program as a whole have already been processed.

Considering the cases including block programs induced by SyncDrain and AsyncDrain connectors, $halt(A, B, s')$ is defined as a function that will be used in the case the block program conditions fails. Then, data flow that were in the ports of the SyncDrain/AsyncDrain evaluated cannot be further considered in this execution steps: channels which have their sink node pointed to A or B .

Intuitively, go is a function that processes a program π with input t as the program's data initially available at ports $p \in \pi$, and returns the next data configuration after processing all connectors and verifying whether they are eligible for data to flow. The return of go depends on a function $fire$ which is bound to return the final configuration of the Reo circuit after an iteration (i.e., the last ports that data flow). We define $sink(s'_j)$ as the sink node of a connector, in this case, the port name where a data item flowing into a Reo connector is bound to. The operation denoted by \cup is the standard set union.

As previously stated, go employs a function named $fire: T \times s \rightarrow T$, a function that returns the firing of all possible data flows in the Reo connector, given the Reo program π and an initial data flow on ports of π . The set T is the set of possible data flows as constructed by the BNF grammar in Definition 5.1.1. The function $fire$ returns the resulting data flow of this execution step by considering the program processed by go as s and the current step's data flow t . Parameter s contains *ReLo* programs as yielded by *parse*.

Definition 5.1.6 (Data marking relation ***fire***).

$$fire(t, s) = \begin{cases} \epsilon, & \text{if } s = \epsilon \\ Ax B \circ fire(t, s'), & \text{if } s = (Ax B) \circ s' \text{ and } Ax \prec t \\ B(f(a)) \circ fire(t, s'), & \text{if } s = (f(D_A) \rightarrow B) \circ s' \text{ and } Ax \prec t \\ Bx \circ fire(t, s'), & \text{if } \begin{cases} s = (A \rightarrow B) \circ s' \text{ and } Ax \prec t, \text{ or} \\ s = (Ax B \rightarrow Bx) \circ s' \text{ and } axb \prec t \end{cases} \end{cases} \quad (5.2)$$

Then, we define f_{ReLo} as the transition relation of a model. It denotes how the transitions of the model fire, i.e., given an input t and a program π denoting a Reo circuit, $f_{ReLo}(t, \pi)$ interfaces with go to return the resulting data flow of π given that data depicted by t are flowing in the connector's ports.

Definition 5.1.7. *Transition relation f_{ReLo}*

$$f_{ReLo}(t, \pi) = go(t, (parse(\pi, []), [])) \quad (5.3)$$

We define $f_{ReLo}(t, \pi^\star)$ as the application of $f_{ReLo}(t, \pi)$ iteratively for the (nondeterministic finite) number of steps denoted by \star , starting with t with π , and considering the obtained intermediate t' in the steps.

A *ReLo* frame is a structure based on Kripke frames [51] formally defined as a tuple $\mathcal{F} = \langle S, \Pi, R_\Pi, \delta, \lambda \rangle$, where each element of \mathcal{F} is described by Definition 5.1.8.

Definition 5.1.8 (*ReLo* frames).

- S is a non-empty enumerable set of states
- A *Reo* program Π .
- $R_\Pi \subseteq S \times S$ is a relation defined as follows.
 - $R_{\pi_i} = \{uR_{\pi_i}v \mid f_{ReLo}(t, \pi_i) \prec \delta(v), t \prec \delta(u)\}$ and π_i is any combination of any atomic program which is a subprogram of Π .
 - $R_{\pi_i^\star} = R_{\pi_i}^\star$, the reflexive transitive closure (RTC) of R_{π_i} .
- $\lambda: S \times \mathcal{N} \rightarrow \mathbb{R}$ is a function that returns the time instant a data item in a data markup flows through a port name of \mathcal{N} .
- $\delta: S \rightarrow T$, is a function that returns data in ports of the circuit in a state $s \in S$, T being the set of possible data flows in the model.

From Definition 5.1.8, a *ReLo* model is formally defined as a tuple $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ by Definition 5.1.9. Intuitively, it is a tuple consisting of a *ReLo* frame and a valuation function, which given a state w of the model and a propositional symbol $\varphi \in \Phi$, maps to either *true* or *false*, depending on the formula satisfiability on state w .

Definition 5.1.9 (*ReLo* models). A model in *ReLo* is a tuple $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$, where \mathcal{F} is a *ReLo* frame and $V: S \times \Phi \rightarrow \{\top, \perp\}$ is the model's valuation function

5.1.1 Semantic notion of *ReLo*

We define *ReLo*'s semantic notion intuitively as follows. Let $\mathcal{M} = \langle \mathcal{F}, \mathbf{V} \rangle$ and p, p_1 and p_2 be propositional formula. The notion of satisfaction of a formula p in \mathcal{M} at a state $s \in S$ of the model, denoted by $\mathcal{M}, s \models p$ is defined as follows.

Definition 5.1.10 (Semantic notion of *ReLo*).

- $\mathcal{M}, s \Vdash p$ iff $V(s, p) = \text{true}$
- $\mathcal{M}, s \Vdash \top$ always
- $\mathcal{M}, s \Vdash \neg\varphi$ iff $\mathcal{M}, s \not\Vdash \varphi$
- $\mathcal{M}, s \Vdash \varphi_1 \wedge \varphi_2$ iff $\mathcal{M}, s \Vdash \varphi_1$ and $\mathcal{M}, s \Vdash \varphi_2$
- $\mathcal{M}, s \Vdash \langle t, \pi \rangle \varphi$ if there exists a state $w \in S$, $s R_\pi w$, and $\mathcal{M}, w \Vdash \varphi$

We denote by $\mathcal{M} \Vdash \varphi$ if φ is satisfied in all states of \mathcal{M} . By $\Vdash \varphi$ we denote that φ is valid in any state of any model.

We recover the circuit in Fig. 4.2 as an example. Let us consider $s = D_X$, (i.e. $t = D1$) and the Sequencer's corresponding model \mathcal{M} . Therefore, $\mathcal{M}, D_X \Vdash \langle t, \pi \rangle p$ holds if $V(D_{X\text{fifo}Y}, p) = \text{true}$ as $D_{X\text{fifo}Y}$ is the only state where $D_X R_\pi D_{X\text{fifo}Y}$. For example, one might state p as “There is no port with any data flow”, hence $V(D_{X\text{fifo}Y}, p) = \text{true}$.

As a usage Example based on the same example, we formalize some properties which may be interesting for this connector to have. Let us consider that the data markup is $t = X1$, \mathcal{M} the model regarding the Sequencer, and the states' subscript denoting which part of the connector have data. The following lemma state that for this data flow, after every single execution of π , it is not the case that the three connected entities have their data equal to 1 simultaneously, but it does have data in its buffer from X to Y .

Example 5.1.2. $[X1, \pi] \neg (D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$, where $t' = f_{ReLo}(t, \pi)$

$\mathcal{M}, D_X \Vdash [X1, \pi] \neg (D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.

$\mathcal{M}, D_{X \rightarrow Y} \Vdash \neg (D_A = 1 \wedge D_B = 1 \wedge D_C = 1) \wedge t' = X1Y$.

$\mathcal{M}, D_{X \rightarrow Y} \Vdash \neg (D_A = 1 \wedge D_B = 1 \wedge D_C = 1)$ and $\mathcal{M}, D_{X \rightarrow Y} \Vdash t' = X1Y$.

Considering the scenarios where is interesting to reason regarding iteration, the notion of $\mathcal{M}, D_X \Vdash \langle t, \pi^* \rangle p$ holds if a state s is reached from D_X by means of R_π^* with $V(s, p) = \top$. If we state p as “the data item of port X equals 1”, it holds because $D_X R_\pi^* D_X$ and $V(D_X, p) = \top$. The following illustrates an example considering this scenario: if there is an execution of π that lasts a nondeterministic finite number of iterations, and there is data in C equal to 1, then there is an execution under the same circumstances where the same data has been in B .

Example 5.1.3. $\langle t, \pi^* \rangle D_C = 1 \rightarrow \langle t, \pi^* \rangle D_B = 1$

$$\mathcal{M}, D_X \Vdash \langle t, \pi^* \rangle D_C = 1 \rightarrow \langle t, \pi^* \rangle D_B = 1$$

$$\mathcal{M}, D_X \Vdash \neg(\langle t, \pi^* \rangle D_C = 1) \vee \langle t, \pi^* \rangle D_B = 1$$

$$\mathcal{M}, D_X \Vdash [t, \pi^*] \neg D_C = 1 \vee \langle t, \pi^* \rangle D_B = 1$$

$$\mathcal{M}, D_X \Vdash [t, \pi^*] \neg D_C = 1 \text{ or } \mathcal{M}, D_X \Vdash \langle t, \pi^* \rangle D_B = 1$$

$$\mathcal{M}, D_X \Vdash \langle t, \pi^* \rangle D_B = 1, \text{ because } \mathcal{M}, D_B \Vdash D_B = 1 \text{ and } D_X R_{\pi^*} R_B.$$

5.1.2 Axiomatic System

We also define an axiomatization of *ReLo* based on other dynamic logics tailored to reason about programs. We discuss and define *ReLo*'s axioms and rules as follows. Let φ and ψ be formulas. Definition 5.1.11 introduces *ReLo*'s axiomatic system. We discuss the proofs of validity (w.r.t. *ReLo*'s model) of **(R)**, **(It)**, and **(In)** in Lemma 1.

Definition 5.1.11 (Axiomatic System).

(PL) *Enough Propositional Logic tautologies*

$$\mathbf{(K)} \quad [t, \pi](\varphi \rightarrow \psi) \rightarrow ([t, \pi]\varphi \rightarrow [t, \pi]\psi)$$

$$\mathbf{(And)} \quad [t, \pi](\varphi \wedge \psi) \leftrightarrow [t, \pi]\varphi \wedge [t, \pi]\psi$$

$$\mathbf{(Du)} \quad [t, \pi]\varphi \leftrightarrow \neg \langle t, \pi \rangle \neg \varphi$$

$$\mathbf{(R)} \quad \langle t, \pi \rangle \varphi \leftrightarrow \varphi \text{ iff } f_{ReLo}(t, \pi) = \epsilon$$

$$\mathbf{(It)} \quad \varphi \wedge [t, \pi][t_{(f,b)}, \pi^*]\varphi \leftrightarrow [t, \pi^*]\varphi, \quad t_{(f,b)} = f_{ReLo}(t, \pi)$$

$$\mathbf{(In)} \quad \varphi \wedge [t, \pi^*](\varphi \rightarrow [t_{(f,b)^*}, \pi]\varphi) \rightarrow [t, \pi^*]\varphi, \\ t_{(f,b)^*} = f_{ReLo}(t, \pi^*)$$

$$\mathbf{(MP)} \quad \frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$$

$$\mathbf{(Gen)} \quad \frac{\varphi}{[t, \pi]\varphi}$$

Axioms **(PL)**, **(K)**, **(And)** and **(Du)** are standard in Modal Logic literature, along with rules **(MP)** and **(Gen)** [39]. Axiom **(It)** denotes the reasoning over nondeterministic iteration denoted by the operator \star , following a similar idea portrayed by its counterpart for PDL. An intuitive interpretation of **(It)** is as follows. If φ holds in the current state and after a single execution of π with t , any finite nondeterministic number of iterations of π with t preserves φ 's truth value, then φ must hold after any (nondeterministic finite) number of iterations of π with t .

Axiom **(In)** denotes a similar idea as the one presented by [39] for Propositional Dynamic Logic (PDL). It enables the inductive reasoning on programs by carrying the following intuitive meaning: “considering that φ holds in the current state, if after any (nondeterministic finite) number of iterations of π with its respective input t φ holds, then it will hold after any number of iterations of π taking t as its input.

We define proofs in *ReLo* as the formula obtained from successive applications of axioms and/or inference rules (MP and Gen) to an axiom. Consistency in *ReLo* is defined as follows: if $\vdash \varphi$, then $\models \varphi$.

5.2 Soundness

In this section, we discuss the soundness of *ReLo* w.r.t the axiomatic system Definition 5.1.11 introduces, formally defining it in terms of Lemma 1 as follows.

Lemma 1 (Soundness of *ReLo*'s axiomatic system). If $\vdash \varphi$ then $\models \varphi$.

1. **(R)**: $\langle t, \pi \rangle \varphi \leftrightarrow \varphi$ if $f_{ReLo}(t, \pi) = \epsilon$

Proof.

Suppose by contradiction that exists a state s from a model $\mathcal{M} = \langle S, \Pi, R_\Pi, \delta, \lambda, V \rangle$ where **(R)** does not hold. There are two possible cases.

(\Rightarrow)

Suppose by contradiction $\mathcal{M}, s \models \langle t, (f, b) \rangle \varphi$ and $\mathcal{M}, s \not\models \varphi$. $\mathcal{M}, s \models \langle t, (f, b) \rangle \varphi$ iff there is a state $v \in S$ such that $s R_\pi v$. Because $f_{ReLo}(t, (f, b)) = \epsilon, s = v$ (i.e., in this execution no other state is reached from s). Therefore, $\mathcal{M}, s \models \varphi$, contradicting $\mathcal{M}, s \not\models \varphi$.

(\Leftarrow)

Suppose by contradiction $\mathcal{M}, s \models \varphi$ and $\mathcal{M}, s \not\models \langle t, (f, b) \rangle \varphi$. In order to $\mathcal{M}, s \not\models \langle t, (f, b) \rangle \varphi$, for every state $v \in S$ such that $s R_\pi v$, $\mathcal{M}, v \not\models \varphi$. Because $f_{ReLo}(t, (f, b)) = \epsilon, s = v$ (i.e., in this execution no other state is reached from s). Therefore, $\mathcal{M}, v \not\models \varphi$, contradicting $\mathcal{M}, v \models \varphi$.

□

2. **(It)**: $\varphi \wedge [t, \pi][t, \pi^*]\varphi \leftrightarrow [t, \pi^*]\varphi$

Proof.

Suppose by contradiction that exists a state s from a model $\mathcal{M} = \langle S, \Pi, R_\Pi, \delta, \lambda, V \rangle$ where **(It)** does not hold. There are two possible cases.

(\Rightarrow)

Suppose by contradiction $\mathcal{M}, s \models \varphi \wedge [t, (f, b)][t, (f, b)^*]\varphi$ and $\mathcal{M}, s \not\models [t, (f, b)^*]\varphi$. Therefore, $\mathcal{M}, s \models \varphi$ and $\mathcal{M}, s \models [t, (f, b)][t, (f, b)^*]\varphi$. For $\mathcal{M}, s \models [t, (f, b)][t, (f, b)^*]\varphi$, every state $w \in S$ such that $s R_\pi w$, $\mathcal{M}, w \models [t, \pi^*]\varphi$. Then, for $\mathcal{M}, w \models [t, \pi^*]\varphi$,

every state $v \in S$ such that $wR_{\pi^*}v$, $\mathcal{M}, v \Vdash \varphi$. From $\mathcal{M}, s \nVdash [t, \pi^*]\varphi$, there is a state $u \in S$ such that $sR_{\pi^*}u$ and $\mathcal{M}, u \nVdash \varphi$. Because $sR_{\pi}w$ and $wR_{\pi^*}v$ from R_{π^*} we have $sR_{\pi^*}v$. Then, for all u such that $sR_{\pi^*}u$, $\mathcal{M}, u \Vdash \varphi$ which contradicts the existence of a state u such that $sR_{\pi^*}u$ and $\mathcal{M}, u \nVdash \varphi$.

(\Leftarrow)

Suppose by contradiction $\mathcal{M}, s \Vdash [t, (f, b)^*]\varphi$ and $\mathcal{M}, s \nVdash \varphi \wedge [t, (f, b)][t, (f, b)^*]\varphi$. Then, $\mathcal{M}, s \Vdash \neg(\varphi \wedge [t, (f, b)][t, (f, b)^*]\varphi)$ which is $\mathcal{M}, s \Vdash \neg\varphi$ or $\mathcal{M}, s \Vdash \neg[t, (f, b)][t, (f, b)^*]\varphi$. In order to $\mathcal{M}, s \Vdash [t, (f, b)^*]\varphi$, for each state $w \in S$ such that sR_{π}^*w , $\mathcal{M}, w \Vdash \varphi$. By the definition of R_{π}^* , sR_{π}^*s . Then, $\mathcal{M}, s \Vdash \varphi$ which contradicts $\mathcal{M}, s \Vdash \neg\varphi$. Now, considering $\mathcal{M}, s \Vdash \neg[t, (f, b)][t, (f, b)^*]\varphi$, for each state $v \in S$ such that $sR_{\pi}v$, $\mathcal{M}, v \nVdash [t, (f, b)^*]\varphi$. Then, to $\mathcal{M}, v \nVdash [t, (f, b)^*]\varphi$, there exists a state $u \in S$ such that vR_{π}^*u and $\mathcal{M}, u \nVdash \varphi$. Because $sR_{\pi}v$ and vR_{π}^*u , sR_{π}^*u . From $\mathcal{M}, w \Vdash \varphi$, there is no state u such that $\mathcal{M}, u \nVdash \varphi$, which is a contradiction. \square

3. (Ind): $\varphi \wedge [t, \pi^*](\varphi \rightarrow [t, \pi]\varphi) \rightarrow [t, \pi^*]\varphi$

Proof.

Suppose by contradiction that exists a state s from a model $\mathcal{M} = \langle S, \Pi, R_{\Pi}, \delta, \lambda, V \rangle$ where (Ind) does not hold. Therefore, suppose $\mathcal{M}, s \Vdash \varphi \wedge [t, (f, b)^*](\varphi \rightarrow [t, (f, b)]\varphi)$ and $\mathcal{M}, s \nVdash [t, (f, b)^*]\varphi$. Then, $\mathcal{M}, s \Vdash \varphi$ and $\mathcal{M}, s \Vdash [t, (f, b)^*](\varphi \rightarrow [t, (f, b)]\varphi)$. Then, for all states $w \in S$ such that sR_{π}^*w , $\mathcal{M}, w \Vdash \varphi \rightarrow [t, (f, b)]\varphi$ which is $\mathcal{M}, w \Vdash \neg\varphi \vee [t, (f, b)]\varphi$. Then, $\mathcal{M}, w \Vdash \neg\varphi$ or $\mathcal{M}, w \Vdash [t, (f, b)]\varphi$. Considering $\mathcal{M}, w \Vdash \neg\varphi$, by R_{π}^* we have sR_{π}^*s . Because $\mathcal{M}, s \Vdash \varphi$, there is a state w (namely, $s = w$) where $\mathcal{M}, w \Vdash \neg\varphi$ does not hold, resulting in a contradiction. Now, considering $\mathcal{M}, w \Vdash [t, (f, b)]\varphi$, for each state $v \in S$ such that $wR_{\pi}v$, $\mathcal{M}, v \Vdash \varphi$. Because sR_{π}^*w and $wR_{\pi}v$, by the definition of R_{π}^* sR_{π}^*v . Then, if $\mathcal{M}, s \nVdash [t, (f, b)^*]\varphi$ then there is a state $u \in S$ such that sR_{π}^*u and $\mathcal{M}, u \nVdash \varphi$. Because for each state $v \in S$ such that $wR_{\pi}v$, $\mathcal{M}, v \Vdash \varphi$, such state u cannot exist as its existence is a contradiction. \square

5.3 Completeness

We start by defining the Fisher-Ladner closure of a formula as the set closed by all of its subformulae, following the idea employed in other modal logic works [14, 39] as follows.

Definition 5.3.1 (Fisher-Ladner Closure). *Let Φ be a the set of all formulae in $ReLo$. The Fischer-Ladner closure of a formula, notation $FL(\varphi)$ is inductively defined as follows:*

- $FL: \Phi \rightarrow 2^\Phi$
- $FL_{(f,b)}: \{\langle t, (f, b) \rangle \varphi\} \rightarrow 2^\Phi$, where (f, b) is a $ReLo$ program and φ a $ReLo$ formula.

These functions are defined as

- $FL(p) = \{p\}$, p an atomic proposition;
- $FL(\varphi \rightarrow \psi) = \{\varphi \rightarrow \psi\} \cup FL(\varphi) \cup FL(\psi)$
- $FL_{(f,b)}(\langle t, (f, b) \rangle \varphi) = \{\langle t, (f, b) \rangle \varphi\}$
- $FL(\langle t, (f, b) \rangle \varphi) = FL_{(f,b)}(\langle t, (f, b) \rangle \varphi) \cup FL(\varphi)$
- $FL_{(f,b)}(\langle t, (f, b)^* \rangle \varphi) = \{\langle t, (f, b)^* \rangle \varphi\} \cup FL_{(f,b)}(\langle t, (f, b) \rangle \langle t, (f, b)^* \rangle \varphi)$

From the definitions above, we prove two lemmas which can be understood as properties that formulae need to satisfy in order to belong to their Fisher-Ladner closure.

Lemma 2. If $\langle t, (f, b) \rangle \psi \in FL(\varphi)$, then $\psi \in FL(\varphi)$

Lemma 3. If $\langle t, (f, b)^* \rangle \psi \in FL(\varphi)$, then $\langle t, (f, b) \rangle \langle t, (f, b)^* \rangle \psi \in FL(\varphi)$

The proofs for Lemmas 2 and 3 are straightforward from Definition 5.3.1. The following definitions regard the definitions of maximal canonical subsets of $ReLo$ formulae. We first extend Definition 5.3.1 to a set of formulae Γ . The Fisher-Ladner closure of a set of formulae Γ , $FL(\Gamma) = FL(\varphi)$, $\forall \varphi \in \Gamma$. Therefore, $FL(\Gamma)$ is closed under subformulae.

We prove that the Fisher-Ladner closure of a finite set Γ is also finite. For the remainder of this section, we will assume that Γ is finite.

Lemma 4. If Γ is a finite set of formulae, then $FL(\Gamma)$ also is a finite set of formulae

Proof. The proof is standard in literature [16]. Intuitively, because FL is defined recursively over a set of formulae Γ into formulae ψ of a formula $\varphi \in \Gamma$, Γ being finite leads to the resulting set of $FL(\Gamma)$ also being finite (at some point, all atomic formulae composing φ will have been reached by FL). \square

Definition 5.3.2 (Atom). *Let Γ be a set of consistent formulae. An atom of Γ is a set of formulae Γ' that is a maximal consistent subset of $FL(\Gamma)$. The set of all atoms of Γ is defined as $At(\Gamma)$.*

Lemma 5. Let Γ a consistent set of formulae and ψ a *ReLo* formula. If $\psi \in FL(\Gamma)$, and ψ is satisfiable then there is an atom of Γ , Γ' where $\psi \in \Gamma'$.

Proof. The proof follows from Lindembaum's lemma. From Lemma 4, as $FL(\Gamma)$ is a finite set, its elements can be enumerated from $\gamma_1, \gamma_2, \dots, \gamma_n, n = |FL(\Gamma)|$. The first set, Γ'_1 contains ψ as the starting point of the construction. Then, for $i = 2, \dots, n$, Γ'_i is the union of Γ'_{i-1} with either $\{\gamma_i\}$ or $\{\neg\gamma_i\}$, respectively whether $\Gamma'_i \cup \{\gamma_i\}$ or $\Gamma'_i \cup \{\neg\gamma_i\}$ is consistent. In the end, we make $\Gamma' = \Gamma'_n$ as it contains the union of all $\Gamma_i, 1 \leq i \leq n$. This is summarized in the following bullets:

- $\Gamma'_1 = \{\psi\};$
- $\Gamma'_i = \begin{cases} \Gamma'_{i-1} \cup \{\gamma_i\}, & \text{if } \Gamma'_{i-1} \cup \{\gamma_i\} \text{ is consistent} \\ \Gamma'_{i-1} \cup \{\neg\gamma_i\}, & \text{otherwise} \end{cases} \quad \text{for } 1 < i < n;$
- $\Gamma = \bigcup_{i=1}^n \Gamma_i$

□

Definition 5.3.3 (Canonical relations over Γ). Let Γ a set of formulae, A, B atoms of Γ ($A, B \in At(\Gamma)$), Π a *ReLo* program and $\langle t, (f, b) \rangle \varphi \in At(\Gamma)$. The canonical relations on $At(\Gamma)$ is defined as S_Π^Γ as follows:

$$AS_\Pi^\Gamma B \leftrightarrow \bigwedge A \wedge \langle t, (f, b) \rangle \bigwedge B \text{ is consistent}$$

$$AS_{\Pi^*}^\Gamma B \leftrightarrow \bigwedge A \wedge \langle t, (f, b)^* \rangle \bigwedge B \text{ is consistent}$$

Definition 5.3.3 states that the relation between two atoms of Γ , A and B is done by the conjunction of the formulae in A with all formulae in B which can be accessed from A with a diamond formula, such that this conjunction is also a consistent formula. Intuitively, it states that A and B are related in S_Π^Γ by every formula φ of B which conjunction with A by means of a diamond results in a consistent scenario.

The following definition is bound to formalize the canonical version of δ as the data markup function.

Definition 5.3.4 (Canonical data markup function δ_c^Γ).

Let $F = \{\langle t_1, (f_1, b_1) \rangle \varphi_1, \langle t_2, (f_2, b_2) \rangle \varphi_2, \dots, \langle t_n, (f_n, b_n) \rangle \varphi_n\}$ be the set of all diamond formula occurring on an atom A of Γ . The canonical data markup is defined as $\delta_c^\Gamma: At(\Gamma) \rightarrow T$ as follows:

- The sequence $\{t_1, t_2, \dots, t_n\} \subseteq \delta(A)$ Therefore, $\{t_1, t_2, \dots, t_n\} \subseteq \delta_c^\Gamma(A)$. Intuitively, this states that all the data flow in the set of formulae must be valid data markups of A , which leads to them to also be valid data markups of δ_c^Γ following Definition 5.3.3.
- for all programs $\pi = (f, b) \in \Pi$, $f_{ReLo}((\delta_c^\Gamma(A)), (f, b)) \prec \delta_c^\Gamma(B) \leftrightarrow AS_\Pi^\Gamma B$.

Definition 5.3.5 (Canonical model). A canonical model over a set of formulae Γ is defined as a ReLo model $\mathcal{M}_c^\Gamma = \langle At(\Gamma), \Pi, S_\Pi^\Gamma, \delta_c^\Gamma, \lambda_c, V_c^\Gamma \rangle$, where:

- $At(\Gamma)$ is the set of states of the canonical model;
- Π is the model's ReLo program;
- S_Π^Γ are the canonical relations over Γ ;
- δ_c^Γ is the canonical markup function;
- $\lambda_c: At(\Gamma) \times \mathcal{N} \rightarrow \mathbb{R}$;
- $V_c^\Gamma: At(\Gamma) \times \varphi \rightarrow \{true, false\}$, namely $V_c^\Gamma(A, p) = \{A \in At(\Gamma) \mid p \in A\}$;

Lemma 6. For all programs $\pi = (f, b)$ that compose Π , $t = \delta_c^\Gamma(A)$:

1. if $f_{ReLo}(t, (f, b)) \neq \epsilon$, then $f_{ReLo}(t, (f, b)) \prec \delta_c^\Gamma(B)$ iff $AS_\Pi^\Gamma B$.
2. if $f_{ReLo}(t, (f, b)) = \epsilon$, then $(A, B) \notin S_\Pi^\Gamma$.

Proof. The proof for 1. is straightforward from Definition 5.3.4. The proof for 2. follows from axiom R . Because $f_{ReLo}(t, (f, b)) = \epsilon$, no other state is reached from the current state, hence no state B related with A by R_Π^Γ can be reached. \square

The following lemma states that canonical models always exists if there is a formula $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$, a set of formulae Γ and a Maximal Consistent Set $A \in At(\Gamma)$. This assures that given the required conditions, a canonical model can always be built.

Lemma 7 (Existence Lemma for canonical models). Let A be an atom of $At(\Gamma)$ and $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$. $\langle t, (f, b) \rangle \varphi \in A \iff$ exists an atom $B \in At(\Gamma)$ such that $AS_\Pi^\Gamma B$, $t \prec \delta_c^\Gamma(A)$ and $\varphi \in B$.

Proof. \Rightarrow

Let $A \in At(\Gamma)$ $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$ and $\langle t, (f, b) \rangle \varphi \in A$. Because $A \in At(\Gamma)$, from Definition 5.3.4 we have $t \prec \delta_c^\Gamma(A)$. From Lemma 5 we have that if $\psi \in FL(\Gamma)$ and ψ is consistent, then there is an atom of Γ , Γ' where $\psi \in \Gamma'$. Rewriting φ as $(\varphi \wedge \gamma) \vee (\varphi \wedge \neg \gamma)$

(a tautology from Propositional Logic), an atom $B \in At(\Gamma)$ can be constructed, because either $\langle t, (f, b) \rangle (\varphi \wedge \gamma)$ or $\langle t, (f, b) \rangle (\varphi \wedge \neg \gamma)$ is consistent. Therefore, considering all formulae $\gamma \in FL(\Gamma)$, $B \in At(\Gamma)$ is constructed with $\varphi \in B$ and $A \wedge (\langle t, (f, b) \rangle \varphi) \wedge B$. From Definition 5.3.3, $AS_{\Pi}^{\Gamma} B$.

\Leftarrow

Let $A \in At(\Gamma)$ and $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$. Also, let $B \in At(\Gamma)$, $AS_{\Pi}^{\Gamma} B$, $t \prec \delta_c^{\Gamma}(A)$, and $\varphi \in B$. Because $AS_{\Pi}^{\Gamma} B$, from Definition 5.3.3, $AS_{\Pi}^{\Gamma} B \leftrightarrow (A \wedge \langle t, (f, b) \rangle \varphi) \wedge B$, $\forall \varphi_i \in B$ is consistent. From $\varphi \in B$, $(A \wedge \langle t, (f, b) \rangle \varphi)$ is also consistent. As $A \in At(\Gamma)$ and $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$, by Definition 5.3.2, as A is maximal, then necessarily $\langle t, (f, b) \rangle \varphi \in A$. \square

The following lemma formalizes the truth notion for a canonical model \mathcal{M}_c^{Γ} , given a state s and a formula φ . It formalizes the semantic notion for canonical models in *ReLo*.

Lemma 8 (Truth Lemma). Let $\mathcal{M}_c^{\Gamma} = \langle At(\Gamma), \Pi, S_{\Pi}^{\Gamma}, \delta_c^{\Gamma}, \lambda, V_c^{\Gamma} \rangle$ be a canonical model over a formula γ . Then, for every state $A \in At(\Gamma)$ and every formula $\varphi \in FL(\gamma)$:

$$\mathcal{M}_c^{\Gamma}, A \models \varphi \iff \varphi \in A$$

Proof. The proof proceeds by induction over the structure of φ .

- Induction basis: suppose φ is a proposition p . Therefore, $\mathcal{M}_c^{\Gamma}, A \models p$. From Definition 5.3.5, \mathcal{M}_c^{Γ} 's valuation function is $V_c^{\Gamma}(p) = \{A \in At(\Gamma) \mid p \in A\}$. Therefore, $p \in A$.
- Induction Hypothesis: Suppose φ is a non atomic formula ψ . Then, $\mathcal{M}_c^{\Gamma}, A \models \psi \iff \psi \in A$, ψ a strict subformula of φ .
- Inductive step: Let us prove it holds for the following cases:
 - Case $\varphi = \neg \psi$: by the induction hypothesis, $\mathcal{M}_c^{\Gamma}, A \models \neg \psi \iff \neg \psi \in A$. Because A is maximal w.r.t. ϕ , then $\neg \psi \in A$ holds by \mathcal{M}_c^{Γ} 's definition.
 - Case $\varphi = \psi_1 \wedge \psi_2$: by the induction hypothesis, $\mathcal{M}_c^{\Gamma}, A \models \psi_1 \wedge \psi_2 \iff \psi_1 \in A$ and $\psi_2 \in A$, which holds by \mathcal{M}_c^{Γ} 's definition.
 - Other connectives ($\vee, \rightarrow, \leftrightarrow$): their proof may be derived from the items above.
 - Case $\varphi = \langle t, (f, b) \rangle \phi$. Then, $\mathcal{M}_c^{\Gamma}, A \models \langle t, (f, b) \rangle \phi \iff \langle t, (f, b) \rangle \phi \in A$:
 \rightarrow

Let $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b) \rangle \phi$. From Definition 5.3.3, there is a state B where $AS_\Pi^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t, (f, b) \rangle \phi \in A$. Therefore, it holds.

←

Let $\mathcal{M}_c^\Gamma, A \not\models \langle t, (f, b) \rangle \phi$. From Definition 5.3.5's valuation function V_c^Γ and Lemma 5, we have $\mathcal{M}_c^\Gamma, A \models \neg \langle t, (f, b) \rangle \phi$. Therefore, for every B where $AS_\Pi^\Gamma B, \mathcal{M}_c^\Gamma, B \models \neg \phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t, (f, b) \rangle \phi \notin A$.

- Case $\varphi = \langle t, (f, b)^* \rangle \phi$. Then, $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b)^* \rangle \phi \iff \langle t, (f, b)^* \rangle \phi \in A$:

→

Let $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b)^* \rangle \phi$. From Definition 5.3.3, there is a state B where $AS_{\Pi^*}^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t, (f, b)^* \rangle \phi \in A$. Therefore, it holds.

←

Let $\mathcal{M}_c^\Gamma, A \not\models \langle t, (f, b)^* \rangle \phi$. From Definition 5.3.5's valuation function V_c^Γ and Lemma 5, we have $\mathcal{M}_c^\Gamma, A \models \neg \langle t, (f, b)^* \rangle \phi$. Therefore, for every B where $AS_{\Pi^*}^\Gamma B, \mathcal{M}_c^\Gamma, B \models \neg \phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t, (f, b)^* \rangle \phi \notin A$.

□

We proceed by formalizing the following lemma, which is bound to show that the properties that define \star for regular *ReLo* models also holds in *ReLo* canonical models.

Lemma 9. Let $A, B \in At(\Gamma)$ and Π a *ReLo* program. If $AS_{\Pi^*} B$ then $AS_\Pi^* B$

Proof. Suppose $AS_{\Pi^*} B$. Define $C = \{C' \in At(\Gamma) \mid AS_\Pi^* C'\}$ as the set of all atoms C' which A reaches by means of S_{Π^*} . We will show that $B \in C$. Let C_c be the maximal consistent set obtained by means of Lemma 5, $C_c = \{\bigwedge C_1 \vee C_2 \vee \dots \bigwedge C_n\}$, where the conjunction of each C_i is consistent, and each C_i is a maximal consistent set. Also, define $t = \delta_c^\Gamma(C_c)$ as the canonical markup of C_c .

Note that $C_c \wedge \langle t, (f, b) \rangle \neg C_c$ is inconsistent: if it was consistent, then for some $D \in At(\Gamma)$ which A cannot reach, $C_c \wedge \langle t, (f, b) \rangle \bigwedge D$ would be consistent, which leads to $\bigwedge C_1 \vee C_2 \vee \dots \vee C_i \vee \langle t, (f, b) \rangle \bigwedge D$ also being consistent, for some C_i . By the definition of C_c , this means that $D \in C$ but that is not the case (because $D \in C_c$ contradicts D not being reached from A and consequently C_c 's definition, as $D \in C_c$ leads to D being reachable from A). Following a similar reasoning, $\bigwedge A \wedge \langle t, (f, b) \rangle C_c$ is also inconsistent and therefore its negation, $\bigwedge \neg(A \wedge \langle t, (f, b) \rangle C_c)$ is consistent, which can be rewritten as $\bigwedge A \rightarrow [t, (f, b)] C_c$.

Because $C_c \wedge \langle t, (f, b) \rangle \neg C_c$ is inconsistent, its negation $\neg(C_c \wedge \langle t, (f, b) \rangle \neg C_c)$ is valid, which can be rewritten to $\vdash C_c \rightarrow [t, (f, b)]C_c$ (I). Therefore, by applying generalization we have $\vdash [t, (f, b)^*](C_c \rightarrow [t, (f, b)]C_c)$. By axiom **(It)**, we derive $\vdash [t, (f, b)]C_c \rightarrow [t, (f, b)^*]C_c$ (II). By rewriting (II) in (I) we derive $C_c \rightarrow [t, (f, b)^*]C_c$. As $\bigwedge A \rightarrow [t, (f, b)]C_c$ is valid, from (II) $\bigwedge A \rightarrow [t, (f, b)^*]C_c$ also is valid. From the hypothesis $AS_{\pi^*}B$ and C_c 's definition, $\bigwedge A \wedge \langle t, (f, b)^* \rangle B$ and $\bigwedge B \wedge C_c$ are consistent (the latter from C_c 's definition). Then, there is a $C_i \in C_c$ such that $\bigwedge B \wedge \bigwedge C$ is consistent. But because each C_i is a maximal consistent set, it is the case that $B = C_i$, which by the definition of C_c leads to AS_{Π}^*B .

□

Definition 5.3.6 (Proper Canonical Model). *The proper canonical model over a set of formulae Γ is defined as a tuple $\langle At(\Gamma), \Pi, R_{\Pi}^{\Gamma}, \delta_{\Pi}^{\Gamma}, \lambda_c, V_{\Pi}^{\Gamma} \rangle$ as follows:*

- $At(\Gamma)$ as the set of atoms of Γ ;
- Π as the ReLo program;
- The relation R of a ReLo program Π is inductively defined as:
 - $R_{\pi} = S_{\pi}$ for each canonical program π ;
 - $R_{\Pi^*}^{\Gamma} = (R_{\Pi}^{\Gamma})^*$;
 - $\Pi = \pi_1 \odot \pi_2 \odot \cdots \odot \pi_n$ a ReLo program, $R_{\Pi} \subseteq S \times S$ as follows:
 - * $R_{\pi_i} = \{uR_{\pi_i}v \mid f_{ReLo}(t, \pi_i) \prec \delta(v)\}$, $t \prec \delta(u)$ and π_i is any combination of any atomic programs which is a subprogram of Π .
- δ_{Π}^{Γ} as the canonical markup function;
- $\lambda_c: At(\Gamma) \times \mathcal{N} \rightarrow \mathbb{R}$;
- $V_c^{\Gamma}(A, p) = \{A \in At(\Gamma) \mid p \in A\}$ as the canonical valuation introduced by Definition 5.3.5.

Lemma 10. Every canonical model for Π has a corresponding proper canonical model: for all programs Π , $S_{\Pi}^{\Gamma} \subseteq R_{\Pi}^{\Gamma}$

Proof. The proof proceeds by induction on Π 's length

- For basic programs π , it follows from Definition 5.3.6:

- Π^* : From Definition 5.1.8, $R_{\pi^*} = R_{\pi}^*$. By the induction hypothesis, $S_{\Pi}^{\Gamma} \subseteq R_{\Pi}^{\Gamma}$. Also from the definition of RTC, we have that if $(S_{\Pi}^{\Gamma}) \subseteq (R_{\Pi}^{\Gamma})$, then $(S_{\Pi}^{\Gamma})^* \subseteq (R_{\Pi}^{\Gamma})^*$.
 (i). From Lemma 9, $S_{\Pi^*}^{\Gamma} \subseteq (S_{\Pi}^{\Gamma})^*$, which leads to $(S_{\Pi}^{\Gamma})^* \subseteq (R_{\Pi}^{\Gamma})^*$ by (i). Finally, $(R_{\Pi}^{\Gamma})^* = (R_{\Pi^*}^{\Gamma})$. Hence, $(S_{\Pi^*}^{\Gamma}) \subseteq (R_{\Pi^*}^{\Gamma})$

□

Lemma 11 (Existence Lemma for Proper Canonical Models). Let $A \in At(\Gamma)$ and $\langle t, (f, b) \rangle \varphi \in FL(\Gamma)$. Then,

$$\langle t, (f, b) \rangle \varphi \in A \leftrightarrow \text{exists } B \in At(\Gamma), AR_{\Pi}^{\Gamma} B, t \prec \delta_c^{\Gamma}(A) \text{ and } \varphi \in B.$$

Proof.

\Rightarrow

Let $\langle t, (f, b) \rangle \varphi \in A$. From Lemma 7 (Existence Lemma for canonical models), There is an atom $B \in At(\Gamma)$ where $AS_{\Pi}^{\Gamma} B, t \prec \delta_c^{\Gamma}(A)$ and $\varphi \in B$. From Lemma 10, $S_{\Pi}^{\Gamma} \subseteq R_{\Pi}^{\Gamma}$. Therefore, there is an atom $B \in At(\Gamma)$ where $AR_{\Pi}^{\Gamma} B, t \prec \delta_c^{\Gamma}(A)$ and $\varphi \in B$.

\Leftarrow

Let B an atom, $B \in At(\Gamma), AR_{\Pi}^{\Gamma} B, t \prec \delta_c^{\Gamma}(A)$ and $\varphi \in B$. The proof follows by induction on the program $\Pi = (f, b)$ as follows:

- a canonical program π_i : this case is straightforward as from Definition 5.3.6, $S_{\pi_i} = R_{\pi_i}$, and consequently $AS_{\pi_i} B, t \prec \delta_c^{\Gamma}(A)$ and (i) $\varphi \in B$. From Lemma 7 and (i), $\langle t, (f, b) \rangle \varphi \in A$.
- Π^* : from Definition 5.3.6, $R_{\Pi^*} = R_{\Pi}^*$. Then, let $B \in At(\Gamma), AR_{\Pi^*}^{\Gamma} B, t \prec \delta_c^{\Gamma}(A)$ and $\varphi \in B$. This means that there is a finite nondeterministic number n where $AR_{\Pi^*}^{\Gamma} B = AR_{\Pi}^{\Gamma} A_1 R_{\Pi}^{\Gamma} A_2 \dots R_{\Pi}^{\Gamma} A_n$, where $A_n = B$. The proof proceeds by induction on n :
 - $n = 1$: $AR_{\Pi}^{\Gamma} B$ and $\varphi \in B$. Therefore, from Lemma 7, $\langle t, (f, b) \rangle \varphi \in A$. From axiom Rec, one may derive $\vdash \langle t, (f, b) \rangle \varphi \rightarrow \langle t, (f, b)^* \rangle \varphi$. By the definition of FL and A 's maximality (as it is an atom of Γ) $\langle t, (f, b)^* \rangle \varphi \in A$.
 - $n > 1$: From the previous proof step and the induction hypothesis, $\langle t, (f, b)^* \rangle \in A_2$ and $\langle t, (f, b) \rangle \langle t, (f, b)^* \rangle \in A_1$. From axiom Rec, one can derive $\vdash \langle t, (f, b) \rangle \langle t, (f, b)^* \rangle \varphi \rightarrow \langle t, (f, b)^* \rangle \varphi$. By the definition of FL, and A 's maximality (as it is an atom of Γ), $\langle t, (f, b)^* \rangle \varphi \in A$.

□

Lemma 12 (Truth Lemma for Proper Canonical Models). Let $\mathcal{M}_c^\Gamma = \langle At(\Gamma), \Pi, R_\Pi^\Gamma, \delta_\Pi^\Gamma, \lambda_c, V_\Pi^\Gamma \rangle$ a proper canonical model constructed over a formula γ . For all atoms A and all $\varphi \in FL(\gamma)$. $\mathcal{M}, A \models \varphi \leftrightarrow \varphi \in A$.

Proof. The proof proceeds by induction over φ .

- induction basis: φ is a proposition p . Therefore, $\mathcal{M}_c^\Gamma, A \models p$ holds from Definition 5.3.6 as $V_c^\Gamma(p) = \{A \in At(\Gamma) \mid p \in A\}$.
- induction hypothesis: suppose φ is a non atomic formula ψ . Then, $\mathcal{M}, A \models \varphi \iff \varphi \in A$, ψ a strict subformula of φ .
- Inductive step: Let us prove it holds for the following cases:
 - Case $\varphi = \neg\psi$: by the induction hypothesis, $\mathcal{M}_c^\Gamma, A \models \neg\psi \iff \neg\psi \in A$. Because A is maximal w.r.t. ϕ , then $\neg\phi \in A$ holds by \mathcal{M}_c^Γ 's definition.
 - Case $\varphi = \psi_1 \wedge \psi_2$: by the induction hypothesis, $\mathcal{M}_c^\Gamma, A \models \psi_1 \wedge \psi_2 \iff \psi_1 \in A$ and $\psi_2 \in A$, which holds by \mathcal{M}_c^Γ 's definition.
 - Other connectives ($\vee, \rightarrow, \leftrightarrow$): their proof may be derived by following the ideas presented in the bullets above.
 - Case $\varphi = \langle t, (f, b) \rangle \phi$. Then, $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b) \rangle \phi \iff \langle t, (f, b) \rangle \phi \in A$:
 - \rightarrow
Let $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b) \rangle \phi$. From Definition 5.3.3, there is an atom B where $AS_\Pi^\Gamma B$ and $\phi \in B$. By Lemma 11, $\langle t, (f, b) \rangle \phi \in A$. Therefore, it holds.
 - \leftarrow
Let $\mathcal{M}_c^\Gamma, A \not\models \langle t, (f, b) \rangle \phi$. From Definition 5.3.5's valuation function V_c^Γ and Lemma 5, we have $\mathcal{M}_c^\Gamma, A \models \neg \langle t, (f, b) \rangle \phi$. Therefore, for every B where $AS_\Pi^\Gamma B$, $\mathcal{M}_c^\Gamma, B \models \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, from Lemma 11 $\langle t, (f, b) \rangle \phi \notin A$.
 - Case $\varphi = \langle t, (f, b)^* \rangle \phi$. Then, $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b)^* \rangle \phi \iff \langle t, (f, b)^* \rangle \phi \in A$:
 - \rightarrow
Let $\mathcal{M}_c^\Gamma, A \models \langle t, (f, b)^* \rangle \phi$. From Definition 5.3.3, there is a state B where $AS_{\Pi^*}^\Gamma B$ and $\phi \in B$. By Lemma 7, $\langle t, (f, b)^* \rangle \phi \in A$. Therefore, it holds.
 - \leftarrow
Let $\mathcal{M}_c^\Gamma, A \not\models \langle t, (f, b)^* \rangle \phi$. From Definition 5.3.5's valuation function V_c^Γ and Lemma 5, we have $\mathcal{M}_c^\Gamma, A \models \neg \langle t, (f, b)^* \rangle \phi$. Therefore, for every B where $AS_{\Pi^*}^\Gamma B$, $\mathcal{M}_c^\Gamma, B \models \neg\phi$. From the induction hypothesis, $\phi \notin B$. Hence, From Lemma 7, $\langle t, (f, b)^* \rangle \phi \notin A$.

□

Theorem 1 (Completeness of *ReLo*). The logic *ReLo* is complete with respect to the class of proper canonical models.

Proof. For every consistent formula A , a canonical model \mathcal{M} can be constructed. From Lemma 5, there is an atom $A' \in At(A)$ with $A \in A'$, and from Lemma 12, $\mathcal{M}, A' \models A$. Therefore, *ReLo*'s modal system is complete with respect to the class of proper canonical models as Definition 5.3.6 proposes. □

5.4 A Tableau for *ReLo*

In this section we propose a tableau method for *ReLo* providing a syntactic proof procedure similar to what other dynamic logics propose [27, 28, 73]. The definitions of tableau and branch are standard in the literature [28, 60, 73] and are as follows.

Definition 5.4.1 (Tableau in *ReLo*). A tableau \mathcal{T} in *ReLo* is a rooted tree with nodes labeled with formulae, prefixed by states in an auxiliary finite sequence of states $S_{\mathcal{T}}$ which guides the decomposition of formulae. A branch \mathcal{B} is a path from the root to a leaf node. A segment \mathcal{S} is a path from the root node to some intermediate node (i.e., a non-leaf node).

Intuitively, a tableau in *ReLo* of a formula $\varphi : T$ denotes a failed attempt to prove $\varphi : F$, which in turn yields φ as true. Branches of the tableau can be interpreted as a tentative of the construction of a model that holds for the initial formula $\varphi : F$, and segments as the intermediate steps taken to construct such model. Definition 5.4.2 formalizes the rules that may be applied to formulas in *ReLo*'s Tableau.

Definition 5.4.2. *Tableau rules for ReLo*

Let w, x denote states, φ a *ReLo* formula, and $\pi = (f, b)$ a *ReLo* program. The rules valid for *ReLo* Tableau are as follows. In what follows, let t'_π be the result of $f_{ReLo}(t, \pi)$ and $\delta(w)$ the data markup function of a state w .

- Propositional rules

$$\begin{array}{ll}
 (\textbf{And-T}) \quad \frac{w: \varphi \wedge \psi : T}{\begin{array}{l} w: \varphi : T \\ w: \psi : T \end{array}} & (\textbf{And-F}) \quad \frac{w: \varphi \wedge \psi : F}{\begin{array}{l} w: \varphi : F \\ w: \psi : F \end{array}} \\
 (\textbf{Or-T}) \quad \frac{w: \varphi \vee \psi : T}{\begin{array}{l} w: \varphi : T \\ w: \psi : T \end{array}} & (\textbf{Or-F}) \quad \frac{w: \varphi \vee \psi : F}{\begin{array}{l} w: \varphi : F \\ w: \psi : F \end{array}}
 \end{array}$$

$$\begin{array}{ll}
(\textit{Neg-T}) \quad \frac{w : \neg\varphi : T}{w : \varphi : F} & (\textit{Neg-F}) \quad \frac{w : \neg\varphi : F}{w : \varphi : T} \\
(\rightarrow -T) \quad \frac{w : \varphi \rightarrow \psi : T}{w : \varphi : F \quad w : \psi : T} & (\rightarrow -F) \quad \frac{w : \varphi \rightarrow \psi : F}{w : \varphi : T \quad w : \psi : F}
\end{array}$$

• *Modal rules*

$$\begin{array}{ll}
(\langle t, \pi \rangle - T) \quad \frac{w : \langle t, \pi \rangle \varphi : T}{x : \varphi : T} & (\langle t, \pi \rangle - F) \quad \frac{w : \langle t, \pi \rangle \varphi : F}{x : \varphi : F} \\
x \text{ is a new state in the sequence of states } S_{\mathcal{T}} \text{ accessible from } w. & x \text{ is any state (new or already existing) in the sequence of states } S_{\mathcal{T}} \text{ with } x \text{ accessible from } w \text{ by } \pi.
\end{array}$$

$$\begin{array}{ll}
([t, \pi] - T) \quad \frac{w : [t, \pi] \varphi : T}{x : \varphi : T} & ([t, \pi] - F) \quad \frac{w : [t, \pi] \varphi : F}{x : \varphi : F} \\
x \text{ is any state (new or already existing) in the sequence of states } S_{\mathcal{T}} \text{ with } x \text{ accessible from } w \text{ by } \pi. & x \text{ is a new state in the sequence of states } S_{\mathcal{T}} \text{ accessible from } w.
\end{array}$$

$$\begin{array}{l}
(\langle t, \pi \rangle_{(\mathcal{R})} - T) \quad \frac{w : \langle t, \pi \rangle \varphi : T, \text{ iff } f_{\text{ReLo}}(t, \pi) = \epsilon}{w : \varphi : T} \\
w \text{ the same state. This rule captures the behavior of axiom } \mathcal{R}. \text{ Its non validity is the normal case for diamond formulas.}
\end{array}$$

The rules for the iteration operator combine prefixed tableau with some ideas from [76] to allow reasoning over iteration. A formula $\langle t, \pi^* \rangle \varphi$ is defined as an eventuality. Whenever an eventuality is found, we introduce it as a fresh (possibly indexed) propositional symbol \mathcal{X}_{\Diamond} where $\mathcal{X}_{\Diamond} \leftrightarrow \langle t, \pi^* \rangle \varphi$. The indexation of these variables is required to identify different eventualities in the proof process. The objective of introducing such variables is to enable the detection of unsuccessful loops where $\langle t, \pi \rangle \varphi$ is not fulfilled (as in Definition 5.4.3).

The introduction of such propositional symbols requires a distinct set \mathcal{X} for them to differ from the set Φ of propositional letters used for formulae. Note that this is also valid for eventualities $[t, \pi^*] \varphi$, in which case the fresh symbol introduced is \mathcal{X}_{\Box} .

• *Program operator rules*

– *Iteration*

$$\begin{array}{c}
\begin{array}{c}
(\langle t, \pi^* \rangle) - T \quad \frac{w: \langle t, \pi^* \rangle \varphi : T}{w: \mathcal{X}_\Diamond : T} \\
\mathcal{X}_\Diamond \leftrightarrow \langle t, \pi^* \rangle \varphi
\end{array}
\qquad
\begin{array}{c}
(\langle t, \pi^* \rangle) - F \quad \frac{w: \langle t, \pi^* \rangle \varphi : F}{w: \varphi : F} \\
\frac{w: \langle t, \pi \rangle \langle t'_\pi, \pi^* \rangle \varphi : F}{\text{The data sequence } t \text{ is a sequence} \\ \text{such that } t \subseteq \delta(w) \text{ and } t'_\pi \prec \\ f_{ReLo}(t, \pi).}
\end{array}
\\[10pt]
\begin{array}{c}
([t, \pi^*]) - T \quad \frac{w: [t, \pi^*] \varphi : T}{w: \varphi : T} \\
\frac{w: [t, \pi][t'_\pi, \pi^*] \varphi : T}{\text{The data sequence } t'_\pi \text{ is a sequence} \\ \text{such that } t \subseteq \delta(w) \text{ and } t'_\pi \prec \\ f_{ReLo}(t, \pi).}
\end{array}
\qquad
\begin{array}{c}
([t, \pi^*]) - F \quad \frac{w: [t, \pi^*] \varphi : F}{w: \mathcal{X}_\Box : F} \\
\mathcal{X}_\Box \leftrightarrow [t, \pi^*] \varphi
\end{array}
\\[10pt]
\begin{array}{c}
(\mathcal{X}_\Diamond) \quad \frac{w: \mathcal{X}_\Diamond : T}{w: \varphi : T \quad w: \langle t, \pi \rangle \mathcal{X}_\Diamond : T}
\end{array}
\qquad
\begin{array}{c}
(\mathcal{X}_\Box) \quad \frac{w: \mathcal{X}_\Box : F}{w: \varphi : F \quad w: [t, \pi] \mathcal{X}_\Box : F}
\end{array}
\end{array}$$

The iteration rules introduced by \mathcal{X}_\Diamond rules can be intuitively detailed as follows: for formulas $\langle t, \pi^* \rangle \varphi$, the reduction of the modality employing \star is done by stating that φ is either valid in the reached state, or we may reduce it once again in another state where φ is valid. The rule \mathcal{X}_\Diamond denote that

In what follows some useful definitions regarding notions of iteration in the tableau are discussed. We follow the methodology proposed by [28].

A segment \mathcal{S} is denoted by a path between two different states in the Tableau. A set of prefixed formulae of a segment \mathcal{S} is the set of all formulae φ in \mathcal{S} that are prefixed by some state w as

$$\mathcal{S}/w = \{\varphi \mid w : \varphi \in \mathcal{S}\}$$

A prefix w is said to be reduced in a segment \mathcal{S} if the modal rules are the only rules not applied yet to the set \mathcal{S}/w . It is fully reduced if all possible rules have been applied.

Intuitively, the reduction states that all iteration rules have been applied in formulae in \mathcal{S}/w (if possible), while the full reduction yields formulae with all transitional (i.e., rules that may change a state) rules applied.

A branch \mathcal{B} is said to be π -completed if all prefixes are reduced, and for every w which is not fully reduced, there is a segment \mathcal{S} and a prefix w' of \mathcal{S} smaller than \mathcal{B} and w which is fully reduced and w' is a copy of w in \mathcal{B} .

A prefix w' of a segment \mathcal{S}' is said to be a copy of another prefix w in a segment \mathcal{S}

if $\mathcal{S}/w = S'/w'$ and they have the same transitions format given by the auxiliary tree of states.

Definition 5.4.3 (fulfilled \mathcal{X}). *An eventuality \mathcal{X} is said to be fulfilled if there is a labeled formula $w : \varphi : T$ in the same segment with $w : \mathcal{X}_\Diamond$ (satisfying it), or \mathcal{X}_\Diamond collapses to a \mathcal{X}_{\Diamond_0} which has already appeared in the branch and is fulfilled. For the case of \mathcal{X}_\Box , it is fulfilled if there is a $w : \varphi : F$ in the same segment with $w : \mathcal{X}_\Box$, or \mathcal{X}_\Box collapses to a \mathcal{X}_{\Box_0} which has already appeared in the branch and is fulfilled.*

An eventuality \mathcal{X}_\Box cannot be fulfilled and collapse to itself at the same time in another state v previously visited because it generates both $\varphi : T$ and $\varphi : F$ from the rule application. Suppose $w : \mathcal{X}_\Box : T$ collapses to a shorter $v : \mathcal{X}_\Box : T$. If it fulfilled \mathcal{X}_\Box , then \mathcal{X}_\Box in state w would not exist, as it would have already been fulfilled in a earlier derivation. The same is also valid for eventualities \mathcal{X}_\Diamond , following the same idea.

Summarizing, this can be interpreted as a unsuccessful loop, in which we try to fulfill eventualities \mathcal{X}_\Box or \mathcal{X}_\Diamond by applying their corresponding rules: the left hand side of the rule is always discarded, with no success on fulfilling the eventualities generated in their right hand side. At some point during the derivation, there will be a state ω with formulae as the same of a previously visited state (i.e., a copy) v where $\mathcal{S}/\omega = S'/v$. One may conclude that this will not end, and the loop will never succeed to close. This is captured in Definition 5.4.4 as an ignorable branch.

Definition 5.4.4. *A branch \mathcal{B} is ignorable if it contains a \mathcal{X} introduced by any of the \mathcal{X} – rules which collapses to itself (either directly or transitively) and its corresponding shorter branch (the left branch resulting from \mathcal{X} rule’s application) is closed.*

The idea Definition 5.4.4 presents is to denote that the looping introduced by \mathcal{X} rules may not close, yielding a unsuccessful loop by always iterating and leaving the rightmost branch “open”. If at some point the proof reaches a \mathcal{X}_i with the same formulae as an already visited state, we can conclude that it will never be fulfilled and therefore ignore the remainder of the branch, as it has already been reduced earlier in the proof and it may introduce a potentially infinite pattern in the formula decomposition process. With Definition 5.4.4, we may define the closure of *ReLo* tableau as follows.

Definition 5.4.5 (Tableau contradiction). *A contradiction in a branch \mathcal{B} of a *ReLo* tableau is defined following the notion of “parent contradiction”: \mathcal{B} is contradictory if there is a state w and a formulae φ , such that $(w : \varphi : T)$ and $(w : \varphi : F)$ are in \mathcal{B} .*

Definition 5.4.6 (Tableau closure). *A tableau \mathcal{T} is closed if all of its branches are either contradictory or ignorable, yielding that the formula syntactically holds in *ReLo*. Conversely, a tableau \mathcal{T} is open if there is at least a branch with no contradictions or it is not ignorable.*

5.4.1 Tableau Usage Examples

In what follows we provide usage examples of the tableau developed for *ReLo*. We show its usage for the axioms defined in Section 5.1.11, stating that they are indeed valid in the proposed Tableau. The rules applied are shown in the rightmost part of the proof, where t and π in the names of rules with modalities are omitted.

- \mathcal{K} : $[t, (f, b)](\varphi \rightarrow \psi) \rightarrow ([t, (f, b)]\varphi \rightarrow [t, (f, b)]\psi)$

| | | |
|----|---|---------------------|
| 1. | $w : [t, (f, b)](\varphi \rightarrow \psi) \rightarrow ([t, (f, b)]\varphi \rightarrow [t, (f, b)]\psi) : F$ | |
| 2. | $w : [t, (f, b)](\varphi \rightarrow \psi) : T$ | 1, $\rightarrow -F$ |
| 3. | $w : ([t, (f, b)]\varphi \rightarrow [t, (f, b)]\psi) : F$ | 1, $\rightarrow -F$ |
| 4. | $w : [t, (f, b)]\varphi : T$ | 3, $\rightarrow -F$ |
| 5. | $w : [t, (f, b)]\psi : F$ | 3, $\rightarrow -F$ |
| 6. | $x : \psi : F$ | 5, $\Box - F$ |
| 7. | $x : \varphi \rightarrow \psi : T$ | 2, $\Box - T$ |
| | <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> $x : \varphi : F$ </div> <div style="text-align: center;"> $x : \psi : T$ </div> </div> | |
| 8. | $x : \varphi : F$ | 5, $\rightarrow -T$ |
| 9. | $x : \varphi : T$ | 4, $\Box - T$ |
| | <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> \times 8, 9 </div> <div style="text-align: center;"> \times 6, 8 </div> </div> | |

- $\text{And} \rightarrow$: $[t, (f, b)](\varphi \wedge \psi) \rightarrow ([t, (f, b)]\varphi \wedge [t, (f, b)]\psi)$

| | | |
|----|--|---------------------|
| 1. | $w : [t, (f, b)](\varphi \wedge \psi) \rightarrow ([t, (f, b)]\varphi \wedge [t, (f, b)]\psi) : F$ | |
| 2. | $w : [t, (f, b)](\varphi \wedge \psi) : T$ | 1, $\rightarrow -F$ |
| 3. | $w : ([t, (f, b)]\varphi \wedge [t, (f, b)]\psi) : F$ | 1, $\rightarrow -F$ |
| | <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> $w : [t, (f, b)]\varphi : F$ </div> <div style="text-align: center;"> $w : [t, (f, b)]\psi : F$ </div> </div> | |
| 4. | $x : \varphi : F$ | 3, $\wedge - F$ |
| 5. | $y : \psi : F$ | 4, $\Box - F$ |
| 6. | $x : \varphi \wedge \psi : T$ | 2, $\Box - T$ |
| 7. | $x : \psi : T$ | 6, $\wedge - T$ |
| 8. | $x : \varphi : T$ | 6, $\wedge - T$ |
| | <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> \times 5, 8 </div> <div style="text-align: center;"> \times 5, 8 </div> </div> | |

- And- \leftarrow : $([t, (f, b)]\varphi \wedge [t, (f, b)]\psi) \rightarrow [t, (f, b)](\varphi \wedge \psi)$

| | | |
|--|--|------------------------|
| 1. | $w : ([t, (f, b)]\varphi \wedge [t, (f, b)]\psi) \rightarrow [t, (f, b)](\varphi \wedge \psi) : F$ | |
| 2. | $w : ([t, (f, b)]\varphi \wedge [t, (f, b)]\psi) : T$ | $1, \rightarrow -F$ |
| 3. | $w : [t, (f, b)](\varphi \wedge \psi) : F$ | $1, \rightarrow -F$ |
| 4. | $w : [t, (f, b)]\varphi : T$ | $2, \wedge -T$ |
| 5. | $w : [t, (f, b)]\psi : T$ | $2, \wedge -T$ |
| 6. | $x : \varphi \wedge \psi : F$ | $3, [] - F$ |
| $\begin{array}{c} \swarrow \quad \searrow \\ x : \varphi : F \quad x : \psi : F \end{array}$ | | |
| 7. | $x : \varphi : F$ | $6, \wedge -F$ |
| 8. | $x : \varphi : T$ | $4, [] - T; 5, [] - T$ |
| | \times | \times |
| | 7,8 | 7,8 |

- $R \rightarrow$: $\langle t, (f, b) \rangle \varphi \rightarrow \varphi, \text{ if } f_{ReLo}(t, (f, b)) = \epsilon$

| | | |
|----|--|---------------------|
| 1. | $w : \langle t, (f, b) \rangle \varphi \rightarrow \varphi, \text{ if } f_{ReLo}(t, (f, b)) : F$ | |
| 2. | $w : \langle t, (f, b) \rangle \varphi : T$ | $1, \rightarrow -F$ |
| 3. | $w : \varphi : F$ | $1, \rightarrow -F$ |
| 4. | $w : \varphi : T$ | $2, <> -R - T$ |
| | \times | |
| | 3,4 | |

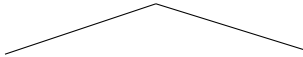
- $R \leftarrow$: $\varphi \rightarrow \langle t, (f, b) \rangle \varphi, \text{ if } f_{ReLo}(t, (f, b)) = \epsilon$

| | | |
|----|--|---------------------|
| 1. | $w : \varphi \rightarrow \langle t, (f, b) \rangle \varphi, \text{ if } f_{ReLo}(t, (f, b)) : F$ | |
| 2. | $w : \varphi : T$ | $1, \rightarrow -F$ |
| 3. | $w : \langle t, (f, b) \rangle \varphi : F$ | $1, \rightarrow -F$ |
| 4. | $w : \varphi : F$ | $2, <> -R - T$ |
| | \times | |
| | 2,4 | |

- It \rightarrow : $\varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi \rightarrow [t, (f,b)^*]\varphi$

| | | |
|----|--|---------------------|
| 1. | $w : \varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi \rightarrow [t, (f,b)^*]\varphi : F$ | |
| 2. | $w : \varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : T$ | $1, \rightarrow -F$ |
| 3. | $w : [t, (f,b)^*]\varphi : F$ | $1, \rightarrow -F$ |
| 4. | $w : \varphi : T$ | $2, \wedge - T$ |
| 5. | $w : [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : T$ | $2, \wedge - T$ |
| 6. | $w : [t, (f,b)^*]\varphi : T$ | $5, [] - T$ |
| | \times 3,6 | |

- It \leftarrow : $[t, (f,b)^*]\varphi \rightarrow \varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi$

| | | |
|----|--|--------------------------|
| 1. | $w : [t, (f,b)^*]\varphi \rightarrow \varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : F$ | |
| 2. | $w : [t, (f,b)^*]\varphi : T$ | $1, \rightarrow -F$ |
| 3. | $w : \varphi \wedge [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : F$ | $1, \rightarrow -F$ |
| |  | |
| 4. | $w : \varphi : F \quad w : [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : F$ | $3, \wedge - F$ |
| 5. | $w : \varphi : T \quad x : [t_{(f,b)}, (f,b)^*]\varphi : F$ | $2, []^* - T; 4, [] - F$ |
| 6. | $w : [t, (f,b)][t_{(f,b)}, (f,b)^*]\varphi : T \quad w : \varphi : T$ | $2, []^* - T$ |
| 7. | \times 4,5 | $2, []^* - T$ |
| 8. | $x : [t_{(f,b)}, (f,b)^*]\varphi : T$ | $7, [] - T$ |
| | \times 5,7 | |

| | | |
|--------|---|-----------------------|
| • Ind: | $\varphi \wedge [t, (f, b)^*](\varphi \rightarrow [t_{(f,b)^*}, (f, b)]\varphi) \rightarrow [t, (f, b)^*]\varphi$ | |
| 1. | $w : \varphi \wedge [t, (f, b)^*](\varphi \rightarrow [t_{(f,b)}, (f, b)]\varphi) \rightarrow [t, (f, b)^*]\varphi : F$ | |
| 2. | $w : \varphi \wedge [t, (f, b)^*](\varphi \rightarrow [t_{(f,b)}, (f, b)]\varphi) : T$ | 1, $\rightarrow - F$ |
| 3. | $w : [t, (f, b)^*]\varphi : F$ | 1, $\rightarrow - F$ |
| 4. | $w : \varphi : T$ | 2, $\wedge - T$ |
| 5. | $w : [t, (f, b)^*](\varphi \rightarrow [t_{(f,b)}, (f, b)]\varphi) : T$ | 2, $\wedge - T$ |
| 6. | $w : \varphi : F$ | 3, $[]^* - F$ |
| 7. | $w : [t_{(f,b)}, (f, b)]\mathcal{X}_{[]} : F$ | 3, $[]^* - F$ |
| 8. | $x : \mathcal{X}_{[]} : F$ | 7, $[] - T$ |
| 9. | $x : \varphi \rightarrow [t_{(f,b)}, (f, b)]\varphi : T$ | 5, $[] - T$ |
| 10. | $x : [t, (f, b)^*](\varphi \rightarrow [t_{(f,b)}, (f, b)]\varphi) : T$ | 5, $[] - T$ |
| 11. | $x : \varphi : F$ | 9, $\rightarrow - T$ |
| 12. | $x : \varphi : F$ | 8, $\mathcal{X}_{[]}$ |
| 13. | $x : [t_{(f,b)}, (f, b)]\mathcal{X}_{[]} : F$ | 8, $\mathcal{X}_{[]}$ |
| 14. | $y : \mathcal{X}_{[]} : F$ | 13, $[] - F$ |
| | \times Ignorable, 14 = 8 | |

5.4.2 Termination

Due to the presence of iteration rules in *ReLo*'s Tableau, it is not terminating as these rules may lead to a infinite number of derivations. Therefore, We define a search strategy for *ReLo* in which the tableau process has termination (i.e., after tableau rule applications, the tableau will eventually stop). The interesting cases here are introduced by rules regarding the \star operator, which may allow a nondeterministic finite number of derivations, which can lead to an infinite tableau.

Lemma 13 (Termination of *ReLo*'s Tableau). The proposed tableau is terminating.

Proof. The proof proceeds by induction on *ReLo*'s tableau rules. The idea of the proof is to show that every rule defined for the tableau reaches a scenario that cannot be further reduced or it does not bring any advantage to do so (which is the case when considering \star rules). From Lemmas 14 and 16, tableau proofs for a formula φ will either result on a closed tableau for $\neg\varphi$, to which φ is satisfiable in *ReLo* or it will result on an open tableau, to which the formula φ is not satisfiable in *ReLo*, and a counter-model can be derived. To

proceed with such induction, we first define (inductively) a systematic procedure which we define the application of rules as follows based on standard arguments in literature [31]:

1. A tableau for φ is a tableau \mathcal{T} initiating with $(w : \varphi : F)$ as the root of the proof tree.
2. The order of the main operators of formulae to be reduced in Steps 3. and 4. must follow the following sequence, first to last from top to bottom:
 - (a) propositional main operators $\{\wedge, \vee, \neg, \rightarrow\}$
 - (b) modal main operators $[t, (f, b)], \langle t, (f, b) \rangle$
 - (c) iteration main operators $[t, (f, b)^*], \langle t, (f, b)^* \rangle, \mathcal{X}_{\Diamond}, \mathcal{X}_{\Box}$
3. If after n applications of tableau expansion rules to formulae in \mathcal{T} as in Step 2 the result is a closed tableau, stop and return \mathcal{T} as a closed tableau.
4. If Step 3 did not yield a closed tableau, choose the formula $k : \psi : \sigma$ closest to the root node in the leftmost branch¹ which has not been reduced. If ψ is a atomic formula, then this step ends. Otherwise, for all open branches \mathcal{B} that contain $(k : \psi : \sigma)$:
 - if ψ is of form $\psi_1 \wedge \psi_2$ and $\sigma = T$, then add $k : \psi_1 : T$ and $k : \psi_2 : T$ at the end of \mathcal{B} . If $\sigma = F$, then add $k : \psi_1 : F$ and $k : \psi_2 : F$ splitting the end of \mathcal{B} , resulting in two new branches from \mathcal{B} , one ending with $k : \psi_1 : T$ and other ending with $k : \psi_2 : T$.
 - if ψ is of form $\psi_1 \vee \psi_2$ and $\sigma = T$, then add $k : \psi_1 : T$ and $k : \psi_2 : T$ splitting the end of \mathcal{B} , resulting in two new branches from \mathcal{B} , one ending with $k : \psi_1 : T$ and other ending with $k : \psi_2 : T$. If $\sigma = F$, then add $k : \psi_1 : F$ and $k : \psi_2 : F$ at the end of \mathcal{B} .
 - if ψ is of form $\neg\psi_1$ and $\sigma = T$, then add $k : \psi_1 : F$ at the end of \mathcal{B} . If $\sigma = F$, then add $k : \psi_1 : T$ at the end of \mathcal{B} .
 - if ψ is of form $\psi_1 \rightarrow \psi_2$, it may be rewritten as $(\neg\psi_1) \vee \psi_2$, therefore it reduces to the case of $\psi_1 \vee \psi_2$.
 - if ψ is of form $\langle t, (f, b) \rangle \psi_1$ and $\sigma = T$, then add $k' : \psi_1 : T$ as a new state which has not been visited yet (i.e., it does not identify a prefix in the current tableau), k' as the next available prefix. If $\sigma = F$, then for each accessible state

¹As this may result in a nondeterministic choice (i.e., more than one unfinished formulae in the same tree level), we focus on the leftmost occurrence.

- k' from k which has already been visited, add $k' : \psi_1 : F$ to the end of each branch.
- if ψ is of form $[t, (f, b)]\psi_1$ and $\sigma = T$, then for each accessible state k' from k which has already been visited, add $k' : \psi_1 : T$ to the end of each branch. If $\sigma = F$, then add $k' : \psi_1 : F$ as a new state which has not been visited yet (i.e., it does not identify a prefix in the current tableau), k' as the next available prefix.
 - if ψ is of form $\langle t, (f, b)^* \rangle \psi_1$ and $\sigma = T$, introduce $k : \mathcal{X}_{\Diamond_i} : T$, where $\mathcal{X}_{\Diamond_i} = \langle t, (f, b)^* \rangle \psi_1$ at the end of \mathcal{B} , and check whether there is a state k_0 in which every formula of k_0 is true in k (and vice-versa), and whether $k_0 : \mathcal{X}_{\Diamond_k} : T$ holds. If this holds, then \mathcal{X}_{\Diamond_i} collapses into \mathcal{X}_{\Diamond_k} , and k is a copy of k_0 . Therefore, \mathcal{B} is a ignorable branch and ψ should not be reduced further. If $\sigma = F$, then add both $k : \psi_1 : F$ and $k : \langle t, \pi \rangle \langle t'_\pi, \pi^* \rangle \psi_1 : F$ at the end of \mathcal{B} . Then, add $k : \langle t'_\pi, \pi^* \rangle \psi_1 : F$ at the end of \mathcal{B} , and check if exists a state k_0 in which every formula of k_0 is true in k (and vice-versa). If it does, then this configuration has already been considered in the proof and this branch is an ignorable branch.
 - if ψ is of form \mathcal{X}_{\Diamond} , then at the end of \mathcal{B} add $k : \psi_1 : T$ at the left resulting branch \mathcal{B}_1 , and $k : \psi_1 : F$ and $k : \langle t, \pi \rangle \mathcal{X}_{\Diamond_k} : T$ at the right resulting branch \mathcal{B}_2 , splitting it in two new branches from \mathcal{B} . Now, add $k' : \mathcal{X}_{\Diamond_k} : T$ at the end of \mathcal{B}_2 . Check whether there is a state k_0 in which every formula of k_0 is true in k' (and vice versa), and there is an eventuality $\mathcal{X}_{\Diamond_{k_0}} = \langle t, (f, b)^* \rangle \psi_1 = \mathcal{X}_{\Diamond_k}'$. If this holds, then $\mathcal{X}_{\Diamond_k}'$ collapses into $\mathcal{X}_{\Diamond_{k_0}}$, and k' is a copy of k_0 . The left resulting branch \mathcal{B}_1 is closed because there is a $k_0 : \psi_1 : F$ (from the previous derivation of \mathcal{X}_{\Diamond_k}), and \mathcal{B}_2 is a ignorable branch.
 - if ψ is of form $[t, (f, b)^*]\psi_1$ and $\sigma = T$, add both $k : \psi_1 : T$ and $k : [t, \pi][t'_\pi, \pi^*]\psi_1 : T$ at the end of \mathcal{B} . Then, add $k : [t'_\pi, \pi^*]\psi_1 : T$ at the end of \mathcal{B} , and check if exists a state k_0 in which every formula of k_0 is true in k (and vice-versa). If it does, then this configuration has already been considered in the proof and this branch is a ignorable branch. if $\sigma = F$, introduce $k : \mathcal{X}_{\Box_i} : T$ at the end of \mathcal{B} , and check whether there is a state k_0 in which every formula of k_0 is true in k (and vice-versa), and $k_0 : \mathcal{X}_{\Box_k} : T$. If this holds, then \mathcal{X}_{\Box_i} collapses into \mathcal{X}_{\Box_k} , and k is a copy of k_0 . Therefore, \mathcal{B} is a ignorable branch and ψ should not be reduced further.
 - if ψ is of form \mathcal{X}_{\Box} , then at the end of \mathcal{B} add $k : \psi_1 : F$ at the left resulting branch \mathcal{B}_1 , and $k : \psi_1 : T$ and $k : [t, \pi]\mathcal{X}_{\Box_k} : F$ at the right resulting branch

\mathcal{B}_2 , splitting it in two new branches from \mathcal{B} . Now, add $k' : \mathcal{X}_{\Box k} : T$ at the end of \mathcal{B}_2 . Check whether there is a state k_0 in which every formula of k_0 is true in k' (and vice versa), and there is an eventuality $\mathcal{X}_{\Box k_0} = [t, (f, b)^*] \psi_1 = \mathcal{X}'_{\Box k}$. If this holds, then $\mathcal{X}_{\Box k'}$ collapses into $\mathcal{X}_{\Box k_0}$, and k' is a copy of k_0 . The left resulting branch \mathcal{B}_1 is closed because there is a $k_0 : \psi_1 : F$ (from the previous derivation of $\mathcal{X}_{\Box k}$), and \mathcal{B}_2 is a ignorable branch.

Following this procedure, all rules which do not refer to the \star operator (i.e., rules denoted by bullets “Logical rules” and “Modal rules” in Definition 5.4.2), decrease the size of the rules’ resulting formulae, either by decomposing logical operators (“Propositional rules”) or modalities (“Modal rules”), eventually reaching an atomic formula φ , except for modalities which may also introduce new states as prefixes for formulae.

As for the rules considering \star operator, their behaviour intuitively introduce a finite non-deterministic number of additional reductions (but no additional new states). Let us follow by contradiction by supposing that, by following the aforementioned procedure, tableau construction goes on forever. It is useful here that the number of formulae composing a formula φ is finite, although the amount of states is potentially infinite. Then, it is possible for some state k to know when all formulas that could have been introduced earlier in the tableau have already been. Consider a chain of states k_1, k_2, k_3, \dots in which k_i relates with k_j , $i < j$ as a result of rules $\langle \rangle - T$ or $\Box - F$ (i.e, rules that introduce a new unused state k_i), either themselves or as a result of some \star rule. Because the range of formulas obtained by rules application is finite, there exists k_i and k_j which are copies, i.e., they have the same associated set of formulae.

This is why the procedure always checks for the existence of such a state: new derivations are only performed for formulae that have not been already reduced, and the check for states which have the same set of formulae prefixed by them is performed to avoid derivations which lead to infinite branches. For the branch θ , if at step $n + 1$ such states do not exist (by induction, the first state k_i has already been found in the first n steps of the procedure, therefore step $n + 1$ will find state k_j in which k_j is a copy of state k_i), then this branch is an infinite branch and it will never be closed. When all formulae have been already reduced, the resulting tableau is either a closed tableau or an open tableau, to which the open branch leads to a counter model for the formula one is trying to prove.

□

5.4.3 Soundness

The proof of *ReLo*'s tableau soundness follows standard techniques in literature [31, 32]. We adapt them to *ReLo*'s reality in order to proceed. The soundness of the tableau proceeds by defining the notion of a satisfiable tableau, and then showing that satisfiability is a loop invariant property.

Let us define a map $m: \Phi \rightarrow \{T, F\}$ as a function that maps a formula (i.e., its labeling state and the formula itself) to T . Intuitively, m states that the tableau formula $w : \varphi$ is valid at the state denoted by its label.

Definition 5.4.7 (Tableau satisfiability). *A ReLo tableau \mathcal{T} is satisfiable if at least one of its branches \mathcal{B} is satisfiable. A branch \mathcal{B} is satisfiable if there is a map m which maps each of its formulae $(w : \varphi) \in \mathcal{B}$ to T .*

Lemma 14 (Satisfiability for *ReLo* Tableau is loop invariant). The application of any tableau expansion rule (as in Definition 5.4.2) to a satisfiable tableau \mathcal{T} will result in another satisfiable tableau \mathcal{T}' .

Proof. Suppose \mathcal{T} is a satisfiable tableau. We want to prove that the application of a tableau expansion rule to some formula ψ in a branch \mathcal{B} of \mathcal{T} results in a satisfiable tableau \mathcal{T}' . The proof proceeds by induction on the tableau expansion rules, showing that for each rule application, the resulting tableau is indeed satisfiable. From Definition 5.4.7, as \mathcal{T} is satisfiable, it has a satisfiable branch. Suppose b is such a satisfiable branch of \mathcal{T} . We have to analyze the following cases:

1. case $b \neq \mathcal{B}$: as no rule was applied in b and b is a branch of \mathcal{T}' , b is still a satisfiable branch of a satisfiable tableau \mathcal{T}' .
2. case $b = \mathcal{B}$: branch \mathcal{B} is satisfiable in \mathcal{T} . We must show that an tableau expansion rule application must yield a satisfiable tableau \mathcal{T}' . We need to consider each of the possible tableau expansion rules that could be applied to formula ψ . For these cases, we only consider rules $(* - T)$ as their dual may be obtained as follows:

- $(And - F)$ may be obtained by the negation of $(Or - T)$
- $(Or - F)$ may be obtained by the negation of $(And - T)$
- $(\rightarrow - F)$ may be obtained by rewriting it to $\neg\varphi_1 \vee \varphi_2$, which reduces to the case of $(Or - F)$
- $(\langle t, \pi \rangle - F)$ may be obtained by the negation of $([t, \pi] - T)$
- $([t, \pi] - F)$ may be obtained by the negation of $(\langle t, \pi \rangle - T)$

- $(\langle t, \pi^* \rangle - F)$ may be obtained by the negation of $([t, \pi^*] - T)$
- $([t, \pi^*] - F)$ may be obtained by the negation of $(\langle t, \pi^* \rangle - T)$

Therefore, let the tableau expansion rule applied to a formula ψ be:

- $(And - T)$: by induction, ψ is $w : \varphi_1 \wedge \varphi_2$ mapped to T by a map m in \mathcal{T} . Therefore, because $w : \varphi_1$ and $w : \varphi_2$ maps both to T by m in \mathcal{B}' as a satisfiable branch, \mathcal{T}' is a satisfiable tableau.
- $(or - T)$: by induction, ψ is $w : \varphi_1 \vee \varphi_2$ mapped to T by a map m in \mathcal{T} . The application of $(or - T)$ yields two different branches from \mathcal{B} , namely \mathcal{B}_1 with φ_1 and \mathcal{B}_2 with φ_2 , and from ψ mapping to T by m , either φ_1 or φ_2 (or both) maps to T by m . If $m(\varphi_1) = T$, then \mathcal{B}_1 denotes a satisfiable branch in \mathcal{T}' , and if $m(\varphi_2) = T$, then \mathcal{B}_2 denotes a satisfiable branch in \mathcal{T}' . In both cases, applying this rule in a formula in the branch \mathcal{B} results in a satisfiable branch \mathcal{B}' of a satisfiable tableau \mathcal{T}' .
- $(\rightarrow - T)$: by induction, ψ is $w : \varphi_1 \rightarrow \varphi_2$, which can be rewritten as $\neg\varphi_1 \vee \varphi_2$, which reduces to the case of $(or - T)$.
- $(\langle t, \pi \rangle - T)$: by induction, ψ is $w : \langle t, \pi \rangle \varphi$ mapped to T by a map m in \mathcal{T} . This rule yields $m(x : \varphi) = T$ in \mathcal{T}' , where φ is labeled with a new state x in the sequence of states allocated to formulas in \mathcal{T} . Therefore, the resulting branch \mathcal{B}' is a satisfiable branch in \mathcal{T}' .
- $([t, \pi] - T)$: by induction, ψ is $w : [t, \pi] \varphi$ mapped to T by a map m in \mathcal{T} . This rule yields $m(x : \varphi) = T$ in \mathcal{T}' , where φ is labeled with a new (or already existing) state in the sequence of states allocated to formulas in \mathcal{T} . Therefore, the resulting branch \mathcal{B}' is a satisfiable branch in \mathcal{T}' .
- $([t, \pi^*] - T)$: by induction, ψ is $w : [t, \pi^*] \varphi$ mapped to T by a map m in \mathcal{T} . The branch \mathcal{B}' that results from applying this rule to ψ is \mathcal{B} with formulae $w : \varphi$ and $w : [t, \pi][t'_\pi, \pi^*] \varphi$, where both formulae maps to t by m . Therefore, \mathcal{B}' is a satisfiable branch, yielding \mathcal{T}' as a satisfiable tableau.
- $(\langle t, \pi^* \rangle - T)$: by induction, ψ is $w : \langle t, \pi^* \rangle \varphi$ with $m(\langle t, \pi^* \rangle \varphi) = T$ in \mathcal{T} . This rule yields a new propositional symbol \mathcal{X}_\emptyset , which in turn may be further reduced by the \mathcal{X}_\emptyset rule. The application of \mathcal{X}_\emptyset results in a tableau \mathcal{T}' with two new branches as follows.
 - \mathcal{B}_1 which results from appending $w : \varphi$ to \mathcal{B} . From the rule, $m(w : \varphi)$ maps to T in \mathcal{B}_1 , rendering it as a satisfiable branch of \mathcal{T}' , which consequently

is a satisfiable tableau.

- \mathcal{B}_2 , which results from appending both $w : \varphi$ and $w : \langle t, \pi \rangle \mathcal{X}_\Diamond$. From the rule, in this branch $m(w : \varphi) = F$ and $m(w : \langle t, \pi \rangle \mathcal{X}_\Diamond) = T$. We need to ensure now that the outcome of subsequent \mathcal{X}_\Diamond also results in a satisfiable tableau, where they may result in ignorable branches. We need to ensure that such branches can be discarded, still resulting on a satisfiable tableau \mathcal{T}' .

Therefore, let us define a order relation R_{\prec} as a relation over formulae \mathcal{X}_\Diamond in a branch \mathcal{B} as follows:

- * $(x_0 : \mathcal{X}_\Diamond) R_{\prec} (x_1 : \mathcal{X}_\Diamond)$ iff $x_0 : \mathcal{X}_\Diamond$ belongs to a segment X_0 , $x_1 : \mathcal{X}_\Diamond$ appears in a segment X_1 and (x_0, X_0) is shorter than (x_1, X_1) .
- * $(x_1 : \mathcal{X}_{\Diamond_1}) R_{\prec} (x_0 : \mathcal{X}_{\Diamond_0})$ iff \mathcal{X}_{\Diamond_1} collapses in \mathcal{X}_{\Diamond_0} and x_0 is the fully reduced copy of x_1 .

Now, let $R_{\prec-T}$ be R_{\prec} 's transitive closure. Therefore, by R_{\prec} , there is a state ω with $(\omega : \mathcal{X}_{\Diamond_\omega})$, where either \mathcal{X}_\Diamond will be fulfilled by some $x_i : \varphi : T$ (as yielded by the result of \mathcal{X}_\Diamond 's rule application), and x_ω is accessible from x_i , or that $(\omega : \mathcal{X}_{\Diamond_\omega}) R_{\prec-T} (\omega : \mathcal{X}_{\Diamond_\omega})$, and for every x_i in $path(\mathcal{X}_\Diamond)$, $x_i : \varphi : F$, where $path(\mathcal{X}_\Diamond)$ is the path of states reached from x_0 until x_ω from successive \mathcal{X}_\Diamond expansions. Intuitively, the first case denotes the scenario where the eventuality \mathcal{X}_\Diamond will be fulfilled, and the second one denotes that it cannot be fulfilled, and hence it is a ignorable branch that can be discarded, where both scenarios leads to a consistent \mathcal{T}' .

□

With Lemma 14, we may proceed with the following lemmas, also standards in the literature [28, 32].

Lemma 15. If there is a closed tableau \mathcal{T} for φ , then φ is not satisfiable

Proof. The proof proceeds by contradiction. Suppose that there is a closed tableau for φ , but φ is satisfiable. The construction of the tableau for φ is the tableau $\langle w : \varphi \rangle$ which is basically φ labeled with a "starting" state w . This tableau is satisfiable (because we supposed φ is satisfiable). Then, from Lemma 14, each subsequent tableau generated by applying the tableau expansion rules to subformulas of φ results in satisfiable tableau, even for the closed tableau \mathcal{T} . But because \mathcal{T} is closed, there is no satisfiable branch \mathcal{B} in \mathcal{T} , which contradicts Lemma 14. □

Lemma 16 (Soundness of *ReLo* tableau). If φ has a tableau proof, then φ is a tautology in *ReLo*.

Proof. A tableau proof for φ comes from a closed tableau for $w : \varphi : F$. From Lemma 15, if $\neg\varphi$ has a closed tableau, then it is not satisfiable. Therefore, it follows that φ is indeed satisfiable. \square

Chapter 6

A *ReLo* Implementation in Coq

In this chapter, we detail and discuss the implementation of *ReLo* as described in Chapter 5, focusing on the logic’s core aspects. We also implement some functionalities regarding model verification, including a verification mechanism that returns whether a model satisfies some *ReLo* formula, a mechanism that tries to find a model that satisfies a formula (if satisfiable), and a tableau proof procedure based on the rules presented in Section 5.4. All the code discussed here is available at <https://github.com/frame-lab/ReoLogicCoq>.

6.1 Core *ReLo* definitions

The formalization starts by implementing *ReLo*’s basic notions referring to frame and model as in Definitions 5.1.8 and 5.1.9. They are formalized as Coq records respectively as `frame` (Definition 6.1.1) and `model` (Definition 6.1.2) as follows:

Definition 6.1.1 (*ReLo* frames in Coq).

```
Record frame := mkframe {  
  S : set state;  
  R : set (state × state);  
  lambda : state → name → QArith_base.Q;  
  delta : state → set dataConnector  
}.
```

- *S* is the set of states;
- *R* is the relation $R_{\Pi} \subseteq S \times S$, structured in Coq as a set of pair of states (s_i, s_j) , denoting that s_j is accessible from s_i ;

- *lambda* is a function that expects a state s and a port name n , returning the time the data was flowing in port name n at state s , and
- *delta* as the function which returns the possible data items of port names at a state s .

Definition 6.1.2 (*ReLo* models in Coq). A *ReLo* model is a *Record* formalized containing two fields, namely *Fr* as a *frame* from Definition 6.1.1, and *V* as the model's valuation function, which maps a state s and a propositional formula φ to *true* if φ holds in state s , and false otherwise.

```
Record model := mkmodel {
  Fr : frame;
  V : state → (dataProp name data) → bool
}.
```

The following definitions introduces Reo connectors based on their behaviour, as explored in Section 4. Definition *flowProgram* introduces Reo programs that denote data flow from a source node to a sink node, while *blockProgram* formalizes the connectors that have their behavior as “blocks” which may enable or hold data flow to the remainder of the connectors (attached to these connectors by a common node):

```
Inductive flowProgram :=
| flowSync : name → name → flowProgram
| flowLossySync : name → name → flowProgram
| flowFifo : name → name → flowProgram
| flowMerger : name → name → name → flowProgram
| flowReplicator : name → name → name → flowProgram
| flowTransform : (data → data) → name → name → flowProgram
| flowFilter : (data → bool) → name → name → flowProgram.

Inductive blockProgram :=
| flowSyncdrain : name → name → blockProgram
| flowaSyncdrain : name → name → blockProgram.
```

We proceed by formalizing the *ReLo* program as $\pi = (f, b)$ as the following Coq definition. Constructor *reoProg* expects two arguments: a *set* *flowProgram* denoting the set of flow programs f and *set* *blockProgram* stands for the set of “blocking” programs b .

```
Inductive reoProgram :=
| reoProg : set flowProgram → set blockProgram → reoProgram.
```

Then, the definition of a Reo connector in Coq is achieved by mapping $\pi = (f, b)$ as a Reo circuit Π out of canonical connectors by means of the following definition:

```

Inductive connector :=
| sync : name → name → connector
| lossySync : name → name → connector
| fifo : name → name → connector
| syncDrain : name → name → connector
| asyncDrain : name → name → connector
| filterReo : (data → bool) → name → name → connector
| transform : (data → data) → name → name → connector
| merger : name → name → name → connector
| replicator : name → name → name → connector.

```

After the definition of `connector`, the possible data flows of each connector are formalized in coq as `dataConnector` shown below.

```

Inductive dataConnector :=
| fifoData : name → data → name → dataConnector
| dataPorts : name → data → dataConnector.

```

A *ReLo* program is defined as $\pi = (f.b)$, where f is the set of connectors that compose the Reo model which are connectors that models data communication between two entities (`flowProgram`), and b is the set of “blocking” connectors, namely the ones that describe a drain behaviour (`blockProgram`).

A program $\pi = (f, b)$ is then converted to a Reo model composed by channels introduced in Figure 4.1 and defined in Coq by `connector`. A lift is performed to each of the programs by the following definition.

```

Definition program2SimpProgram (prog : reoProgram) : set connector :=
  match prog with
  | reoProg setFlow setBlock ⇒ (block2Reo setBlock) ++ (flow2Reo setFlow)
  end.

```

Then, we define `program` as an inductive type which depicts the type of possible data firing for each Reo connector modelled as a *ReLo* program. Its constructors will be used later by the firing relation to determine how data flow between which ports. The mapping between the `connector` and its corresponding `program` is straightforward, except for `sync`, `merger` and `replicator` which share `asyncTo` as the program. Constructor `transformTo` stands for the Transform channel, and expects a transformation function of support $data \rightarrow data$, and port names A and B , where the resulting data flow is $f(D_A)$, with D_A the data flow in the port at the moment. Constructor `filterTo` denotes the data flow for Filter channel, where the predicate over the data flow is modelled as an evaluation

of a function $f : data \rightarrow bool$. Therefore, data will flow from A to B only if $f(D_A) = true$.

```

Inductive program :=
| to : name → name → program
| asyncTo : name → name → program
| fifoAlt : name → name → program
| transformTo : (data → data) → name → name → program
| filterTo : (data → bool) → name → name → program
| SBlock : name → name → program
| ABlock : name → name → program.

```

We proceed by defining `parse` as in Definition 5.1.4 in the following Coq code. It reflects the order of processing of each Reo channel that compose the Reo model as Definition 5.1.4 proposes, with the same purpose of correctly evaluate the programs, independently of the order they are used to construct the model as a whole.

```

Fixpoint parse (pi: list connector) (s : list program) : list program :=
  match pi with
  | [] ⇒ s
  | a::t ⇒ match a with
    | sync a b ⇒ (parse(t) (s ++ [to a b]))
    | lossySync a b ⇒ (parse(t) (s ++ [asyncTo a b]))
    | fifo a b ⇒ (parse t s) ++ [fifoAlt a b]
    | syncDrain a b ⇒ (parse(t) ([SBlock a b] ++ s))
    | asyncDrain a b ⇒ (parse(t) [ABlock a b] ++ s)
    | filterReo f a b ⇒ (parse(t) ([filterTo f a b] ++ s))
    | transform f a b ⇒ (parse(t) ([transformTo f a b] ++ s))
    | merger a b c ⇒ (parse(t) (s ++ [(to a c); (to b c)]))
    | replicator a b c ⇒ (parse(t) (s ++ [(to a b); (to a c)]))
  end
end.

```

The next definition is that of the function `fire`, which is given in Definition 5.1.6, and with a data flow of ports composing a program π and the program π itself models the resulting data flow as the result of firing all eligible connectors. It is used by function `go` as in Definition 5.1.5.

```

Fixpoint fire (t: set dataConnector) (s : set goMarks) (acc: set dataConnector)
: set (set dataConnector) :=
  match s with

```

```

| [] ⇒ match acc with
  | [] ⇒ []
  | x::y ⇒ [acc]
end
| ax::l ⇒ match ax with
  | goTo a b ⇒ if (existsb (fun x : (dataConnector) ⇒ match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t)) then
    match (port2port (goTo a b)
      (filter(fun y : (dataConnector) ⇒ match y with
        | dataPorts name1 data ⇒ (equiv_decb name1 a)
        | _ ⇒ false
        end) (t))) with
    | None ⇒ (fire t l acc)
    | Some x ⇒ (fire t l (x::acc))
    end
  else fire t l acc
  | goTransform f a b ⇒ if (existsb
    (fun x : (dataConnector) ⇒ match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t)) then
    match (port2portTr f (goTransform f a b)
      (filter(fun x : (dataConnector) ⇒ match x with
        | dataPorts name1 data ⇒ (equiv_decb name1 a)
        | _ ⇒ false
        end) (t))) with
    | None ⇒ (fire t l acc)
    | Some x ⇒ (fire t l ((x::acc)))
    end
  else fire t l acc
  | goFilter f a b ⇒ if (existsb (fun x : (dataConnector) ⇒ match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false

```

```

    end) (t)) then
    match (port2portFil f (goTo a b)
    (filter(fun x : (dataConnector) ⇒ match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t))) with
    | None ⇒ (fire t l acc)
    | Some x ⇒ (fire t l ((x::acc)))
    end
    else fire t l acc
  | goFifo a x b ⇒ if (existsb (fun x : (dataConnector) ⇒ match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t))
    then (fire t l ((fifoData a x b)::acc))
    else fire t l acc
  | goFromFifo a x b ⇒
    if (existsb (fun x : (dataConnector) ⇒ match x with
    | fifoData name1 data name2 ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t))
    then (fire t l ((dataPorts b x)::acc))
    else fire t l acc

end

end.

```

Therefore, the function `fire` expects as input the current data flow as t , a set of possible firings as s and acc as the output which will contain the next expected data flow of the whole connector. Its execution coordinates which ports and connectors fire, adding the result of firing them to acc . Auxiliary function `port2port` searches in the data flow t if, for each of the possible firings to be considered in s there is a data flowing in their source nodes. If this is the case, then the data found is the one to be considered in the connector's firing. The exception is for the case where a FIFO connector has data in it, where then `fire` will check in the data flow t whether the FIFO has data in it.

Note that `fire` employs auxiliary functions like `port2port`, `port2portTr`, and `port2portFil`, which retrieve the destination nodes of a connector when firing, and in the case of Trans-

form and Filter connectors, respectively applies the function to the data item flowing from the sink node to the source node, and only transmits the data to the sink node if the filtering criteria are satisfied.

Function *fire* also has as one of its parameters a variable s as a set of *goMarks*, which are effectively processed by it. The inductive type *goMarks* denote the type of possible data flows of each Reo connector modelled in *connectors* and it is defined as follows:

Inductive *goMarks* :=

- | *goTo* : $name \rightarrow name \rightarrow goMarks$
- | *goFifo* : $name \rightarrow data \rightarrow name \rightarrow goMarks$
- | *goFromFifo* : $name \rightarrow data \rightarrow name \rightarrow goMarks$
- | *goTransform* : $(data \rightarrow data) \rightarrow name \rightarrow name \rightarrow goMarks$
- | *goFilter* : $(data \rightarrow bool) \rightarrow name \rightarrow name \rightarrow goMarks$.

Constructor *goMarks* formalizes the data flows of channels *Sync*, *LossySync*, *Merger* and *Replicator* with *goTo*, the data respectively puring into and flowing *FIFO* channels as *goFifo* and *goFromFifo*, the flow of *Transform* and *Filter* channels as *goTransform* and *goFilter*, respectively. Each of these constructors will be used by *go* to structure the connectors eligible to data flow, which in turn will be effectively processed by *fire*.

With the definition of *fire*, we define *go* following Definition 5.1.5: for each connector $\pi_i \in s$, the function will check whether the specified conditions in Definition 5.1.5 are satisfied by the data item in t as follows:

- **to A B** as the *Sync* connector: *go* will then add *goTo A B* according to iff A has data in it in t .
- **asyncTo A B** as the *LossySync* connector: *go* will add *goTo A B* and *goTo A A* as data flows to be processed iff A has data in it in t . This simulates the two possible outcomes form the *LossySync* execution: either the data flows from A to B, or it stays in port name A denoting it was lost on its way to B.
- **fifoAlt A B** as the *FIFO* connector: *go* needs to consider two cases:
 1. There is data coming into the FIFO channel: iff A has data in it in t , then *go* adds *goFifo A B* to denote that there is data bound to enter the *FIFO* from its source node;
 2. There is data coming into the FIFO channel: iff there is data in the FIFO in t , then *go* adds *goFromFifo A B* to denote that there is data bound to flow out of the FIFO channel to its sink node.

- **transformTo** t A B as the *Transform* connector: iff A has data in it in t , then it adds **goTransform** f A B to denote the data flow of A to B with the data item in A transformed by f .
- **filterTo** p A B as the *Filter* connector: **transformTo** t A B as the *Transform* connector: iff A has data in it in t , then it adds **goFilter** f A B to denote the data flow of A to B will take place only if the data item satisfies the property specified in f .
- **SBlock** A B as the *SyncDrain* connector: if both port names have data flowing simultaneously in the channel, then it continues normal processing for the remainder of the connector. Otherwise, **go** will remove from further processing connectors with a sink node equal to one of the nodes of the SyncDrain channel being processed.
- **ABlock** A B as the *AsyncDrain*: if only one of the port names has data flowing simultaneously in the channel, then it continues normal processing for the remainder of the connector. Otherwise, **go** will remove from further processing connectors with a sink node equal to one of the nodes of the SyncDrain channel being processed.

```

Fixpoint go ( $s$ : set program) ( $k$ : nat) ( $acc$ : set goMarks) ( $t$ : set dataConnector) :
set (set dataConnector) :=
  match  $k$  with
  | 0  $\Rightarrow$  fire  $t$   $acc$  []
  | Datatypes.S  $n$   $\Rightarrow$  match  $s$  with
    | []  $\Rightarrow$  fire  $t$   $acc$  []
    |  $prog::s'$   $\Rightarrow$  match  $prog$  with
      | to  $a$   $b$   $\Rightarrow$  if existsb (fun  $x$ : (dataConnector)  $\Rightarrow$  match  $x$  with
        | dataPorts  $name1$   $data$   $\Rightarrow$  (equiv_decb  $name1$   $a$ )
        | _  $\Rightarrow$  false
      end) ( $t$ ) then
        if negb((existsb (fun  $x$ : goMarks  $\Rightarrow$  match  $x$  with
          | goTo  $name1$   $name2$   $\Rightarrow$  (equiv_decb  $name2$   $b$ )
          | goFifo  $name1$   $data$   $name2$   $\Rightarrow$  (equiv_decb  $name2$   $b$ )
          | goFromFifo  $name1$   $data$   $name2$   $\Rightarrow$  (equiv_decb  $name2$   $b$ )
          | goTransform  $f$   $name1$   $name2$   $\Rightarrow$  (equiv_decb  $name2$   $b$ )
          | goFilter  $f$   $name1$   $name2$   $\Rightarrow$  (equiv_decb  $name2$   $b$ )
        end) ( $acc$ )))
      then (go  $s'$   $n$  ( $acc++[goTo a b]$ )  $t$ )
      else

```

```

      (go s' n (swap acc (goTo a b)) t) ++ (go s' n (acc) t)
    else (go s' n acc t)
  | asyncTo a b ⇒ if (existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 name2 ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t)) then
    if negb((existsb (fun x : goMarks ⇒ match x with
    | goTo name1 name2 ⇒ (equiv_decb name2 b)
    | goFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goFromFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goTransform f name1 name2 ⇒ (equiv_decb name2 b)
    | goFilter f name1 name2 ⇒ (equiv_decb name2 b)
    end) (acc)))
    then (go s' n (acc++[goTo a b]) t) ++
      (go s' n (acc++[goTo a a]) t)
    else
      (go s' n ((swap acc (goTo a b))++[goTo a b]) t) ++
      (go s' n (acc) t)
    else (go s' n acc t)
  | transformTo f a b ⇒ if existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t) then
    if negb((existsb (fun x : goMarks ⇒ match x with
    | goTo name1 name2 ⇒ (equiv_decb name2 b)
    | goFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goFromFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goTransform f name1 name2 ⇒ (equiv_decb name2 b)
    | goFilter f name1 name2 ⇒ (equiv_decb name2 b)
    end) (acc)))
    then (go s' n (acc++[goTransform f a b]) t)
    else (go s' n (swap acc (goTransform f a b)) t) ++
      (go s' n (acc) t)

```

```

    else (go s' n acc t)
  | filterTo f a b ⇒ if existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t) then
    if negb((existsb (fun x : goMarks ⇒ match x with
    | goTo name1 name2 ⇒ (equiv_decb name2 b)
    | goFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goFromFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goTransform f name1 name2 ⇒ (equiv_decb name2 b)
    | goFilter f name1 name2 ⇒ (equiv_decb name2 b)
    end) (acc)))
    then
      (go s' n (acc++[goFilter f a b]) t)
    else
      (go s' n (swap acc (goFilter f a b)) t) ++ (go s' n (acc) t)
    else
      (go s' n acc t)
  | fifoAlt a b ⇒ if (existsb (fun x : (dataConnector) ⇒
    match x with
    | fifoData name1 data name2 ⇒ (equiv_decb name1 a)
    | _ ⇒ false
    end) (t)) then
    if negb((existsb (fun x : goMarks ⇒ match x with
    | goTo name1 name2 ⇒ (equiv_decb name2 b)
    | goFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goFromFifo name1 data name2 ⇒ (equiv_decb name2 b)
    | goTransform f name1 name2 ⇒ (equiv_decb name2 b)
    | goFilter f name1 name2 ⇒ (equiv_decb name2 b)
    end) (acc)))
    then
      (go s' n (acc++dataConnectorToGoMarksFifo(t)) t) ++
      (go s' n (acc) t)
    else

```

```

      (go s' n (acc++dataConnectorToGoMarksFifo(t)) t)
    else
      if (existsb (fun x : (dataConnector) ⇒ match x with
        | fifoData name1 data name2 ⇒ (equiv_decb name1 a)
        | dataPorts name1 name2 ⇒ (equiv_decb name1 a)
      end) (t)) then
        (go s' n (acc++(dataConnectorToGoMarksPorts t
          (fifoAlt a b))) t)
      else (go s' n acc t)
  | SBlock a b ⇒ if (existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
  end) t) && (existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 b)
    | _ ⇒ false
  end) t) ||
    negb((existsb (fun x : (dataConnector) ⇒
      match x with
      | dataPorts name1 data ⇒ (equiv_decb name1 a)
      | _ ⇒ false
    end) t)) &&
      negb(existsb (fun x : (dataConnector) ⇒ match x with
        | dataPorts name1 data ⇒ (equiv_decb name1 b)
        | _ ⇒ false
      end) t) then
        (go s' n acc t)
      else
        (go (halt [a;b] s') n acc t)
  | ABlock a b ⇒ if negb((existsb (fun x : (dataConnector) ⇒
    match x with
    | dataPorts name1 data ⇒ (equiv_decb name1 a)
    | _ ⇒ false
  end) t)) &&

```

```

negb((existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 b)
| _ ⇒ false
end) t)) ||
(existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 a)
| _ ⇒ false
end) t) &&
(existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 b)
| _ ⇒ false
end) t)
then (go (halt [a;b] s') n acc t)
else
if negb((existsb (fun x : (dataConnector) ⇒
match x with
| dataPorts name1 data ⇒ (equiv_decb name1 a)
| _ ⇒ false
end) t)) &&
(existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 b)
| _ ⇒ false
end) t) ||
((existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 a)
| _ ⇒ false
end) t)) &&
negb(existsb (fun x : (dataConnector) ⇒ match x with
| dataPorts name1 data ⇒ (equiv_decb name1 b)
| _ ⇒ false
end) t)
then (go s' n acc t)
else (go (halt [a;b] s') n acc t)
end
end
end

```

end.

Each of the conditions processed by `go` also considers when there is more than a connector with the same sink node, in which cases data may flow nondeterministically. These cases are handled by `go` for each connector, in which the function will generate a output considering each scenario: in the following Coq piece of code, we explain the idea referring to the case of *Sync* as **to A B**.

```
(go s' n (swap acc (goTo a b)) t) ++ (go s' n (acc) t)
```

The output for this case will recursively call `go` in two scenarios:

1. excluding all other connectors that have the same sink node B as the connector being evaluated (employing `swap`) as an auxiliary definition: `(go s' n (swap acc (goTo a b)) t)`
2. not considering the current connector `(go s' n (acc) t)`.

Therefore, `swap` is defined as follows.

```
Fixpoint swap (s: set goMarks) (current: goMarks) : set goMarks :=
  match s with
  | [] => []
  | dataMark::t => match dataMark with
    | goTo a b => match current with
      | goTo u v => if (equiv_decb b v)
        then (goTo u v)::(swap t current)
        else (goTo a b)::(swap t current)
      | goFifo u w v => if (equiv_decb b v)
        then (goFifo u w v)::(swap t current)
        else (goTo a b)::(swap t current)
      | goFromFifo u w v => if (equiv_decb b v)
        then (goFromFifo u w v)::(swap t current)
        else (goTo a b)::(swap t current)
      | goTransform f u v => if (equiv_decb b v)
        then (goTransform f a b)::(swap t current)
        else (goTo a b)::(swap t current)
      | goFilter f u v => if (equiv_decb b v)
        then (goFilter f a b)::(swap t current)
        else (goTo a b)::(swap t current)
    end
```

```

| goTransform f a b ⇒ match current with
  | goTo u v ⇒ if (equiv_decb b v)
    then (goTo u v)::(swap t current)
    else (goTransform f a b)::(swap t current)
  | goFifo u w v ⇒ if (equiv_decb b v)
    then (goFifo u w v)::(swap t current)
    else (goTransform f a b)::(swap t current)
  | goFromFifo u w v ⇒ if (equiv_decb b v)
    then (goFromFifo u w v)::(swap t current)
    else (goTransform f a b)::(swap t current)
  | goTransform f u v ⇒ if (equiv_decb b v)
    then (goTransform f a b)::(swap t current)
    else (goTransform f a b)::(swap t current)
  | goFilter f' u v ⇒ if (equiv_decb b v)
    then (goFilter f' a b)::(swap t current)
    else (goTransform f a b)::(swap t current)
    end
| goFilter f a b ⇒ match current with
  | goTo u v ⇒ if (equiv_decb b v)
    then (goTo u v)::(swap t current)
    else (goFilter f a b)::(swap t current)
  | goFifo u w v ⇒ if (equiv_decb b v)
    then (goFifo u w v)::(swap t current)
    else (goFilter f a b)::(swap t current)
  | goFromFifo u w v ⇒ if (equiv_decb b v)
    then (goFromFifo u w v)::(swap t current)
    else (goFilter f a b)::(swap t current)
  | goTransform f' u v ⇒ if (equiv_decb b v)
    then (goTransform f' a b)::(swap t current)
    else (goFilter f a b)::(swap t current)
  | goFilter f u v ⇒ if (equiv_decb b v)
    then (goFilter f a b)::(swap t current)
    else (goFilter f a b)::(swap t current)
    end
| goFifo a data b ⇒ match current with

```



```

| goTo x y ⇒ if (equiv_decb b y)
  then (goTo x y)::(swap t current)
  else (goFifo a data b)::(swap t current)
| goFifo x y z ⇒ if (equiv_decb b z)
  then (goFifo x y z)::(swap t current)
  else (goFifo a data b)::(swap t current)
| goFromFifo x y z ⇒ if (equiv_decb b z)
  then (goFromFifo x y z)::(swap t current)
  else (goFifo a data b)::(swap t current)
| goTransform f y z ⇒ if (equiv_decb b z)
  then (goTransform f y z)::(swap t current)
  else (goFifo a data b)::(swap t current)
| goFilter f u v ⇒ if (equiv_decb b v)
  then (goFilter f a b)::(swap t current)
  else (goFifo a data b)::(swap t current)
  end
| goFromFifo a data b ⇒ match current with
| goTo x y ⇒ if (equiv_decb b y)
  then (goTo x y)::(swap t current)
  else (goFromFifo a data b)::(swap t current)
| goFifo x y z ⇒ if (equiv_decb b z)
  then (goFifo x y z)::(swap t current)
  else (goFromFifo a data b)::(swap t current)
| goFromFifo x y z ⇒ if (equiv_decb b z)
  then (goFromFifo x y z)::(swap t current)
  else (goFromFifo a data b)::(swap t current)
| goTransform f y z ⇒ if (equiv_decb b z)
  then (goTransform f y z)::(swap t current)
  else (goFromFifo a data b)::(swap t current)
| goFilter f u v ⇒ if (equiv_decb b v)
  then (goFilter f a b)::(swap t current)
  else (goFromFifo a data b)::(swap t current)
  end
end
end.

```

Function `go` deals with the “blocking programs” induced by connectors `SyncDrain` and `AsyncDrain` as specified in Definition 5.1.5 (and discussed right before `go`’s Coq code as `ABlock` and `SBlock`): if the conditions required to enable data flow from these connectors are not met, it introduces a call to a definition $halt(a, b, s')$ where a and b are the port names of the connector, and s' is the remainder of the connector to be processed. Then, `halt` will exclude the connectors from s' which contains the same sink nodes as the ones in the connector (either `SyncDrain` or `AsyncDrain`) from the firing processing `fire` performs when finishing `go`’s execution.

Definition `halt` ($names : set\ name$) ($s' : set\ program$) :=
`removePortNames s' (haltAux names names s' (length s'))`

The firing relation f as Definition 5.1.7 formalizes is implemented by the following Coq definition:

Definition `f` ($t : set (set\ dataConnector)$) ($pi : set\ connector$) :=
`flat_map (go (parse pi [])) (length (parse pi [])) [] t.`

6.2 Model Verification

After implementing the concepts of *ReLo* in Section 6.1, we proceed by implementing the notion of a *ReLo* formula in Coq. This definition will pave the way to our model verification tool, and for the formalization of the tableau procedure we’ll present in Section 6.4. We start by formalizing the notion of a syntactic *ReLo* program as `syntacticProgram` as an inductive type with two constructors: `sProgram` denoting a standard *ReLo* program, and `star` to denote the finite non-deterministic iteration of a *ReLo* program.

Inductive `syntacticProgram` :=
 | `sProgram` : `reoProgram` → `syntacticProgram`
 | `star` : `reoProgram` → `syntacticProgram`.

With the definition of `syntacticProgram`, we formalize `formula` as an inductive type that formalizes *ReLo* formulae in Coq. Each of its constructors denote a different formula component in *ReLo*: `proposition` stands for a regular propositional formulae, modalities $\langle \rangle$ and $[]$ are respectively formalized as `diamond` and `box`, propositional operators $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ are formalized respectively by constructors `and`, `or`, `neg`, `imp`, `bimpl`. Constructors `chiFormulaBox` and `chiFormulaBox` formalizes respectively the propositional symbols \mathcal{X}_\square and \mathcal{X}_\diamond . For these constructors, the parameter `nat` denotes the index of the \mathcal{X} variable derived, following the definitions in Section 5.4 and its parameter `formula` denotes the formula which has been rewritten as an eventuality \mathcal{X}_\square or \mathcal{X}_\diamond . The inductive type `formula` will also be employed in the tableau implemented in Section 6.4.

```

Inductive formula :=
| proposition : (dataProp name data) → formula
| box : set dataConnector → syntaticProgram → formula → formula
| diamond : set dataConnector → syntaticProgram → formula → formula
| and : formula → formula → formula
| or : formula → formula → formula
| neg : formula → formula
| imp : formula → formula → formula
| biImpl : formula → formula → formula
| chiFormulaBox : nat → formula → formula
| chiFormulaBox : nat → formula → formula.

```

These definitions allow us to start the development of the framework which, given a *ReLo* program Π and a *ReLo* formula φ , returns a model of Π in which φ is valid (if exists). If φ does not hold in Π , Coq will return a model with the formulae valid, based on φ 's processing. We formalize the notion of satisfaction of formulas $[t, (f, b)]\varphi$ and $\langle t, (f, b) \rangle$, φ is a propositional symbol respectively as **boxSatisfactionPi** and **diamondSatisfactionPi**.

Definition **boxSatisfactionPi** with a model \mathcal{M} as m , a propositional formula p and a set of states (which denotes the actual configuration of the model reached after executing a \square modality) $states$ returns true if p is valid in every state in $states$. This is performed by evaluating V as the model m 's valuation function with each state of $states$ and the propositional formula p . Conversely, **diamondSatisfactionPi** with the same parameters will yields true if p is valid in at least one state in $states$.

```

Definition boxSatisfactionPi (m:model) (p : dataProp name data)
(states: set state) :=
if (states == []) then false else
forallb (fun x : state ⇒ (V(m)x p)) states.
Definition diamondSatisfactionPi (m:model) (p : dataProp name data)
(states : set state) :=
if (states == []) then false else
existsb (fun x : state ⇒ (V(m)x p)) states.

```

The formalization proceeds with function **singleModelStep** as the core of the model verification process in our implementation. It recursively decomposes the formula inputted as *formula* φ as a composite formula until propositional symbols are reached as follows:

- if φ is a proposition p denoted by the match with **proposition**, **singleModelStep** evaluates whether the proposition p is valid in the model m at the state s

- If φ is the negation of another formula $\neg\psi$ denoted by the match with **neg**, then **singleModelStep** recursively calls itself to check whether ψ is valid, yielding the boolean negation of result of this recursive call.
- If φ is the conjunction of two formulae $\phi_1 \wedge \phi_2$ denoted by the match with **and**, then **singleModelStep** recursively calls itself to check whether both ψ_1 and ψ_2 are valid.
- If φ is the disjunction of two formulae $\phi_1 \vee \phi_2$, then **or** recursively calls itself to check whether either ψ_1 or ψ_2 are valid.
- If φ is the implication of two formulae $\phi_1 \rightarrow \phi_2$, then **imp** recursively calls itself to check whether either $\neg\psi_1$ or ψ_2 are valid (by considering the definition of connective \rightarrow in terms of \vee).
- If φ is the bi-implication of two formulae $\phi_1 \leftrightarrow \phi_2$, then **biImpl** recursively calls itself to check whether either $\psi_1 \rightarrow \psi_2$ and $\psi_2 \rightarrow \psi_1$ are valid (by considering the definition of connective \rightarrow in terms of \vee).
- If φ is $[t, (f, b)]\psi$, then **singleModelStep** needs to consider 1) the program that is within the modality, i.e., if it is either a regular program or a iteration of a program with \star operator, and 2) the structure of ψ to accordingly recursively process it with **singleModelStep**, similar to how it is performed for the propositional operators. From 1), **singleFormulaVerify** will either retrieve a set of all states $\{w \in S \mid (s, w) \in R_\Pi\}$ by means of **retrieveRelatedStatesFromV**, or it will calculate the reflexive transitive closure of R_Π by means of **RTC** to obtain the set of states $\{w \in S \mid (s, w) \in R_{\Pi^\star}\}$. The processing by **singleModelStep** will require that ψ to be valid in all states w found by **retrieveRelatedStatesFromV** for both cases, which is done by the standard function **forallb** employed.
- If φ is $\langle t, (f, b) \rangle \psi$, then **singleModelStep** needs to consider 1) the program that is within the modality, i.e., if it is either a regular program or a iteration of a program with \star operator, and 2) the structure of ψ to accordingly recursively process it with **singleModelStep**, similar to how it is performed for the propositional operators. From 1), **singleFormulaVerify** will either retrieve a set of all states $\{w \in S \mid (s, w) \in R_\Pi\}$ by means of **retrieveRelatedStatesFromV**, or it will calculate the reflexive transitive closure of R_Π by means of **RTC** to obtain the set of states $\{w \in S \mid (s, w) \in R_{\Pi^\star}\}$. The processing by **singleModelStep** will require that ψ to be valid in at least one state w found by **retrieveRelatedStatesFromV** for both cases, which is done by the standard function **existsb** employed.

```

Fixpoint singleModelStep (m:model) (formula : formula) (s:state) : bool :=
  match formula with
  | chiFormulaBox x phi | chiFormulaBox x phi => false
  | proposition p => (V(m) s p)
  | neg p => negb (singleModelStep m p s)
  | and a b => (singleModelStep m a s) && (singleModelStep m b s)
  | or a b => (singleModelStep m a s) || (singleModelStep m b s)
  | imp a b => negb (singleModelStep m a s) || (singleModelStep m b s)
  | bimpl a b => (negb (singleModelStep m a s) || (singleModelStep m b s)) &&
    (negb (singleModelStep m b s) || (singleModelStep m a s))
  | box t pi p' => match pi with
    | sProgram reo => match p' with
      | chiFormulaBox x phi | chiFormulaBox x phi => false
      | proposition p'' => boxSatisfactionPi (m) (p'')
        (retrieveRelatedStatesFromV (R(Fr(m))) s)
      | box t' pi' p'' => forallb (singleModelStep m p')
        (retrieveRelatedStatesFromV (R(Fr(m))) s)
      | diamond t' pi' p'' => existsb (singleModelStep m p')
        (retrieveRelatedStatesFromV (R(Fr(m))) s)
      | and a b => (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s))) &&
        (forallb (singleModelStep m b)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s))))
      | or a b => (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s))) ||
        (forallb (singleModelStep m b)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s))))
      | neg a => negb (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
      | imp a b => (negb (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s)))) ||
        (forallb (singleModelStep m b)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s))))
      | bimpl a b => (negb (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (R(Fr(m))) s)))) ||

```

```

      (forallb (singleModelStep  $m$   $b$ )
        ((retrieveRelatedStatesFromV (R(Fr( $m$ )))  $s$ ))) &&
      (negb (forallb (singleModelStep  $m$   $b$ )
        ((retrieveRelatedStatesFromV (R(Fr( $m$ )))  $s$ )))) ||
      (forallb (singleModelStep  $m$   $a$ )
        ((retrieveRelatedStatesFromV (R(Fr( $m$ )))  $s$ )))
    end
  | star  $reo \Rightarrow$  match  $p'$  with
  | chiFormulaBox  $x$   $\phi i$  | chiFormulaBox  $x$   $\phi i \Rightarrow$  false
  | proposition  $p'' \Rightarrow$  boxSatisfactionPi ( $m$ ) ( $p''$ )
    (retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )
  | box  $t' \pi i' p'' \Rightarrow$  forallb (singleModelStep  $m$   $p'$ )
    (retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )
  | diamond  $t' \pi i' p'' \Rightarrow$  existsb (singleModelStep  $m$   $p'$ )
    (retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )
  | and  $a$   $b \Rightarrow$  (forallb (singleModelStep  $m$   $a$ )
    ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ ))) &&
    (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))
  | or  $a$   $b \Rightarrow$  (forallb (singleModelStep  $m$   $a$ )
    ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ ))) ||
    (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))
  | neg  $a \Rightarrow$  negb (forallb (singleModelStep  $m$   $a$ )
    ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))
  | imp  $a$   $b \Rightarrow$  (negb (forallb (singleModelStep  $m$   $a$ )
    ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))) ||
    (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))
  | bimpl  $a$   $b \Rightarrow$  (negb (forallb (singleModelStep  $m$   $a$ )
    ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))) ||
    (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ ))) &&
    (negb (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ )))) ||
    (forallb (singleModelStep  $m$   $b$ )
      ((retrieveRelatedStatesFromV (RTC( $m$ ))  $s$ ))) ||

```

```

      (forallb (singleModelStep m a)
        ((retrieveRelatedStatesFromV (RTC(m)) s)))
      end

    end

  | diamond t pi p' ⇒ match pi with
    | sProgram reo ⇒ match p' with
    | chiFormulaBox x phi | chiFormulaBox x phi ⇒ false
    | proposition p'' ⇒ diamondSatisfactionPi (m) (p'')
      (retrieveRelatedStatesFromV (R(Fr(m))) s)
    | box t' pi' p'' ⇒ forallb (singleModelStep m p')
      (retrieveRelatedStatesFromV (R(Fr(m))) s)
    | diamond t' pi' p'' ⇒ existsb (singleModelStep m p')
      (retrieveRelatedStatesFromV (R(Fr(m))) s)
    | and a b ⇒ (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s))) &&
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
    | or a b ⇒ (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
    | neg a ⇒ negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
    | imp a b ⇒ (negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
    | bimpl a b ⇒ (negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s))) &&
      (negb (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))) ||
      (existsb (singleModelStep m a)

```

```

      ((retrieveRelatedStatesFromV (R(Fr(m))) s)))
end
| star reo ⇒ match p' with
| chiFormulaBox x phi | chiFormulaBox x phi ⇒ false
| proposition p'' ⇒ diamondSatisfactionPi (m) (p'')
      (retrieveRelatedStatesFromV (RTC(m)) s)
| box t' pi' p'' ⇒ forallb (singleModelStep m p')
      (retrieveRelatedStatesFromV (RTC(m)) s)
| diamond t' pi' p'' ⇒ existsb (singleModelStep m p')
      (retrieveRelatedStatesFromV (RTC(m)) s)
| and a b ⇒ (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s))) &&
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))
| or a b ⇒ (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))
| neg a ⇒ negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))
| imp a b ⇒ (negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))
| bimpl a b ⇒ (negb (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))) ||
      (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (RTC(m)) s))) &&
      (negb (existsb (singleModelStep m b)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))) ||
      (existsb (singleModelStep m a)
      ((retrieveRelatedStatesFromV (RTC(m)) s)))
end

```

As `singleModelStep`'s definition shows, it relies on two standard Coq functions and two user defined functions to deal with modalities. Definition `forallb` expects a function

$f: S \rightarrow \text{bool}$ and a set of states s of type S , returning true if $f(s_i) = \text{true}$ for each s_i in the set of states s , while `existsb` also expects a function $f: S \rightarrow \text{bool}$ and a set of states s of type S , but returning true if $f(s_i) = \text{true}$ for at least one s_i in the set of states s . Note that it also considers the formulas \mathcal{X}_{\Diamond} and \mathcal{X}_{\Box} as they are also members of `formula`, but as they are only introduced by the tableau (explained in Section 6.4), `singleModelStep` must not process it

The first user defined is `retrieveRelatedStatesFromV`, which given a set of states `setStates` and a state s it retrieves a set containing all states w such as pairs $(s, w) \in \text{setStates}$. Then, `RTC` calculates the reflexive transitive closure by applying the union operation of the model m 's transition relation `R`, its transitive closure returned by `getTransitive`, and its reflexive closure returned by `getReflexive`.

```

Fixpoint retrieveRelatedStatesFromV (setStates : set (state × state)) (s : state)
  : set state :=
  match setStates with
  | nil ⇒ nil
  | a :: states ⇒ if s == (fst a)
                  then (snd a) :: (retrieveRelatedStatesFromV states s)
                  else retrieveRelatedStatesFromV states s
  end.

```

```

Definition RTC (m:model) : set (state × state) :=
  set_union equiv_dec (R(Fr(m))) (set_union equiv_dec (getTransitive m) (getReflexive m)).

```

We define `singleFormulaVerify` as the top-level definition that checks whether a formula p is valid in a model m , given a set of data markup t denoting the data flow for port names of the Reo circuit. The verification employs `singleModelStep` with m as the model and p as the formula, applying it to each state denoted by t as the starting state of `singleModelStep`.

```

Definition singleFormulaVerify (m : model) (p : formula)
  (t: set dataConnector) : bool :=
  (forallb (fun x ⇒ eqb x true) (map (singleModelStep m p) (getState m t))).

```

6.3 Model Construction

In this section we detail the construction of a framework that calculates a model that accepts a formula (if exists). If the formula is not valid, the framework returns a model which is induced by the formula evaluated (by expanding all the modalities of a formula,

until every subformulae has already been reduced).

The framework is driven by the following ideas:

- For each state s reached during the execution, calculate and store the propositions (of type `dataProp`) in the valuation function V .
- For each state w reached from a state s , add (s, w) in the relation R
- Decompose the formula φ to be evaluated until all propositional symbols have been reached. Formulas with no iteration operators \star are inductively decomposed, while formulas with modalities that contain program operators \star require an upper bound n that limits its execution to a finite number of steps n . The validity of \star formulas then will be achieved (if possible) in at most n executions for each modality with \star .

The calculation of propositions that are valid in the connector considers three different scenarios:

1. The data item D_A is in port A (i.e., propositions like $D_A = 1$, where 1 is the data item)
2. Ports A and B have the same data item at the same time (i.e., propositions like $D_A = D_B$)
3. The data item D is in a buffer (FIFO connector)

The implementation of the item 1 starts with function `retrieveSinglePortProp`. It is a function that with a set of data flows t , a state denoted as a natural number $index$, and a port name n returns a proposition `dataPorts n x` if t describes a data flow of port n with data item x .

```

Fixpoint retrieveSinglePortProp ( $t$  : set (dataConnector name data)) ( $index$  : nat)
( $n$  : name) : set (dataProp name data) :=
  match  $t$  with
  | []  $\Rightarrow$  []
  |  $a :: t' \Rightarrow$  match  $a$  with
    | dataPorts  $a$   $x \Rightarrow$  if ( $n == a$ ) then [dataInPorts  $n$   $x$ ]
      else retrieveSinglePortProp  $t'$   $index$   $n$ 
    | fifoData  $a$   $x$   $b \Rightarrow$  retrieveSinglePortProp  $t'$   $index$   $n$ 
    end
  end.

```

Item 2 is then implemented by means of `portsHaveTheSameData`, a function that with two data flows $n1$ and $n2$ creates a proposition `dataInPorts data a b` if they have the same data item in their flow.

Definition `portsHaveSameData` ($n1 : (\text{dataConnector } name \text{ data})$) ($n2 : (\text{dataConnector } name \text{ data})$) : `set (dataProp name data)` :=

```
match n1, n2 with
| dataPorts a x, dataPorts b y => if x == y then [dataBothPorts data a b] else []
| -, - => []
end.
```

Function `portsHaveSameData` is then extended to a set of data flows t with the implementation of `retrieveTwoPortsProp`. It checks in the whole data flow of the model at a specific state $index$ as t to build valid propositions of type $D_A = D_B$ in state $index$.

Fixpoint `retrieveTwoPortsProp` ($index : nat$) ($t : \text{set (dataConnector name data)}$) ($n : (\text{dataConnector name data})$) : `set (dataProp name data)` :=

```
match t with
| [] => []
| a :: t' => portsHaveSameData a n ++ (retrieveTwoPortsProp index t' n)
end.
```

Then, `retrieveFIFOdataProp` recovers propositions regarding the data within buffers as in item 3. Given a data flow t and a state $index$, `retrieveFIFOdataProp` checks in t if there is a flow denoting data stored within FIFO connectors. For each data item found, it creates a proposition stating that D_A (D_A as the data item in the flow) is in FIFO within port names a and b .

Fixpoint `retrieveFIFOdataProp` ($index : nat$) ($t : \text{set (dataConnector name data)}$) ($n : \text{dataConnector name data}$) : `set (dataProp name data)` :=

```
match t with
| fifodata :: t' => match fifodata with
| fifoData a x b => if (equiv_decb fifodata n)
then [dataInFifo a x b] ++
(retrieveFIFOdataProp index t' n)
else (retrieveFIFOdataProp index t' n)
| dataPorts a b => (retrieveFIFOdataProp index t' n)
end
| [] => []
end.
```

These functions are then joined by `buildValidPropositions`, a definition which unifies the three possible sets of propositions created by `retrieveSinglePortProp`, `portsHaveSameData`, and `retrieveFIFOdataProp`. It will process each of the aforementioned functions, considering all port names of the Reo model as N .

Definition `buildValidPropositions` (N : set name) ($index$: nat) (t : set (dataConnector name data)) : set (dataProp name data) :=

```

  ((flat_map(retrieveTwoPortsProp index t) (t))) ++
  (flat_map(retrieveSinglePortProp t index) (N)) ++
  (flat_map(retrieveFIFOdataProp index t) t).

```

The construction of the model effectively is split into two parts. When evaluating a program Π and a formula φ , we first construct a model \mathcal{M}_0 considering a starting state 0 as the state where a data flow $t \in \delta(0)$, and the valid proposition in 0 are the ones derived by `buildValidPropositions` considering t . If φ is a proposition, then there is no further processing to be performed and \mathcal{M}_0 is returned. Otherwise, considering \mathcal{M}_0 as the starting point, φ is decomposed and processed, so for each state j reached, their corresponding information is added to the model (the state, the pair (i, j) is added to R_Π as the model's transition relation, and the valid propositions of the reached state based on $\delta(j)$). this search is done in a depth-search fashion employing the definitions we detail as follows.

We proceed by formalizing a definition which will be useful when formalizing the resulting model \mathcal{M}_0 's valuation function V . `getProp` will be employed to return whether a proposition n is in a set of propositions `setProp`, where `setProp` denotes $V(0, n)$.

Fixpoint `getProp` (`setProp`: set (dataProp name data)) (n : (dataProp name data)) : bool :=

```

  match setProp with
  | [] => false
  | a::t => if (equiv_decb a n) then true else getProp t n
  end.

```

Then, definition `getValFunctionProp` will employ `getProp` to define a valuation function for \mathcal{M}_0 with the same parameters V expects.

Definition `getValFunctionProp` (N : set name) (t :set (dataConnector name data)) ($index$: nat) (s :nat) (p :(dataProp name data)) :=

```

  getProp ((buildValidPropositions N index t)) p.

```

The Frame \mathcal{F} for model \mathcal{M}_0 is formalized by `buildPropFrame`, which yields a Coq model as in Definition 6.1.1 as follows. Function `setStateForProp` returns a set contain-

ing a single state s as the set S of states of the model, while notation $[]$ is an empty set of relations R , `lambdaForProp` returns a standard `lambda` for the data flow t , and `deltaForProp` returns t as the state's own data markup, as there is no other states to be reached.

Definition `buildPropFrame` (t : set (dataConnector name data)) (s :nat) :=
`mkframe` (setStateForProp s) (`[]`) (`lambdaForProp`) (`deltaForProp` t).

Then, \mathcal{M}_0 is defined by `buildPropModel`, which follows Definition 6.1.2 with a valuation function yielded by `getValFunctionProp`.

Definition `buildPropModel` (N : set name)(t : set (dataConnector name data))
(s :nat) := `mkmodel` (`buildPropFrame` t s) (`getValFunctionProp` N t s).

The definitions detailed above comprise the first part of the model construction framework. We focus now on the second half, which aims to provide the construction of models for formulas φ which are not propositions with \mathcal{M}_0 as a basis for the processing. The development will build intermediate models $\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}$ until the end of the processing is reached (i.e., the formula has been fully reduced). Each \mathcal{M}_i is the result of evaluating the current set of states and the possible data flows the model is currently in, which will culminate in \mathcal{M} as the resulting Coq model.

The formalization proceeds by formalizing a function which retrieves the valid propositions of a given state n . For the whole model \mathcal{M} , we implement the structure of the propositions as a set of pairs s, p , meaning that a proposition p is valid at state s . Therefore, `getValFunction` with a proposition $prop$, a state $state$, and a set of propositions $props$ returns a set containing all propositions in $props$ which are valid in the current state, i.e., the first value of the .

Fixpoint `getValFunction` ($props$: set (nat \times (set (dataProp name data))))
($state$: nat) ($prop$: (dataProp name data)) :=
`match` $props$ `with`
| `[]` \Rightarrow `false`
| $stateAndProp :: moreProps$ \Rightarrow `if` `fst`($stateAndProp$) == $state$ `then`
`set_mem` `equiv_dec` ($prop$) (`snd`($stateAndProp$))
`else` `getValFunction` $moreProps$ $state$ $prop$
`end`.

The valuation function needs to be updated to consider each reached state and their corresponding valid propositions for each verification step. To this extent, we formalize a Coq `Record` that keeps track of the propositions and the states they are valid on as `calcProps`. This information is constantly updated in the model being constructed.

```

Record calcProps := mkcalcProps {
  statesAndProps : set (nat × set (dataProp name data));
  propCounter : nat
}.

```

Definition `addInfoToModel` adds this information to intermediate models \mathcal{M}_i . Given a intermediate model m as \mathcal{M}_{i-1} , the current state being evaluated as $dest$ and the state considered when evaluating M_{i-1} (i.e., the state that $dest$ was reached from) as $origin$, t as the resulting set of data flows in the Reo model, by firing $f(t_{i-1}, \Pi)$, and $dataMarkups$ as the current set of states and the propositions that are valid on them, `addInfoToModel` adds $dest$ as a state of the model in S , a pair $(origin, dest)$ denoting that the states are directly related in R , the new data markup function δ for \mathcal{M}_i is calculated for the state $dest$ based on t , resulting on the frame \mathcal{F} constructed by `mkframe`.

```

Definition addInfoToModel (m: model name nat data) (origin:nat) (dest: nat)
  (N: set name) (t: (set (dataConnector name data)))
  (dataMarkups : (set (nat × (set (dataConnector name data))))) (calc : calcProps) :=
mkmodel
  (mkframe (set_add equiv_dec dest (S(Fr(m))))
    (set_add equiv_dec (origin, dest) (R(Fr(m)))) (lambda(Fr(m)))
    (getDelta dataMarkups))
  (getValFunction (statesAndProps (getNewValFunc calc N [t] dest))).

```

The model \mathcal{M}_i is constructed with the resulting frame \mathcal{F} as aforementioned with `mkmodel`, with its resulting valuation function considering all the states already visited and their corresponding valid propositions stored in $calc$ by means of `getNewValFunc`. Therefore, `addInfoToModel` recalculates the resulting valuation function by creating a new `calcProps`, in which the new information regarding the propositions valid in state $dest$ are added to the model. The propositions valid in $dest$ are calculated by `buildValidPropositions` considering each element of the data flow t . Its result is then sent back to `addInfoToModel`, which adds the resulting set of valid propositions as its valuation function, by retrieving the value of the field `statesAndProps` of the `calcProps` returned by `getNewValFunc`.

```

Definition getNewValFunc (calc: calcProps) (N: set name)
  (t:set (set (dataConnector name data))) (state: nat) :=
mkcalcProps (set_add equiv_dec (state, flat_map
  (buildValidPropositions N (propCounter(calc)) t) (statesAndProps calc))
  (propCounter(calc) + length (flat_map (buildValidPropositions N state) t))).

```

Definition `addInfoToModel` lets the formalization consider all possible states that can

be reached during the model construction. Function `processIntermediateStep` extends the usage of `addInfoToModel` to add all possible “next states” that can be reached from *origin*: with *m* as the last obtained model \mathcal{M}_{i-1} , *N* as the set of port names that compose the model, *visitedStates* as a set that tracks all states that have been already visited by the model construction process, *calcProps* as the auxiliary structure that keeps track of the states and their valid propositions, and *nextSetOfStates* as the set of states reached by $f(t, \Pi)$ from the current state *origin*, `addInfoToModel` employs `addInfoToModel` for each state reached from *index* in *nextSetOfStates*, where each element of *nextSetOfStates* are of form $(s, dataMarkup)$. Therefore, $(fst(currentState))$ is the state itself, and $(snd(currentState))$ is the data flow that identifies such state.

```

Fixpoint processIntermediateStep (m: model name nat data) (origin : nat)
  (N: set name) (visitedStates : (set (nat × (set (dataConnector name data)))))
  (calc : calcProps) (nextSetOfStates : set (nat × set (dataConnector name data)))
:= match nextSetOfStates with
| [] ⇒ (m, (visitedStates, calc))
| currentState :: moreStates ⇒
    processIntermediateStep
      (addInfoToModel m origin (fst(currentState)) N (snd(currentState))
      (set_add equiv_dec ((fst(currentState)), (snd(currentState))) visitedStates)
      calc) origin N
      (set_add equiv_dec ((fst(currentState)), (snd(currentState))) visitedStates)
      (getNewValFunc calc N [(snd(currentState))] (fst(currentState)))
      moreStates
end.

```

Then, `processGeneralStep` uses `processIntermediateStep` to consider all current reached states, i.e., all possible states to which new states can be reached from the current processing step. It decomposes the current set of states to be evaluated as pairs $(s, dataMarkup)$ as elements of *currentSetofStates*. For each pair $(s, dataMarkup)$, `processGeneralStep` calls `processIntermediateStep` with its corresponding parameters as follows: *m* is the current model to which information will be added, *index* as each of the current states to be evaluated (i.e, the first element of the pair $(s, dataMarkup)$ — *s*), *N* as the set of port names of the Reo model, *visitedStates* as the set of all visited states so far by the process, *calc* as the supporting structure that holds information about all states and the propositions valid on them, and *nextSetOfstates* as the resulting states that can be reached from firing the current state *s*’s corresponding data markup *dataMarkup* as the second

element of the pair $(s, dataMarkup)$, i.e., $f(dataMarkup, \Pi)$.

```

Fixpoint processGeneralStep (m: model name nat data) (N: set name)
  (visitedStates : (set (nat × (set (dataConnector name data)))))
  (calc : calcProps) (currentSetOfStates : set (nat × set (dataConnector name data)))
  (pi : (reoProgram name data)) (index : nat) :=
match currentSetOfStates with
| [] ⇒ (m, (visitedStates, (index, calc)))
| currentState :: moreStates ⇒
    processGeneralStep (fst(processIntermediateStep m
      (fst(currentState)) N visitedStates calc
      (getNewIndexesForStates (f([snd(currentState)] )
      (program2SimpProgram (pi))) visitedStates index))) N
      (fst(snd(processIntermediateStep m (fst(currentState)) N visitedStates calc
      (getNewIndexesForStates (f([snd(currentState)] )
      (program2SimpProgram (pi))) visitedStates index))))
      (snd(snd(processIntermediateStep m (fst(currentState)) N visitedStates calc
      (getNewIndexesForStates (f([snd(currentState)] )
      (program2SimpProgram (pi))) visitedStates index)))) (moreStates) pi
      (calculateAmountNewStates (fst(snd(processIntermediateStep m
      (fst(currentState)) N visitedStates calc
      (getNewIndexesForStates (f([snd(currentState)] )
      (program2SimpProgram (pi))) visitedStates index)))) index)
end.

```

To obtain the corresponding indexes for new states reached by $f(dataMarkup, \Pi)$ to `processIntermediateStep` as its parameter `nextSetOfStates` employs `getNewIndexesForStates`. it is a function which will consider the creation of fresh indexes to denote states that have not been visited yet. If $f(dataMarkup, \Pi)$ results in some already visited state, then `nextSetOfStates` will yield the index of the already visited state. It uses `liftIndex`, a function which will either retrieve a new index for the next visited state (if it has not been visited), or return the corresponding state's index from the set of already visited states.

```

Definition getNewIndexesForStates (t : set (set (dataConnector name data)))
  (visStates: set (nat × (set (dataConnector name data)))) (index : nat) :=
  liftIndex index (getVisitedStates t visStates).

```

Function `getNewIndexesForStates` also employs `getVisitedStates`, which based on the

data flow t and the set of already visited states $visStates$ returns the indexes of states already depicted by data flows in t .

```

Fixpoint liftIndex (index : nat)
  (currentStates : set (option nat × set (dataConnector name data))) :=
  match currentStates with
  | [] ⇒ []
  | visState :: moreStates ⇒ match (fst(visState)) with
    | None ⇒ (index, (snd(visState))) ::
      (liftIndex (Datatypes.S index) moreStates)
    | Some a ⇒ (a, (snd(visState))) :: (liftIndex index moreStates)
  end

end.

```

The next step is to formalize the top level definition which will use `processGeneralStep` to implement the search to retrieve the resulting model after processing the formula. Definition `getModel` is the main function that recursively searches for a *ReLo* model induced by ϕ . Given a set of port names n , a data flow t it will decompose the formula until all atomic subformulas of ϕ have been processed.

The model construction relies on *processGeneralStep*, which updates the model for each modality processed. The remainder of the parameters supports the process as follows: m is the intermediate model obtained so far, $index$ is the next available number to denote a state in the model, $setStates$ is the set of pairs $(state, t)$ of all states visited and their corresponding data flow, $calc$ is the auxiliary structure `calcProps` which holds the states and which propositions are valid on them, and $upperBound$ establishes a limit of processing of each iteration found in the modalities that compose ϕ .

```

Fixpoint getModel (m: model name nat data) (n: set name)
  (t: set (set (dataConnector name data))) (index:nat)
  (phi: (formula name data)) (setStates: set (nat × (set (dataConnector name data))))
  (calc : calcProps) (upperBound : nat) :=
  match phi with
  | proposition p ⇒ m
  | chiFormulaBox x phi | chiFormulaBox x phi ⇒ m
  | diamond t' pi p ⇒ match pi with
    | sProgram pi' ⇒
      getModel (fst(processGeneralStep m n setStates calc (getNewIndexesForStates t setStates index) pi' ( index) )) n

```

```

      (f(t)(program2SimpProgram (pi'))))
      (fst(snd(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi' (index) ))))
      p (fst(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi' index)))
      (snd(snd(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi' (index)))))) upperBound
    | star pi' ⇒
      fst(expandStarFormulas m n (t) (index) phi (setStates) (calc)
upperBound)
    end
  | box t' pi p ⇒ match pi with
    | sProgram pi' ⇒ getModel (fst(processGeneralStep m n setStates
calc (getNewIndexesForStates t setStates index) pi'
  ( index) )) n
      (f(t)(program2SimpProgram (pi'))))
      (fst(snd(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi' (index) ))))
      p (fst(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi' index)))
      (snd(snd(snd(processGeneralStep m n setStates calc (getNewIndex-
esForStates t setStates index) pi'
        (index)))))) upperBound
    | star pi' ⇒
      fst(expandStarFormulas m n (t) (index) phi (setStates)
(calc) upperBound)
    end
  | and a b | or a b | imp a b | bimpl a b ⇒ (getModel (getModel m n t index a
setStates calc upperBound) n t index b setStates calc) upperBound
  | neg a ⇒ (getModel m n t index a setStates calc upperBound)
end.

```

The top-level function that employs `getModel` is formalized as `constructModel`, that defines as parameters n as the set of port names of the Reo model, t the starting data flow of the process, and phi the formula to be evaluated. It calls `getModel` considering an initial model m supplied by `buildPropModel` as one that contains a single state where the

propositions derived from t are valid.

Definition `constructModel` (n : set name) (t : set (set (dataConnector name data)))
 (ϕ : (formula name data)) ($upperBound$: nat) :=
`getModel` (`buildPropModel` n (`hd` [] t) 0) n t 1 ϕ
 (`getNewIndexesForStates` t [] 0) (`mkcalcProps` [] 0) $upperBound$.

6.4 A tableau for *ReLo* in Coq

We provide an implementation in Coq of the core definitions of a *ReLo* Tableau as in Section 5.4. The development presented in this section aims to provide means for unfolding the tableau during the proof process, to check whether a branch is closed and even whether a tableau is open or closed, considering the particularities when it comes to formulae \mathcal{X}_\Diamond and \mathcal{X}_\Box . These functionalities will allow us to check whether the formulae is valid or not, following Definition 5.4.6.

The development starts by structuring proof trees as `binTree`, a binary tree structure which nodes have content of type $(w : \varphi : \gamma)$, where w is a label (state) of where the formula φ has value $\gamma \in \{T, F\}$.

Inductive `binTree` : Type :=
 | `nilLeaf` : binTree
 | `leaf` : (state \times (((formula name data)) \times bool)) \rightarrow binTree
 | `node` : (state \times (((formula name data)) \times bool)) \rightarrow binTree \rightarrow binTree \rightarrow binTree.

A proof tree denoted by *binTree* has three type of nodes:

- `nilLeaf` denoting an empty node, required for the definition of `binType` not to be ill-formed.
- `leaf` as a leaf node, with no nodes reachable from them.
- `node` as an intermediate node of the proof tree. This node type may have up to two descending nodes, either empty nodes (denoting no nodes in this branch), leaf nodes or intermediate nodes.

Note that `leaf` could be formalized in terms of `node` with both descendant nodes as `nilLeaf`. However, to ease the proof tree reading and processing, we decided to split leaf and intermediate nodes by using different type constructors.

With *binType*, we proceed to formalize *ReLo* Tableau as Definition 5.4.1 in Coq, a **Record** detailed in Definition 6.4.1.

Definition 6.4.1 (*ReLo* Tableau in Coq). A tableau in Coq is depicted as *tableau* as a structure containing two fields: *proofTree* as the tree structure to syntactically reason over *ReLo* formulae, and *statesTree* as the auxiliary states structure that holds the information of how states used in the proof relate to each other.

```
Record tableau := mkTableau {
  proofTree : binTree;
  statesTree : set (state × state)
}.
```

We continue by formalizing a function that constructs tableau from Coq *ReLo* formulas as detailed in Section 6.3, with the inductive type *formula*. The smallest tableau for a formula φ is a proof tree rooted with φ as F in a state w , which is also the starting point for tableau proofs. Therefore, *formula2Tableau* builds a tableau by supposing it is false in a state 0 as the starting point of the proof.

```
Definition formula2Tableau (phi: formula name data) :=
  mkTableau (leaf (0, (phi, false))) ([]).
```

The two following definitions are support functions to operate on the proof trees as *binTree*. Function *searchBinTree*'s objective is to recursively traverse the tree as a whole, searching whether there is a node with its contents equal to the ones provided in variable *nodeContent* (i.e., the same state, the same formula and its value in the state). It returns *false* if no node in proof tree t satisfying these conditions are found.

```
Fixpoint searchBinTree (t: binTree nat name data)
  (nodeContent : nat × (((formula name data)) × bool)) :=
  match t with
  | nilLeaf _ _ _ => false
  | leaf phi => (equiv_decb (fst(nodeContent)) (fst(phi))) && (equiv_decb (fst(snd(nodeContent)))
    (fst(snd(phi)))) && (equiv_decb (snd(snd(nodeContent))) (snd(snd(nodeContent))))
  | node phi a b => if (equiv_decb (fst(nodeContent)) (fst(phi))) && (equiv_decb
    (fst(snd(nodeContent))) (fst(snd(phi))))
    && (equiv_decb (snd(snd(nodeContent))) (snd(snd(nodeContent))))
    then true
    else searchBinTree a nodeContent ||
      searchBinTree b nodeContent
  end.
```

Alternatively, *searchBinTreeNode* works similarly to *searchBinTree*, but instead of returning boolean values if the node is found in t , *searchBinTreeNode* returns the node

itself as a `binTree` structure. It employs `option` type from Coq standard library to denote cases when the node is not found.

```

Fixpoint searchBinTreeNode (t: binTree nat name data)
  (nodeContent : nat × (((formula name data)) × bool)) :=
  match t with
  | nilLeaf _ _ _ ⇒ None
  | leaf phi ⇒ if (equiv_decb (fst(nodeContent)) (fst(phi)))
    && (equiv_decb (fst(snd(nodeContent))) (fst(snd(phi))))
    && (equiv_decb (snd(snd(nodeContent))) (snd(snd(nodeContent))))
    then Some (leaf phi) else None
  | node phi a b ⇒ if (equiv_decb (fst(nodeContent)) (fst(phi)))
    && (equiv_decb (fst(snd(nodeContent))) (fst(snd(phi))))
    && (equiv_decb (snd(snd(nodeContent))) (snd(snd(nodeContent))))
    then Some (leaf phi)
    else if searchBinTree a nodeContent
    then searchBinTreeNode a nodeContent
    else searchBinTreeNode b nodeContent
  end.

```

We proceed by providing definitions that will be employed in the tableau proof process, regarding the construction of the proof tree after applying a tableau rule as proposed in Section 5.4. Function `addLeftToTableau` adds non-branching rules to the proof tree specified in *t*. Parameter *t'* is the placeholder to which the result of the application of a rule in the proof tree *t* will be supplied. *leafNode* is a variable that denotes the leaf node of the branch the derivation in *t'* must be added. Function `equalFormula` is a function that basically calls `equiv_decb` to check whether two formulae of type `formula` are equal.

```

Fixpoint addLeftToTableau (t : binTree nat name data)
  (t' : binTree nat name data) (leafNode : nat × (((formula name data)) × bool)) :
  (binTree nat name data) :=
  match t with
  | nilLeaf _ _ _ ⇒ t
  | leaf phi ⇒ if (equiv_decb (fst(leafNode)) (fst(phi)))
    && (equalFormula (fst(snd(leafNode))) (fst(snd(phi))))
    && (equiv_decb (snd(snd(leafNode))) (snd(snd(phi))))
    then ((node phi) t' (nilLeaf _ _ _))
    else (leaf phi)

```

```

| node phi a b => (node phi) (addLeftToTableau a t' leafNode)
                      (addLeftToTableau b t' leafNode)

end.

```

In addition to `addLeftToTableau`, we also provide `addBranchLeftToTableau` as a function which adds branching rules to the proof tree specified in t , complementing `addLeftToTableau`. Parameters $b1$ and $b2$ respectively denote the left and right branches generated by branching rules as in Definition 5.4.2. They are added to the end of the branch denoted by $leafNode$, in a similar fashion as `addLeftToTableau` does, but adding each branch as a different branch.

```

Fixpoint addBranchLeftToTableau (t : binTree nat name data)
  (b1 : binTree nat name data) (b2 : binTree nat name data)
  (leafNode : nat × (((formula name data)) × bool)) : (binTree nat name data) :=
match t with
| nilLeaf _ _ => t
| leaf phi => if (equiv_decb (fst(leafNode)) (fst(phi)))
               && (equalFormula (fst(snd(leafNode))) (fst(snd(phi))))
               && (equiv_decb (snd(snd(leafNode))) (snd(snd(phi))))
               then ((node phi) (b1) (b2))
               else (leaf phi)
| node phi a b => ((node phi) (addBranchLeftToTableau a b1 b2 leafNode)
                    (addBranchLeftToTableau b b1 b2 leafNode))

end.

```

The formalization proceeds by defining now the core function in applying tableau rules. Function `tableauRules` is responsible for implementing the map between the formula in a tableau node, and mapping the result of the application as new node(s) appended to the end of a branch of the proof tree. It takes as parameters t as the proof tree of a tableau \mathcal{T} , $origT$ as a copy of t which will be used to produce the final tree, $statesTree$ as the set of accessed states of the tableau, $nodeContent$ as the formula which one wants to apply the rule, $state$ as the next index available for a new state (as rules \Box -F and \Diamond -T require a new, not visited state), $destState$ as the state where formulas \Diamond -F and \Box -T will be expanded to (since their resulting formulae can employ an already visited state or a new one), $indexchiFormulaBox$ and $indexchiFormulaBoxmond$ respectively as the current index to keep track of how many \mathcal{X}_{\Box} and \mathcal{X}_{\Diamond} have already been instantiated, and $leafNode$ as the leaf node of the branch where the result of the rule application must be added in

the proof tree.

With these parameters, `tableauRules` searches in the proof tree t in the branch ended by `leafNode` if there is a node with content denoted by `nodeContent` as a node $(w : \varphi : \gamma)$. By matching the combination of formula φ and γ as one of the rules in Definition 5.4.2, `tableauRules` returns the result of the rule application as a new tableau \mathcal{T}' with the rule's result appended to `leafNode`. For each `formula` constructor, `tableauRules` implements the rule application result considering both T and F rules. The T case is considered in the `if (snd(snd(phi)))` clauses after each of the cases, and the F case in the corresponding `else`.

Definition `tableauRules`

```
(t: binTree nat name data) (origT: binTree nat name data)
(statesTree : set (nat × nat))
(nodeContent : nat × (((formula name data)) × bool))
(state : nat) (destState : nat) (indexchiFormulaBox : nat)
(indexchiFormulaBoxmond : nat)
(leafNode : nat × (((formula name data)) × bool)) :=
match (searchBinTreeNode t nodeContent) with
| None ⇒ (origT, statesTree)
| Some tx ⇒
match tx with
| nilLeaf _ _ ⇒ (tx, statesTree)
| leaf phi ⇒ if (equiv_decb (fst(nodeContent)) (fst(phi)))
&& (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
&& (equiv_decb (snd(snd(nodeContent))) (snd(snd(phi))))
then match (fst(snd(phi))) with
| proposition p ⇒
(origT, statesTree)
| chiFormulaBox n phi' ⇒ if negb (snd(snd(phi))) then
match phi' with
| box t' pi p ⇒ match pi with
| star pi' ⇒ ((addBranchLeftToTableau (origT)
(leaf ((fst(phi)), (p, false))) (node ((fst(phi)), (p, true)))
(leaf ((fst(phi)),
((box t' pi) (chiFormulaBox indexchiFormulaBox phi'), false)))
(nilLeaf nat name data)) leafNode), (statesTree))
```

```

      | sProgram pi' ⇒ (origT , statesTree)
    end
  | _ ⇒ (origT , statesTree)
end
else (origT , statesTree)
| chiFormulaBox n phi' ⇒ if (snd(snd(phi))) then
  match phi' with
  | diamond t' pi p ⇒ match pi with
    | star pi' ⇒ ((addBranchLeftToTableau (origT)
      (leaf ((fst(phi)) , (p , true)))
      (node ((fst(phi)) , (p , false))
      (leaf ((fst(phi)) , ((diamond t' pi)
      (chiFormulaBox indexchiFormulaBoxmond phi' ) , true)))
      (nilLeaf nat name data))) leafNode) , (statesTree))
    | sProgram pi' ⇒ (origT , statesTree)
  end
  | _ ⇒ (origT , statesTree)
end
else (origT , statesTree)
| and phi1 phi2 ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT) ((node ((fst(phi)) , (phi1 , true))
  (leaf ((fst(phi)) , (phi2 , true)))
  (nilLeaf nat name data))) leafNode) , (statesTree))
  else ((addBranchLeftToTableau (origT)
  (leaf ((fst(phi)) , (phi1 , false))
  (leaf ((fst(phi)) , (phi2 , false)) leafNode) , (statesTree))
| or phi1 phi2 ⇒ if (snd(snd(phi))) then
  ((addBranchLeftToTableau (origT) (leaf ((fst(phi)) , (phi1 , true)))
  (leaf ((fst(phi)) , (phi2 , true)) leafNode) , (statesTree))
  else ((addLeftToTableau (origT)
  ((node ((fst(phi)) , (phi1 , false))
  (leaf ((fst(phi)) , (phi2 , false))
  (nilLeaf nat name data))) leafNode) , (statesTree))
| neg phi1 ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)

```



```

      (leaf ((fst(phi)), (((phi1)) , false))) leafNode), (statesTree))
    else ((addLeftToTableau (origT)
      (leaf ((fst(phi)), (((phi1)) , true))) leafNode), (statesTree))
  | box t' pi p => match pi with
  | sProgram pi' => if (snd(snd(phi))) then
    ((addLeftToTableau (origT)
      ((leaf ((destState), (p , true)))) leafNode), (statesTree))
    else ((addLeftToTableau (origT)
      ((leaf ((state), (p , false)))) leafNode) ,
      (set_add equiv_dec ((fst(phi)), (state)) statesTree ))
  | star pi' => if (snd(snd(phi))) then
    ((addLeftToTableau (origT)
      ((node ((fst(phi)), (p , true)) (leaf ((fst(phi)),
        (((box t' (sProgram pi'))(box t' (star pi') p))) , true)))
      (nilLeaf nat name data))) leafNode), (statesTree))
    else ((addLeftToTableau (origT) ((leaf ((fst(phi)),
      ((chiFormulaBox indexchiFormulaBox p) , false))))
      leafNode), (statesTree))
    end
  | diamond t' pi p => match pi with
  | sProgram pi' => if (snd(snd(phi))) then
    ((addLeftToTableau (origT)
      ((leaf ((state), (p , true)))) leafNode) ,
      (set_add equiv_dec ((fst(phi)), (state)) statesTree ))
    else ((addLeftToTableau (origT)
      ((leaf ((destState), (p , false)))) leafNode), (statesTree))
  | star pi' => if (snd(snd(phi))) then
    ((addLeftToTableau (origT) ((leaf ((fst(phi)),
      ((chiFormulaBox indexchiFormulaBoxmond p) , true))))
      leafNode), (statesTree))
    else ((addLeftToTableau (origT)
      ((node ((fst(phi)), (p , false))
        (leaf ((fst(phi)), (((diamond t' (sProgram pi'))
          (diamond t' (star pi') p))) , false)))
      (nilLeaf nat name data))) leafNode), (statesTree))

```

```

    end
  | imp  $\phi_1 \phi_2 \Rightarrow$  if (snd(snd( $\phi$ ))) then
    ((addBranchLeftToTableau (origT)
      (leaf ((fst( $\phi$ )), ( $\phi_1$  , false)))
      (leaf ((fst( $\phi$ )), ( $\phi_2$  , true))) leafNode), (statesTree))
    else ((addBranchLeftToTableau (origT)
      ((node ((fst( $\phi$ )), ( $\phi_1$  , true))
        (leaf ((fst( $\phi$ )), ( $\phi_2$  , false)))) (nilLeaf nat name data))
      (nilLeaf nat name data) leafNode), (statesTree))
  | _  $\Rightarrow$  (tx, statesTree)

end

else (tx, statesTree)
| node  $\phi x y \Rightarrow$  if (equiv_decb (fst(nodeContent)) (fst( $\phi$ )))
  && (equalFormula (fst(snd(nodeContent))) (fst(snd( $\phi$ ))))
  && (equiv_decb (snd(snd(nodeContent))) (snd(snd( $\phi$ ))))
  then match (fst(snd( $\phi$ ))) with
| proposition  $p \Rightarrow$  (origT, statesTree)
| chiFormulaBox  $n \phi'$   $\Rightarrow$  if negb (snd(snd( $\phi$ ))) then
  match  $\phi'$  with
| box  $t' \pi p \Rightarrow$  match  $\pi$  with
| star  $\pi' \Rightarrow$  ((addBranchLeftToTableau (origT)
  (leaf ((fst( $\phi$ )), ( $p$  , false)))
  (node ((fst( $\phi$ )), ( $p$  , true))
  (leaf ((fst( $\phi$ )), ((box  $t' \pi$ 
  (chiFormulaBox indexchiFormulaBox  $\phi'$ ) , false)))
  (nilLeaf nat name data)) leafNode), (statesTree))
| sProgram  $\pi' \Rightarrow$  (origT, statesTree)
end
| _  $\Rightarrow$  (origT, statesTree)
end

else (origT, statesTree)
| chiFormulaBox  $n \phi' \Rightarrow$  if (snd(snd( $\phi$ ))) then
match  $\phi'$  with
| diamond  $t' \pi p \Rightarrow$  match  $\pi$  with
| star  $\pi' \Rightarrow$  ((addBranchLeftToTableau (origT)

```

```

      (leaf ((fst(phi)), (p , true)))
      (node ((fst(phi)), (p , false))
      (leaf ((fst(phi)), ((diamond t' pi)
      (chiFormulaBox indexchiFormulaBoxmond phi') , true)))
      (nilLeaf nat name data)) leafNode) , (statesTree))
    | sProgram pi' ⇒ (origT , statesTree)
  end
| _ ⇒ (origT , statesTree)
end
else (origT , statesTree)
| and phi1 phi2 ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)
  ((node ((fst(phi)), (phi1 , true))
  (leaf ((fst(phi)), (phi2 , true)))
  (nilLeaf nat name data))) leafNode) , (statesTree))
  else ((addBranchLeftToTableau (origT)
  (leaf ((fst(phi)), (phi1 , false))
  (leaf ((fst(phi)), (phi2 , false)) leafNode) , (statesTree))
| or phi1 phi2 ⇒ if (snd(snd(phi))) then
  ((addBranchLeftToTableau (origT)
  (leaf ((fst(phi)), (phi1 , true))
  (leaf ((fst(phi)), (phi2 , true)) leafNode) , (statesTree))
  else ((addLeftToTableau (origT)
  ((node ((fst(phi)), (phi1 , false))
  (leaf ((fst(phi)), (phi2 , false))
  (nilLeaf nat name data))) leafNode) , (statesTree))
| neg phi1 ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)
  (leaf ((fst(phi)), (((phi1)) , false)) leafNode) , (statesTree))
  else ((addLeftToTableau (origT)
  (leaf ((fst(phi)), (((phi1)) , true)) leafNode) , (statesTree))
| box t' pi p ⇒ match pi with
| sProgram pi' ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)
  ((leaf ((destState), (p , true))) leafNode) , (statesTree))

```

```

      else ((addLeftToTableau (origT)
        ((leaf ((state), (p , false)))) leafNode) ,
        (set_add equiv_dec ((fst(phi)), (state)) statesTree ))
| star pi' ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT) ((node ((fst(phi)), (p , true))
    (leaf ((fst(phi)), ((box t' (sProgram pi')
      (box t' (star pi') p)))) , true)))
    (nilLeaf nat name data))) leafNode), (statesTree))
  else ((addLeftToTableau (origT)((leaf ((fst(phi)),
    ((chiFormulaBox indexchiFormulaBox p) , true)))) leafNode),
    (statesTree))
end
| diamond t' pi p ⇒ match pi with
| sProgram pi' ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)
    ((leaf ((state), (p , true)))) leafNode) ,
    (set_add equiv_dec ((fst(phi)), (state)) statesTree ))
  else ((addLeftToTableau (origT)
    ((leaf ((destState), (p , false)))) leafNode), (statesTree))
| star pi' ⇒ if (snd(snd(phi))) then
  ((addLeftToTableau (origT)
    ((leaf ((fst(phi)),
      ((chiFormulaBox indexchiFormulaBox mond p) , true))))
    leafNode), (statesTree))
  else ((addLeftToTableau (origT) ((node ((fst(phi)), (p , true))
    (leaf ((fst(phi)), ((diamond t' (sProgram pi')
      (diamond t' (star pi') p)))) , false)))
    (nilLeaf nat name data))) leafNode), (statesTree))
end
| imp phi1 phi2 ⇒ if (snd(snd(phi))) then
  ((addBranchLeftToTableau (origT)
    (leaf ((fst(phi)), (phi1 , false)))
    (leaf ((fst(phi)), (phi2 , true))) leafNode), (statesTree))
  else ((addBranchLeftToTableau (origT)
    ((node ((fst(phi)), (phi1 , true))

```

```

      (leaf ((fst(phi)), (phi2 , false))) (nilLeaf nat name data))
      (nilLeaf nat name data) leafNode), (statesTree))
| _ => (tx , statesTree)
end
else (tx , statesTree)
end
end.

```

To summarize, Tables 6.1, 6.2, and 6.3 denote the relation of each rule with the corresponding Coq code in *applyRule* that adds it to the proof tree *t*. The tables respectively shows the rules defined for propositional connectors, modalities and program operators. The left column is the tableau rule, while the right column denotes the piece of Coq code of *applyRule* which constructs the result of its application.

A top-level function to ease the process of applying a tableau rule to one of its nodes' contents is needed, to ease its usage. Function *applyRule* is the top level function used to *tableauRules* in the tableau process. Therefore, *applyRule* has as parameters *t* as a tableau defined by type *tableau*, *nodeContent* as the content of the rule to be applied in the proof tree of *t*, *state* as the next available state, *indexchiFormulaBox* and *indexchiFormulaBoxmond* respectively as the next indexes for \mathcal{X}_{\Box} and $\mathcal{X}_{\Box\Diamond}$ formulae, and *leafNode* as the leaf node of the branch where the result of the rule application must be appended. It then will apply the rule corresponding to the content of *nodeContent*, adding its result in the resulting tableau after *leafNode*. The resulting tableau *t'* is of type *tableau* and consists of *t* with the rule applied by *tableauRules*, and the set of states updated when applicable.

Definition *applyRule* (*t*: *tableau* nat name data)

```

  (nodeContent : nat × (((formula name data)) × bool)) (state : nat)
  (indexchiFormulaBox : nat) (indexchiFormulaBoxmond : nat)
  (leafNode : nat × (((formula name data)) × bool)) :=
  (mkTableau (fst(tableauRules (proofTree(t)) (proofTree(t)) (statesTree(t)) nodeContent
    state indexchiFormulaBox indexchiFormulaBoxmond leafNode))
    (snd(tableauRules (proofTree(t)) (proofTree(t)) (statesTree(t)) nodeContent
    state indexchiFormulaBox indexchiFormulaBoxmond leafNode)))).

```

The next step in the formalization is to provide means to check whether a tableau is closed. Intuitively, a tableau is closed if all of its branches have a contradiction (i.e., the branch's leaf node formula must contradict some formula in its branch). The idea we implement is to retrieve all leaf nodes, and for each leaf node found, get its corresponding

| Tableau Rule | Coq Tableau code |
|--|---|
| $\frac{w: \varphi \wedge \psi : T}{\begin{array}{l} w: \varphi : T \\ w: \psi : T \end{array}}$ $\frac{w: \varphi \wedge \psi : F}{\begin{array}{l} w: \varphi : F \\ w: \psi : F \end{array}}$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) ((node ((fst(phi)), (phi1 , true)) (leaf ((fst(phi)), (phi2 , true))) (nilLeaf nat name data)))) leafNode), (statesTree)) else ((addBranchLeftToTableau (origT) (leaf ((fst(phi)), (phi1 , false))) (leaf ((fst(phi)), (phi2 , false))) leafNode), (statesTree)) </pre> |
| $\frac{w: \varphi \vee \psi : T}{\begin{array}{l} w: \varphi : T \\ w: \psi : T \end{array}}$ $\frac{w: \varphi \vee \psi : F}{\begin{array}{l} w: \varphi : F \\ w: \psi : F \end{array}}$ | <pre> if (snd(snd(phi))) then ((addBranchLeftToTableau (origT) (leaf ((fst(phi)), (phi1 , true))) (leaf ((fst(phi)), (phi2 , true))) leafNode), (statesTree)) else ((addLeftToTableau (origT) ((node ((fst(phi)), (phi1 , false)) (leaf ((fst(phi)), (phi2 , false))) (nilLeaf nat name data)))) leafNode), (statesTree)) </pre> |
| $\frac{w: \neg \varphi : T}{w: \varphi : F}$ $\frac{w: \neg \varphi : F}{w: \varphi : T}$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) (leaf ((fst(phi)), ((phi1)) , false))) leafN- ode), (statesTree)) else ((addLeftToTableau (origT) (leaf ((fst(phi)), ((phi1)) , true))) leafN- ode), (statesTree)) </pre> |
| $\frac{w: \varphi \rightarrow \psi : T}{\begin{array}{l} w: \varphi : F \\ w: \psi : T \end{array}}$ $\frac{w: \varphi \rightarrow \psi : F}{\begin{array}{l} w: \varphi : T \\ w: \psi : F \end{array}}$ | <pre> if (snd(snd(phi))) then ((addBranchLeftToTableau (origT) (leaf ((fst(phi)), (phi1 , false))) (leaf ((fst(phi)), (phi2 , true))) leafNode), (statesTree)) else ((addBranchLeftToTableau (origT) ((node ((fst(phi)), (phi1 , true)) (leaf ((fst(phi)), (phi2 , false))) (nilLeaf nat name data))) (nilLeaf nat name data) leafNode), (statesTree)) </pre> |

Table 6.1: Summarization of *applyRule* per tableau rule — propositional Rules.

branch to check whether it has a contradiction. If there is an open branch, then the tableau is not closed, and the formulae is not valid in the proposed tableau.

The set of leaf nodes is retrieved by `getAllLeafNodes`, a function that given a proof

tree t returns all leaf nodes found in it.

```

Fixpoint getAllLeafNodes (t: binTree nat name data) : set (binTree nat name data)
:= match t with
| nilLeaf _ _ _  $\Rightarrow$  []
| leaf phi  $\Rightarrow$  [leaf phi]
| node phi a b  $\Rightarrow$  (getAllLeafNodes a) ++ (getAllLeafNodes b)
end.

```

We proceed with `getBranch` to enable a branch by branch analysis on whether a tableau is closed or not. Therefore, `getBranch` with a proof tree t and a leaf node as `nodeContent`, `getBranch` yields a branch which the leaf node has the content denoted by `nodeContent`.

```

Fixpoint getBranch (t: binTree nat name data)
(nodeContent : nat  $\times$  (((formula name data))  $\times$  bool)) :=
match t with
| nilLeaf _ _ _  $\Rightarrow$  t
| leaf phi  $\Rightarrow$  if (equiv_decb (fst(nodeContent)) (fst(phi)))
&& (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
&& (equiv_decb (snd(snd(nodeContent))) (snd(snd(nodeContent))))
then (leaf phi) else (nilLeaf nat name data)
| node phi a b  $\Rightarrow$  if searchBinTree a nodeContent
then (node phi (getBranch a nodeContent) (nilLeaf nat name data))
else getBranch b nodeContent
end.

```

This definition is followed by `isBranchContradictory'`, a function that checks within a branch t whether there is a contradiction with its leaf node as `nodeContent`, returning true if a contradiction is found, and false otherwise. The contradiction is depicted by the boolean disjunctions denoted as `||`.

```

Fixpoint isBranchContradictory' (t: binTree nat name data)
(nodeContent : nat  $\times$  (((formula name data))  $\times$  bool)) :=
match t with
| nilLeaf _ _ _  $\Rightarrow$  false
| leaf phi  $\Rightarrow$  (equiv_decb (fst(nodeContent)) (fst(phi)))
&& (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
&& ((equiv_decb (snd(snd(nodeContent))) (negb(snd(snd(phi))))) ||
(equiv_decb (negb(snd(snd(nodeContent)))) (snd(snd(phi)))))

```

```

| node phi a b => if (equiv_decb (fst(nodeContent)) (fst(phi)))
    && (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
    && ((equiv_decb (snd(snd(nodeContent))) (negb(snd(snd(phi)))))
    || ((equiv_decb (negb(snd(snd(nodeContent)))) (snd(snd(phi)))))
    then true
    else (isBranchContradictory' a nodeContent)
    || (isBranchContradictory' b nodeContent)

end.

```

This definition also comes with a dual that retrieves the nodes where a contradiction was found in the branch, instead of only finding that the branch has a contradiction. Function `retrieveContradictoryNodes` returns a pair containing the node that contradicted the leaf node `nodeContent` if the branch has a contradiction. Otherwise, it returns `None` to denote no contradictory nodes in the branch `t` exists.

```

Fixpoint retrieveContradictoryNodes (t: binTree nat name data)
  (nodeContent : nat × (((formula name data)) × bool)) :=
  match t with
  | nilLeaf _ _ => None
  | leaf phi => if (equiv_decb (fst(nodeContent)) (fst(phi)))
    && (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
    && ((equiv_decb (snd(snd(nodeContent))) (negb(snd(snd(phi)))))
    || ((equiv_decb (negb(snd(snd(nodeContent)))) (snd(snd(phi)))))
    then Some (phi, nodeContent) else None
  | node phi a b => if (equiv_decb (fst(nodeContent)) (fst(phi)))
    && (equalFormula (fst(snd(nodeContent))) (fst(snd(phi))))
    && ((equiv_decb (snd(snd(nodeContent))) (negb(snd(snd(phi)))))
    || ((equiv_decb (negb(snd(snd(nodeContent)))) (snd(snd(phi)))))
    then Some (phi, nodeContent)
    else if searchBinTree a nodeContent
    then retrieveContradictoryNodes a nodeContent
    else retrieveContradictoryNodes b nodeContent

end.

```

The next step is to enable the verification of whether the tableau is either closed or open. First, it is necessary to analyze a single branch of the tableau. Then, it is extended to comprise the tableau as a whole. We also need to consider the cases where the leaf node of the tableau contains formulae $\mathcal{X}_{\langle \rangle}$ or \mathcal{X}_{\square} , where the condition for the branch to

be ignorable (presented in Section 5.4) must be satisfied.

We proceed by collecting all states in the tableau with *getStates*. This function with *t* as the branch to be evaluated returns all different states present in the corresponding branch, which will be useful when searching in the branch for states which are copy of the state of the leaf nodes, when their formulas are of type \mathcal{X}_{\Diamond} or \mathcal{X}_{\Box} .

```

Fixpoint getStates (t: binTree nat name data) : set (nat) :=
  match t with
  | nilLeaf _ _ _ => []
  | leaf phi => [fst(phi)]
  | node phi a b => set_union equiv_dec
    (set_union equiv_dec ([fst(phi)]) (getStates a)) (getStates b)
  end.

```

The next definition is bound to deal with eventualities \mathcal{X}_{\Diamond} or \mathcal{X}_{\Box} . Given a branch as *t* and a state denoted by a natural number as *state*, *getFormulae* returns a set of all formulae of *t* indexed by state *state*. it will be used to find whether there is a state *w* in branch *t* with the set of formulas

```

Fixpoint getFormulae (t : binTree nat name data) (state:nat) :
  set (formula name data) :=
  match t with
  | nilLeaf _ _ _ => []
  | leaf phi => if (fst(phi)) == state then [fst(snd(phi))] else []
  | node phi a b => if (fst(phi)) == state
    then set_union equiv_dec (set_union equiv_dec ([fst(snd(phi))])
      (getFormulae a state)) (getFormulae b state)
    else set_union equiv_dec (getFormulae a state)
      (getFormulae b state)
  end.

```

Therefore, *checkCopyState* is a function that searches whether there is a set of states *statesFromBranch* of the branch *t* with their formulas equal to the set of formulae *formulaStateQui*. If there is a state $n \in \text{statesFromBranch}$ which contains exactly the formulae *formulaStatesQui* and vice-versa, *checkCopyState* yields **true** as its result.

```

Fixpoint checkCopyState (t : binTree nat name data) (statesFromBranch : set nat)
  (formulaStateQui : set (formula name data)) : bool :=
  match statesFromBranch with
  | [] => false

```

```

| st :: moreStates ⇒ set_eq (getFormulae t st) (formulaStateQui)
    || (checkCopyState t moreStates formulaStateQui)

end.

```

After the formalization of both `isBranchContradictory` and `checkCopyState`, we may implement a top level function in which will take a leaf node of a branch as *nodeContent* and a branch *t* to check whether *t*'s leaf node contradicts any of the formulas of *t*. Therefore, *isBranchContradictory* splits this verification in two cases.

- if *nodeContent* is of form \mathcal{X}_{\Diamond} or \mathcal{X}_{\Box} , then it must check whether the eventuality has been fulfilled (by means of `isBranchContradictory'`) or if it makes the branch ignorable (by means of `checkCopyStates`).
- if *nodeContent* is not of form \mathcal{X}_{\Diamond} or \mathcal{X}_{\Box} , then it searches within the branch if there is a node containing the formula in *nodeContent* with opposite validity.

Definition `isBranchContradictory` (*t*: binTree nat name data)

```

(nodeContent : nat × (((formula name data)) × bool)) :=
match (fst(snd(nodeContent))) with
| chiFormulaBox n phi | chiFormulaBox n phi ⇒
    (isBranchContradictory' t nodeContent)
    || (checkCopyState t (getStates t) (getFormulae t (fst(nodeContent))))
| _ ⇒ isBranchContradictory' t nodeContent

end.

```

| Tableau Rule | Coq Tableau code |
|---|--|
| $\frac{w: \langle t, \pi \rangle \varphi : T}{x: \varphi : T}$ $\frac{w: \langle t, \pi \rangle \varphi : F}{x: \varphi : F}$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) ((leaf ((state), (p , true)))) leafNode), (set_add equiv_dec ((fst(phi)), (state)) statesTree)) else ((addLeftToTableau (origT) ((leaf ((destState), (p , false)))) leafNode), (statesTree)) </pre> |
| $\frac{w: [t, \pi] \varphi : T}{x: \varphi : T}$ $\frac{w: [t, \pi] \varphi : F}{x: \varphi : F}$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) ((leaf ((destState), (p , true)))) leafNode), (statesTree)) else ((addLeftToTableau (origT) ((leaf ((state), (p , false)))) leafNode), (set_add equiv_dec ((fst(phi)), (state)) statesTree)) </pre> |
| $\frac{w: \langle t, \pi^* \rangle \varphi : T}{w: \mathcal{X}_\Diamond : T}$ $\mathcal{X}_\Diamond = \langle t, \pi^* \rangle \varphi$ $\frac{w: \langle t, \pi^* \rangle \varphi : F}{w: \varphi : F}$ $w: \langle t, \pi \rangle \langle t'_\pi, \pi^* \rangle \varphi : F$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) ((leaf ((fst(phi)), ((chiFormulaBox indexchiFormulaBoxmond p) , true)))) leafNode), (statesTree)) else ((addLeftToTableau (origT) ((node ((fst(phi)), (p , false)) (leaf ((fst(phi)), (((diamond t' (sProgram pi')) (diamond t' (star pi') p)))) , false))) (nilLeaf nat name data))) leafNode), (statesTree)) end </pre> |
| $\frac{w: [t, \pi^*] \varphi : T}{w: \varphi : T}$ $w: [t, \pi][t'_\pi, \pi^*] \varphi : T$ $\frac{w: [t, \pi^*] \varphi : F}{w: \mathcal{X}_\Box : F}$ $\mathcal{X}_\Box = [t, \pi^*] \varphi$ | <pre> if (snd(snd(phi))) then ((addLeftToTableau (origT) ((node ((fst(phi)), (p , true)) (leaf ((fst(phi)), (((box t' (sProgram pi'))(box t' (star pi') p))) , true))) (nilLeaf nat name data))) leafNode), (statesTree)) else ((addLeftToTableau (origT) ((leaf ((fst(phi)), ((chiFormulaBox indexchiFormulaBox p) , false)))) leafNode), (statesTree)) end </pre> |

Table 6.2: Summarization of *applyRule* per tableau rule — modal rules.

| Tableau Rule | Coq Tableau code |
|--|---|
| $\frac{w: \mathcal{X}_{\Diamond} : T}{\begin{array}{c} w: \varphi : T \quad w: \varphi : F \\ w: \langle t, \pi \rangle \mathcal{X}_{\Diamond} : T \end{array}}$ | <pre> if (snd(snd(phi))) then match phi' with diamond t' pi p => match pi with star pi' => ((addBranchLeftToTableau (origT) (leaf ((fst(phi)), (p , true))) (node ((fst(phi)), (p , false))) (leaf ((fst(phi)), ((diamond t' pi) (chiFormulaBox indexchiFormulaBoxmond phi') , true))) (nilLeaf nat name data)) leafNode) , (statesTree)) sProgram pi' => (origT , statesTree) end _ => (origT , statesTree) end else (origT , statesTree) </pre> |
| $\frac{w: \mathcal{X}_{\Box} : F}{\begin{array}{c} w: \varphi : F \quad w: \varphi : T \\ w: [t, \pi] \mathcal{X}_{\Box} : F \end{array}}$ | <pre> match phi' with box t' pi p => match pi with star pi' => ((addBranchLeftToTableau (origT) (leaf ((fst(phi)), (p , false))) (node ((fst(phi)), (p , true))) (leaf ((fst(phi)), ((box t' pi) (chiFormulaBox indexchiFormulaBox phi') , false))) (nilLeaf nat name data)) leafNode) , (statesTree)) sProgram pi' => (origT , statesTree) end _ => (origT , statesTree) end else (origT , statesTree) </pre> |

Table 6.3: Summarization of *applyRule* per tableau rule — iteration rules.

Chapter 7

Usage Examples

The current chapter provides usage examples of the formalisms hereby implemented. The proposed framework is further explored, considering the usage of all concepts introduced in Sections 5 and 6.

7.1 Sequencing Entities' communication in *ReLo*

Let us recover the example of the Sequencer model, introduced in Figure 4.2. The *ReLo* program Π is formalized from the following of flow programs in *SequencerProgram* as *pi*, a standard program as defined by *reoProgram*'s constructor *sProgram*. This example's code can be found in the project's repository, in a file named "SequencerEx.v".

Definition *SequencerProgram* := $[flowFifo\ nat\ D\ E; flowSync\ nat\ E\ A; flowFifo\ nat\ E\ F; flowSync\ nat\ F\ B; flowFifo\ nat\ F\ G; flowSync\ nat\ G\ C; flowSync\ nat\ G\ D]$.

Definition *pi* := *sProgram* (*reoProg SequencerProgram []*).

A user-defined model can be input in the system for verification. suppose the following structure denotes a simplification of the Sequencer's behavior as *sequencerModel*, where *sequencerFrame* is a user defined frame as in Definition 6.1.1, and *sequencerValuation* is the model's valuation function as in Definition 6.1.2.

Definition *sequencerModel* := *mkmodel sequencerFrame sequencerValuation*.

Definition *sequencerFrame* := *mkframe* [*DA;DB;DC;DD;DE;DF;DG;D_DFIFOE;D_EFIFO;D_FFIFOG*] [(*DD,D_DFIFOE*);(*D_DFIFOE,DE*);(*DE,DA*);(*DE,D_EFIFO*);(*D_EFIFO,DF*);(*DF,DB*);(*DF,D_FFIFOG*);(*D_FFIFOG,DG*);(*DG,DC*);(*DG,DD*)] *sequencerLambda deltaSequencer*.

The valuation function *sequencerValuation* denotes which formulae are valid on each state of the model *sequencerModel* using *getPropositionSequencer* as the function which stores the set of formulae valid for each state. An empty set of valid formulae denotes

that there is no formulae valid in this state.

Definition *sequencerValuation* ($s : \text{statesSequencer}$) ($p : (\text{dataProp ports nat})$) :=
 $\text{existsb } (\text{fun } x : (\text{dataProp ports nat}) \Rightarrow \text{equiv_decb } p \ x)$
 $(\text{getPropositionSequencer } s).$

Definition *getPropositionSequencer* ($s : \text{statesSequencer}$) :=
 $\text{match } s \text{ with}$
 $| DA \Rightarrow [\text{dataInPorts } A \ 0; \text{dataInPorts } A \ 1]$
 $| DB \Rightarrow [\text{dataInPorts } B \ 0; \text{dataInPorts } B \ 1]$
 $| DC \Rightarrow [\text{dataInPorts } C \ 0; \text{dataInPorts } C \ 1]$
 $| DD \Rightarrow [\text{dataInPorts } D \ 0; \text{dataInPorts } D \ 1]$
 $| DE \Rightarrow []$
 $| DF \Rightarrow []$
 $| DG \Rightarrow []$
 $| D_EFIFO \Rightarrow []$
 $| D_FFIFO \Rightarrow []$
 $| D_DFIFO \Rightarrow []$
 end.

Properties may be formalized as *ReLo* formulae in Coq and validated on the model. The following formula states that “after every execution of *pi* with *t* denoting a data flow of item 1 in port name *D*, no data flow is present on ports *A*, *B* and *C*. The below evaluation of *singleFormulaVerify* yields **true** because the model [sequencerModel] indeed satisfies the property modelled.

Definition $t := [\text{dataPorts } D \ 1].$
Eval $\text{compute in } \text{singleFormulaVerify sequencerModel}$
 $(\text{box } t \ \text{pi } (((\text{neg } ((\text{and } (\text{and } (\text{proposition } (\text{dataInPorts } A \ 1))$
 $(\text{proposition } (\text{dataInPorts } B \ 1))))$
 $(\text{proposition } (\text{dataInPorts } C \ 1))))))) \ t.$

Another example of formulae can be stated as follows: “given that there is a nondeterministic execution where there is a data item 1 in port *D*, then there is a nondeterministic execution where the same data item will be in port *C*.”

Eval $\text{compute in } \text{singleFormulaVerify sequencerModel}$
 $(\text{box } t \ \text{pi } (((\text{neg } ((\text{and } (\text{and } (\text{proposition } (\text{dataInPorts } A \ 1))$
 $(\text{proposition } (\text{dataInPorts } B \ 1))))$
 $(\text{proposition } (\text{dataInPorts } C \ 1))))))) \ t.$

The formalization presented in Section 6.3 is also useful for scenarios where the user

only wants to reason over a Reo model and a *ReLo* formulae, leaving the task to construct a model for the Coq implementation as in variable *example1ReLo*. It uses *constructModel* with the Reo model's port names, the data flow denoted by *t*, and a formula that states “given that there is a nondeterministic execution where there is a data item 1 in port *C*, then there is a nondeterministic execution where the same data item will be in port *B*”. Variable *example1ReLo* then will return a *ReLo* model in which the formula is valid. Then, the resulting model in *example1ReLo* can be used by the model verification apparatus implemented in Section 6.2.

Definition *example1ReLo* :=

```
Eval compute in constructModel [A ; B ; C ; D ; E ; F ; G] [t]
(imp (box t piStar'((((((((proposition (dataInPorts C 1))))))))))
  (box t piStar'((((((((proposition (dataInPorts B 1))))))))))
  (length(SequencerProgram) × 3).
```

Definition *validityExample1ReLo* := *singleFormulaVerify example1ReLo*

```
(imp (box t piStar'((((((((proposition (dataInPorts C 1))))))))))
  (box t piStar'((((((((proposition (dataInPorts B 1)))))))))) t).
```

7.2 Modelling Smart Cities entities interaction in *ReLo*

We recover the Reo model in Figure 4.3 from [36] to instantiate a Smart Crossroad scenario, in which the traffic lights interact in a Reo model to obtain the token which will enable them to allow cars pass. The Coq code of this example may be found in the project's repository in a file named “SmartCitiesEx.v”.

We proceed by stating the set of port names composing the model as *modelPortsType*.

Inductive *modelPortsType* := *A* | *Y* | *B* | *X* | *I* | *J* | *L* | *K* | *M* | *N* | *C*.

The corresponding Reo model is formalized as the following *ReLo* program. The model contains a Transform channel which function is to invert its data flow. The transform function is *swap01* as follows.

Definition *SmartCitiesModelProg* := [*flowMerger nat A B Y*; *flowSync nat Y X*; *flowSync nat X I*; *flowSync nat X J*; *flowTransform swap01 J L*; *flowSync nat I K*; *flowSync nat L M*; *flowFifo nat K N*; *flowMerger nat M N C*].

Definition *swap01* (*n:nat*) : *nat* :=

```
match n with
| 0 ⇒ 1
| 1 ⇒ 0
| Datatypes.S o ⇒ (Datatypes.S o)
```

end.

Let us suppose the starting data flow is item 1 in port name A . In this example, we will consider the iteration of program *SmartCitiesModelProg* as *piStar*.

Definition $t := [dataPorts\ A\ 1]$.

Definition $piStar := star\ (reoProg\ SmartCitiesModelProg\ [])$.

Then, the behaviour of the model can be analyzed as follows. The idea of the connector is that a single data input 1 will result in both data items 0 and 1 reaching port name C , denoting that both semaphores get to let their cars pass. This can be formalized as $(D_A = 1) \rightarrow (\langle t, \pi^* \rangle D_A = C \wedge \langle t, \pi^* \rangle D_C = 1)$, which in Coq can be formalized as follows.

$(imp\ (((prop\ (dataInPorts\ A\ 1))))$
 $(and\ (box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 0))))$
 $(box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 1))))))$

Therefore, the model calculated based in the formula above can be calculated by *constructModel* resulting in *SmartCitiesEx1*.

Definition $SmartCitiesEx1 :=$

$constructModel\ [A\ ;\ Y\ ;\ B\ ;\ X\ ;\ I\ ;\ J\ ;\ L\ ;\ K\ ;\ M\ ;\ N\ ;\ C]\ [t]$
 $(imp\ (((prop\ (dataInPorts\ A\ 1))))$
 $(and\ (box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 0))))$
 $(box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 1))))))$
 $(length(SmartCitiesModelProg) \times 3)$.

The validity of the formula in *SmartCitiesEx1* can then be assessed by *singleFormulaVerify*.

Eval **compute** in *singleFormulaVerify SmartCitiesEx1*

$(imp\ (((prop\ (dataInPorts\ A\ 1))))$
 $(and\ (box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 0))))$
 $(box\ t\ piStar\ (((prop\ (dataInPorts\ C\ 1))))))\ t$.

7.3 Byzantine Consensus

In this example, we show a *ReLo* formalization of the Byzantine consensus detailed in Chapter 4. We detail the implementation of the Reo model in Figure 4.4. The code of this example can be found in the project's repository in a file named "ByzantineConsensus.v".

We start by formalizing the Reo model in Figure 4.4 as $\pi = (f, b)$, where $f = BizantineConsensusFlowProgram$ and $b = BizantineConsensusBlockProgram$, which will be combined in *pi* to form the pair (f, b) of a *ReLo* program. Definition *piStar* is the iteration of π as π^* .

Definition *BizantineConsensusFlowProgram* := $[flowFilter\ timerProp\ R'\ B'; flowSync\ nat\ B'\ A;$

$flowSync\ nat\ B'\ X; flowFilter\ timerProp\ X\ CR'; flowFilter\ timerProp\ A\ VR';$

$flowFilter\ timerProp\ B\ VB'; flowFilter\ timerBPPProp\ A'\ B;$

$flowFilter\ timerAPProp\ A'\ P; flowFilter\ timerProp\ P\ R; flowSync\ nat\ R\ Y;$

$flowSync\ nat\ R\ Z; flowFilter\ timerVAProp\ Y\ VA'; flowFilter\ timerProp\ Z\ CA']$.

Definition *BizantineConsensusBlockProgram* := $[flowSyncdrain\ A\ X; flowaSyncdrain\ VR'\ CR'; flowaSyncdrain\ VB'\ A; flowaSyncdrain\ VA'\ CA'; flowSyncdrain\ Y\ Z]$.

Definition *pi* := *sProgram* (*reoProg* *BizantineConsensusFlowProgram* *BizantineConsensusBlockProgram*).

Definition *piStar* := *star* (*reoProg* *BizantineConsensusFlowProgram* *BizantineConsensusBlockProgram*).

Each of the Filter connectors above contains a predicate which specifies a condition for the data to flow for the specific port. In this example, we denote that the data to flow from nodes CA', CR, and A' is 1. If it is zero, then it is the case it flows from A to B only. As stated above, we model the notion of a temporizer as a special predicate which knows when to fire the transition. For the sake of simplicity, it assumes that the timer is always set to immediately fire, but the data will flow to sink nodes VB', VR' and VA' only if it equals 0.

Definition *timerProp* (*n* : *nat*) := *true*.

Definition *timerAPProp* (*n* : *nat*) :=

match *n* *with*

| 1 \Rightarrow *true*

| 0 \Rightarrow *false*

| *Datatypes.S* *m* \Rightarrow *false*

end.

Definition *timerBPPProp* (*n*:*nat*) :=

match *n* *with*

| 0 \Rightarrow *true*

| *Datatypes.S* *m* \Rightarrow *false*

end.

Definition *timerVAPProp* (*n*:*nat*) :=

match *n* *with*

| 0 \Rightarrow *true*

| *Datatypes.S* *m* \Rightarrow *false*

end.

One of the properties that can be checked in this model is as follows. The replica will only set its state to commit (by sending data to node "CA'") if the conditions required are met, and the preset timer for the V's have not been fired. This can be modelled as the following *ReLo* formulae: $[t, \pi^*](D_{CA'} = 1) \rightarrow [t, \pi^*](\neg(D_{VA'}) \wedge \neg(D_{VB'}) \wedge \neg(D_{VR'}))$ as the *ReLo* used to find the model yielded by *property1*. Definition *t* states that the initial data flow in this example is the data item 1 in port name *A*.

Definition *t* := $[dataPorts\ A'\ 1]$.

Definition *property1* := **Eval** compute in *constructModel*

$[R'; B'; A; X; CR'; VR'; B; VB'; A'; P; R; Y; Z; VA'; CA'] [t]$
 $((imp\ (box\ t\ piStar(prop\ (dataInPorts\ CA'\ 1))))$
 $\quad (box\ t\ piStar(and\ (and\ (neg(prop\ (dataInPorts\ VA'\ 1)))$
 $\quad\quad (neg(prop\ (dataInPorts\ VB'\ 1))))$
 $\quad\quad (neg(prop\ (dataInPorts\ VR'\ 1))))))$
 $(length(BizantineConsensusFlowProgram) + length(BizantineConsensusBlockProgram)).$

The formula above can be verified in the model obtained by *property1* by means of *singleFormulaVerify* as follows.

Eval compute in *singleFormulaVerify* *property1*

$((imp\ (box\ t\ piStar(prop\ (dataInPorts\ CA'\ 1))))$
 $\quad (box\ t\ piStar(and\ (and\ (neg(prop\ (dataInPorts\ VA'\ 1)))$
 $\quad\quad (neg(prop\ (dataInPorts\ VB'\ 1))))\ (neg(prop\ (dataInPorts\ VR'\ 1))))))$
t.

Another property that can be verified is as follows. Conversely to the data flow setting the replica to the commit state in nodes with port names CA', data will flow to the node denoting the view change only if there is no data in the change nodes CA' and CR'. This property along with *property1* states the notion that "no replica can set its state to commit and change view simultaneously".

Definition *property2* := **Eval** compute in *constructModel*

$[R'; B'; A; X; CR'; VR'; B; VB'; A'; P; R; Y; Z; VA'; CA'] [t']$
 $((imp\ ((box\ t'\ piStar(prop\ (dataInPorts\ VA'\ 0))))$
 $\quad (box\ t'\ piStar((and\ (neg(prop\ (dataInPorts\ CA'\ 0)))$
 $\quad\quad (neg(prop\ (dataInPorts\ CR'\ 0))))))$
 $(length(BizantineConsensusFlowProgram) + length(BizantineConsensusBlockProgram)).$

It can also be verified with *singleFormulaVerify* with the model obtained by *property2* as follows.

Eval compute in *singleFormulaVerify* property2

```
((imp ((box t' piStar(proposition (dataInPorts VA' 0))))
  (box t' piStar((and (neg(proposition (dataInPorts CA' 0)))
    (neg(proposition (dataInPorts CR' 0)))))))) t.
```

7.4 *ReLo* Tableau proofs

In this section, we will recover Axiom \mathcal{K} 's proof in Section 5.4.1 to show how tableau proofs can be addressed in Coq. Its corresponding Coq code is in the project's repository, in a file named "TableauEx.v". The axiom can be formalized in Coq as *axiomKTableau*. In this example, we denote φ and ψ respectively as Coq *ReLo* formulas (*proposition (dataInPorts A 1)*) and (*proposition (dataInPorts B 1)*). After each Coq tableau rule application, the resulting proof tree is also shown as Listings 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7.

Definition *axiomKTableau* :=

```
(imp ((box t (sProgram (reoProg [flowLossySync nat A B] []))
  (imp (proposition (dataInPorts A 1))
    (proposition (dataInPorts B 1)))))
  (imp ((box t (sProgram (reoProg [flowLossySync nat A B] []))
    (proposition (dataInPorts A 1)))
    ((box t (sProgram (reoProg [flowLossySync nat A B] []))
      (proposition (dataInPorts B 1)))).
```

Listing 7.1: Tableau obtained from formula *axiomKTableau*

```
1 = { | proofTree :=
2   leaf (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
3     (imp (proposition (dataInPorts A 1))
4       (proposition (dataInPorts B 1)))))
5     (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
6       (proposition (dataInPorts A 1)))
7       (box t (sProgram (reoProg [flowLossySync nat A B] []))
8         (proposition (dataInPorts B 1))), false));
9   statesTree := []
10 }
```

Then, the analysis of the tableau can be done by applying rules using *applyRules* successively. It starts with *formula2TableauKApp1* as the result of applying rule $(\rightarrow -F)$

to the initial proof tree in *axiomKTableau* as the smallest tableau containing \mathcal{K} as false. To proceed, it is necessary to supply the node which the derivation rule will be applied, the corresponding indexes introduced in *applyRule*'s definition, and the leaf node that identifies the branch to which the result of the rule application must be added to the proof tree.

Definition *formula2TableauKApp1* :=
applyRule (*formula_eqDec* *Hfil* *portsEq* *nat_eqDec*) *formula2TableauK*
 (0, (*imp* (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*imp* (*proposition* (*dataInPorts* *A* 1))
 (*proposition* (*dataInPorts* *B* 1)))))
 (*imp*
 (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*proposition* (*dataInPorts* *A* 1)))
 (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*proposition* (*dataInPorts* *B* 1))))), *false*)) 0 0 0 0
 (0, (*imp* (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*imp* (*proposition* (*dataInPorts* *A* 1))
 (*proposition* (*dataInPorts* *B* 1)))))
 (*imp*
 (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*proposition* (*dataInPorts* *A* 1)))
 (*box* *t* (*sProgram* (*reoProg* [*flowLossySync* *nat* *A* *B*] []))
 (*proposition* (*dataInPorts* *B* 1))))), *false*)).

Listing 7.2: Tableau obtained from formula *formula2TableauKApp1*

```

1 = {| proofTree :=
2 node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
3   (imp (proposition (dataInPorts A 1))
4     (proposition (dataInPorts B 1)))))
5   (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
6     (proposition (dataInPorts A 1))
7     (box [] (sProgram (reoProg [flowLossySync nat A B] []))
8       (proposition (dataInPorts B 1))))) , false))
9 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10   (imp (proposition (dataInPorts A 1))

```

```

11  (proposition (dataInPorts B 1))), true))
12  (leaf (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
13    (proposition (dataInPorts A 1)))
14    (box t (sProgram (reoProg [flowLossySync nat A B] []))
15    (proposition (dataInPorts B 1))), false)))
16  (nilLeaf nat ports nat)) (nilLeaf nat ports nat);
17  statesTree := []
18  |}

```

The derivation proceeds with *formula2TableauKApp2*, which applies the rule $(\rightarrow -T)$ in the right hand of \mathcal{K} 's main implication obtained in *formula2TableauKApp1*.

Definition *formula2TableauKApp2* :=

```

applyRule (formula_eqDec Hfil portsEq nat_eqDec) formula2TableauKApp1
(0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
  (proposition (dataInPorts A 1)))
  (box t (sProgram (reoProg [flowLossySync nat A B] []))
  (proposition (dataInPorts B 1))), false)) 0 0 0 0
(0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
  (proposition (dataInPorts A 1)))
  (box t (sProgram (reoProg [flowLossySync nat A B] []))
  (proposition (dataInPorts B 1))), false)).

```

Listing 7.3: Tableau obtained from formula *formula2TableauKApp2*

```

1  = {| proofTree :=
2  node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
3    (imp (proposition (dataInPorts A 1))
4    (proposition (dataInPorts B 1))))
5    (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
6    (proposition (dataInPorts A 1)))
7    (box t (sProgram (reoProg [flowLossySync nat A B] []))
8    (proposition (dataInPorts B 1))), false))
9  (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10    (imp (proposition (dataInPorts A 1))
11    (proposition (dataInPorts B 1))), true))
12  (node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
13    (proposition (dataInPorts A 1)))

```

```

14   (box t (sProgram (reoProg [flowLossySync nat A B] []))
15   (proposition (dataInPorts B 1))), false))
16 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
17   (proposition (dataInPorts A 1))), true))
18 (leaf (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
19   (proposition (dataInPorts B 1))), false)))
20 (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
21 (nilLeaf nat ports nat)) (nilLeaf nat ports nat);
22 statesTree := []
23 |}

```

Then, the next rule to be applied is $([]-F)$ which was obtained in *formula2TableauKApp2*. Note that, the next index has been updated to consider a new state, denoted by the number 1.

Definition *formula2TableauKApp3* :=
applyRule (formula_eqDec Hfil portsEq nat_eqDec) formula2TableauKApp2
 (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
 (proposition (dataInPorts B 1))), false)) 1 0 0 0
 (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
 (proposition (dataInPorts B 1))), false)).

Listing 7.4: Tableau obtained from formula *formula2TableauKApp3*

```

1  { | proofTree :=
2  node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
3    (imp (proposition (dataInPorts A 1))
4      (proposition (dataInPorts B 1)))))
5    (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
6      (proposition (dataInPorts A 1))
7        (box t (sProgram (reoProg [flowLossySync nat A B] []))
8          (proposition (dataInPorts B 1))))) , false))
9  (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10    (imp (proposition (dataInPorts A 1))
11      (proposition (dataInPorts B 1))))) , true))
12  (node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
13    (proposition (dataInPorts A 1))
14    (box t (sProgram (reoProg [flowLossySync nat A B] []))

```

```

15  (proposition (dataInPorts B 1))), false))
16  (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
17    (proposition (dataInPorts A 1))), true))
18  (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
19    (proposition (dataInPorts B 1))), false))
20  (leaf (1, (proposition (dataInPorts B 1), false)))
21  (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
22  (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
23  (nilLeaf nat ports nat);
24  statesTree := [(0, 1)]
25  |}

```

The derivation obtained in *formula2TableauKApp4* is the application of rule $(\rightarrow -T)$ on the formula obtained from *formula2TableauKApp1* as the implication of the left hand side of \mathcal{K} 's main implication.

Definition *formula2TableauKApp4* :=
applyRule (*formula_eqDec* *Hfil* *portsEq* *nat_eqDec*) *formula2TableauKApp3*
 (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
 (imp (proposition (dataInPorts A 1))
 (proposition (dataInPorts B 1))), true)) 1 1 0 0
 (1, (proposition (dataInPorts B 1), false)).

Listing 7.5: Tableau obtained from formula *formula2TableauKApp4*

```

1  = { | proofTree :=
2  node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
3    (imp (proposition (dataInPorts A 1))
4      (proposition (dataInPorts B 1)))))
5    (imp(box t (sProgram (reoProg [flowLossySync nat A B] []))
6      (proposition (dataInPorts A 1)))
7      (box t (sProgram (reoProg [flowLossySync nat A B] []))
8        (proposition (dataInPorts B 1))))) , false))
9  (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10    (imp (proposition (dataInPorts A 1))
11      (proposition (dataInPorts B 1))), true))
12    (node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
13      (proposition (dataInPorts A 1))

```

```

14   (box t (sProgram (reoProg [flowLossySync nat A B] []))
15   (proposition (dataInPorts B 1))), false))
16 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
17   (proposition (dataInPorts A 1))), true))
18 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
19   (proposition (dataInPorts B 1))), false))
20 (node (1, (proposition (dataInPorts B 1), false))
21 (leaf (1, (imp (proposition (dataInPorts A 1))
22   (proposition (dataInPorts B 1))), true)))
23   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
24   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
25   (nilLeaf nat ports nat)) (nilLeaf nat ports nat);
26 statesTree := [(0, 1)]
27 |}

```

The next derivation is *formula2TableauKApp5* is bound to apply rule $\rightarrow -T$ to the implication obtained in *formula2TableauKApp4*.

Definition *formula2TableauKApp5* :=
applyRule (*formula_eqDec Hfil portsEq nat_eqDec*) *formula2TableauKApp4*
 (1, (*imp* (*proposition* (*dataInPorts A* 1))
 (*proposition* (*dataInPorts B* 1)), *true*)) 1 1 0 0
 (1, (*imp* (*proposition* (*dataInPorts A* 1))
 (*proposition* (*dataInPorts B* 1)), *true*)).

Listing 7.6: Tableau obtained from formula *formula2TableauKApp5*

```

1 = {| proofTree :=
2   node(0, (imp(box t (sProgram (reoProg [flowLossySync nat A B] []))
3     (imp (proposition (dataInPorts A 1))
4       (proposition (dataInPorts B 1)))))
5     (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
6       (proposition (dataInPorts A 1))
7       (box t (sProgram (reoProg [flowLossySync nat A B] []))
8         (proposition (dataInPorts B 1))))) , false))
9   (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10     (imp (proposition (dataInPorts A 1))
11       (proposition (dataInPorts B 1))), true))
12   (node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))

```



```

13   (proposition (dataInPorts A 1)))
14   (box t (sProgram (reoProg [flowLossySync nat A B] [])))
15   (proposition (dataInPorts B 1))), false))
16 (node (0,(box t (sProgram (reoProg [flowLossySync nat A B] [])))
17   (proposition (dataInPorts A 1)), true))
18 (node (0,(box t (sProgram (reoProg [flowLossySync nat A B] [])))
19   (proposition (dataInPorts B 1))), false))
20 (node (1, (proposition (dataInPorts B 1), false))
21 (node (1,(imp (proposition (dataInPorts A 1))
22   (proposition (dataInPorts B 1)), true))
23   (leaf (1, (proposition (dataInPorts A 1), false)))
24   (leaf (1, (proposition (dataInPorts B 1), true))))
25   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
26   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
27   (nilLeaf nat ports nat)) (nilLeaf nat ports nat);
28 statesTree := [(0, 1)]
29 |}

```

The last derivation applied is *formula2TableauKApp6*, which will apply ($\Box - T$) in the left branch generated by *formula2TableauKApp5*. The resulting tableau will result in a contradiction in both branches obtained in *formula2TableauKApp5* as the right branch already have a contradiction.

Definition *formula2TableauKApp6* :=
applyRule (*formula_eqDec Hfil portsEq nat_eqDec*)
formula2TableauKApp5
 (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
 (*proposition* (*dataInPorts A* 1)), *true*)) 1 1 0 0
 (1, (*proposition* (*dataInPorts A* 1), *false*)).

Listing 7.7: Tableau obtained from formula *formula2TableauKApp6*

```

1 = {| proofTree :=
2   node (0,(imp(box t (sProgram (reoProg [flowLossySync nat A B] [])))
3     (imp (proposition (dataInPorts A 1))
4       (proposition (dataInPorts B 1)))))
5     (imp (box t (sProgram (reoProg [flowLossySync nat A B] [])))
6       (proposition (dataInPorts A 1)))
7     (box t (sProgram (reoProg [flowLossySync nat A B] [])))

```

```

8   (proposition (dataInPorts B 1))), false))
9 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
10   (imp (proposition (dataInPorts A 1))
11   (proposition (dataInPorts B 1))), true))
12 (node (0, (imp (box t (sProgram (reoProg [flowLossySync nat A B] []))
13   (proposition (dataInPorts A 1))
14   (box t (sProgram (reoProg [flowLossySync nat A B] []))
15   (proposition (dataInPorts B 1))), false))
16 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
17   (proposition (dataInPorts A 1)), true))
18 (node (0, (box t (sProgram (reoProg [flowLossySync nat A B] []))
19   (proposition (dataInPorts B 1)), false))
20 (node (1, (proposition (dataInPorts B 1), false))
21 (node (1, (imp (proposition (dataInPorts A 1))
22   (proposition (dataInPorts B 1)), true))
23   (node (1, (proposition (dataInPorts A 1), false))
24     (leaf (1, (proposition (dataInPorts A 1), true)))
25     (nilLeaf nat ports nat))
26   (leaf (1, (proposition (dataInPorts B 1), true))))
27   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
28   (nilLeaf nat ports nat)) (nilLeaf nat ports nat))
29   (nilLeaf nat ports nat)) (nilLeaf nat ports nat);
30 statesTree := [(0, 1)]
31 |}

```

The user can check whether the obtained tableau after the rule application is a closed tableau by means of *checkContradictoryBranch*. It will calculate each branch in proof tree *formula2TableauKApp6* and return **true** if all branches are closed, or **false** otherwise.

```

Eval compute in isTableauClosed (formula_eqDec Hfil portsEq nat_eqDec)
(proofTree(formula2TableauKApp6)).

```

With the model obtained in Listing 7.7, *isTableauClosed* will compute two branches: one ending in row 24 and other ending in row 26. The branch ending in row 24 has a contradiction with the node in row 23 regarding formula (proposition (dataInPorts A 1)). The branch ending in row 26 has a contradiction with the node in row 20 regarding formulae (proposition (dataInPorts B 1)). As all branches have a contradiction, the tableau is closed.

Chapter 8

Conclusions and Further Work

Reo is a widely used tool to model new systems out of the coordination of already existing pieces of software. It has been used in a variety of domains, drawing the attention of researchers from different locations around the world. This has resulted in Reo having many formal semantics proposed, each one employing different formalisms: operational, co-algebraic, and coloring semantics are some of the types of semantics proposed for Reo.

This work presents *ReLo*, a dynamic logic to reason about Reo models. We have discussed its core definitions, syntax, semantic notion, providing soundness and completeness proofs for it. We also discussed a Coq implementation of *ReLo* and some useful tools to enable its usage in a computational environment, like the search for a model that satisfies a formula, and a *ReLo* tableau implementation. *ReLo* naturally subsumes the notion of Reo programs and models in its syntax and semantics, and implementing its core concepts in Coq enables the usage of Coq’s proof apparatus to reason over Reo models with *ReLo*.

Future work may consider the integration of the current implementation of *ReLo* with ReoXplore¹, a web platform conceived to reason about Reo models, the study of *ReLo* and other Reo formal semantics (i.e., Constraint Automata and its core variants) to evaluate the expressiveness of possible extensions of *ReLo*, and the adaptation of *ReLo* to subsume other Reo channels created for other formal semantics to increase its expressibility, enabling the modelling of properties like actions and timing constraints performed by some other formal semantics [20,49]. Specifically regarding *ReLo* complexity results and the completeness of *ReLo*’s tableau are also proposed as future work.

¹<https://github.com/frame-lab/ReoXplore>

References

- [1] ABRIAL, J. B-tool reference manual. b-core (uk) ltd, 1991.
- [2] AHO, A. V., ULLMAN, J. D. *Foundations of computer science*. Computer Science Press, Inc., 1992.
- [3] ARBAB, F. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366.
- [4] ARBAB, F. Coordination for component composition. *Electronic Notes in Theoretical Computer Science* 160 (2006), 15 – 40. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).
- [5] ARBAB, F., BAIER, C., DE BOER, F., RUTTEN, J. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling* 6, 1 (2007), 59–82.
- [6] ARBAB, F., KOKASH, N., MENG, S. Towards using reo for compliance-aware business process modeling. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2008), Springer, p. 108–123.
- [7] ARBAB, F., RUTTEN, J. J. A coinductive calculus of component connectors. In *International Workshop on Algebraic Development Techniques* (2002), Springer, p. 34–55.
- [8] ARDESHIR-LARIJANI, E., FARHADI, A., ARBAB, F. Simulation of hybrid reo connectors. In *2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)* (2020), IEEE, p. 1–10.
- [9] ATKINSON, C., KUHNE, T. Model-driven development: a metamodeling foundation. *IEEE software* 20, 5 (2003), 36–41.
- [10] BAIER, C. Probabilistic models for reo connector circuits. *J. UCS* 11, 10 (2005), 1718–1748.
- [11] BAIER, C., SIRJANI, M., ARBAB, F., RUTTEN, J. Modeling component connectors in reo by constraint automata. *Science of computer programming* 61, 2 (2006), 75–113.
- [12] BARTZIA, E.-I., STRUB, P.-Y. A formal library for elliptic curves in the coq proof assistant. In *International Conference on Interactive Theorem Proving* (2014), Springer, p. 77–92.
- [13] BELLA, G., PAULSON, L. C., MASSACCI, F. The verification of an industrial payment protocol: The set purchase phase. In *Proceedings of the 9th ACM conference on Computer and communications security* (2002), ACM, p. 12–20.

- [14] BENEVIDES, M., LOPES, B., HAEUSLER, E. H. Towards reasoning about petri nets: A propositional dynamic logic based approach. *Theoretical Computer Science* 744 (2018), 22–36.
- [15] BENZMÜLLER, C., PALEO, B. W. Interacting with modal logics in the coq proof assistant. In *International Computer Science Symposium in Russia* (2015), Springer, p. 398–411.
- [16] BLACKBURN, P., DE RIJKE, M., VENEMA, Y. Cambridge tracts in theoretical computer science, 2001.
- [17] BRUNI, R., MONTANARI, U. Zero-safe nets: Comparing the collective and individual token approaches. *Information and computation* 156, 1-2 (2000), 46–89.
- [18] CARBONNEAUX, Q., HOFFMANN, J., REPS, T., SHAO, Z. Automated resource analysis with coq proof objects. In *International Conference on Computer Aided Verification* (2017), Springer, p. 64–85.
- [19] CHELLAS, B. F. *Modal logic: an introduction*. Cambridge university press, 1980.
- [20] CHEN, X., SUN, J., SUN, M. A hybrid model of connectors in cyber-physical systems. In *International Conference on Formal Engineering Methods* (2014), Springer, p. 59–74.
- [21] CHLIPALA, A. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.
- [22] CLARKE, D. Coordination: Reo, nets, and logic. In *International Symposium on Formal Methods for Components and Objects* (2007), Springer, p. 226–256.
- [23] COE, T., MATHISEN, T., MOLER, C., PRATT, V. Computational aspects of the pentium affair. *IEEE Computational Science and Engineering* 2, 1 (1995), 18–30.
- [24] COQUAND, T. *Une théorie des constructions*. Tese de Doutorado, Paris 7, 1985.
- [25] COQUAND, T. *An analysis of Girard’s paradox*. Tese de Doutorado, INRIA, 1986.
- [26] COQUAND, T., PAULIN, C. Inductively defined types. In *COLOG-88* (1990), Springer, p. 50–66.
- [27] DE GIACOMO, G., MASSACCI, F. Tableaux and algorithms for propositional dynamic logic with converse. In *International Conference on Automated Deduction* (1996), Springer, p. 613–627.
- [28] DE GIACOMO, G., MASSACCI, F. Combining deduction and model checking into tableaux and algorithms for converse-PDL. *Information and Computation* 162, 1-2 (2000), 117–137.
- [29] DELAHAYE, D. A tactic language for the system coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2000), Springer, p. 85–95.

- [30] DOWEK, G., FELTY, A., HERBELIN, H., HUET, G., MURTHY, C., PARENT, C., PAULIN-MOHRING, C., WERNER, B. The coq proof assistant: Version 8.13, 2020. User manual in <https://coq.inria.fr/distrib/current/refman/>.
- [31] FITTING, M. *Proof methods for modal and intuitionistic logics*, vol. 169. Springer Science & Business Media, 1983.
- [32] FITTING, M. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.
- [33] GARSON, J. Modal Logic. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., summer 2021 ed. Metaphysics Research Lab, Stanford University, 2021.
- [34] GEUVERS, H. Proof assistants: History, ideas and future. *Sadhana* 34, 1 (2009), 3–25.
- [35] GONTHIER, G. The four colour theorem: Engineering of a formal proof. In *Computer mathematics*. Springer, 2008, p. 333–333.
- [36] GRILO, E., LOPES, B. Modelling and certifying smart cities in reo circuits. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)* (2020), IEEE, p. 453–458.
- [37] GRILO, E., LOPES, B. Relo: a dynamic logic to reason about reo circuits1. In *Pre-Proceedings of the 15th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA)* (2020), p. 32.
- [38] HAREL, D., KOZEN, D., TIURYN, J. *Other Variants of PDL*. MIT Press, 2000.
- [39] HAREL, D., KOZEN, D., TIURYN, J. Dynamic logic. In *Handbook of philosophical logic*. Springer, 2001, p. 99–217.
- [40] HOARE, C. An overview of some formal methods for program design. *Computer*, 9 (1987), 85–91.
- [41] HONGFEI, D., ZHANG, E. Neo: A distributed network for the smart economy. Neo Foundation, 2015.
- [42] JACKSON, D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (2002), 256–290.
- [43] JONGMANS, S.-S. T., ARBAB, F. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science* 22, 1 (2012).
- [44] KLEIN, J., KLÜPPELHOLZ, S., STAM, A., BAIER, C. Hierarchical modeling and formal verification. an industrial case study using reo and vereofy. In *International Workshop on Formal Methods for Industrial Critical Systems* (2011), Springer, p. 228–243.
- [45] KNIGHT, J. C. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering* (2002), ACM, p. 547–550.

- [46] KNIGHT, J. C., DEJONG, C. L., GIBBLE, M. S., NAKANO, L. G. Why are formal methods not used more widely? In *Fourth NASA formal methods workshop* (1997), Citeseer.
- [47] KOKASH, N., ARBAB, F. Formal design and verification of long-running transactions with extensible coordination tools. *IEEE Transactions on Services Computing* 6, 2 (2011), 186–200.
- [48] KOKASH, N., CHANGIZI, B., ARBAB, F. A semantic model for service composition with coordination time delays. In *International Conference on Formal Engineering Methods* (2010), Springer, p. 106–121.
- [49] KOKASH, N., KRAUSE, C., DE VINK, E. Reo+ mcrl2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing* 24, 2 (2012), 187–216.
- [50] KOKASH, N., KRAUSE, C., DE VINK, E. P. Data-aware design and verification of service compositions with reo and mcrl2. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010), p. 2406–2413.
- [51] KRIPKE, S. A. A completeness theorem in modal logic. *The journal of symbolic logic* 24, 1 (1959), 1–14.
- [52] LAMPORT, L., SHOSTAK, R., PEASE, M. The byzantine generals problem. In *ACM Transactions on Programming Languages and System*. 1982, p. 382–401.
- [53] LEROY, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices* (2006), vol. 41, ACM, p. 42–54.
- [54] LETOUZEY, P. A new extraction for coq. In *International Workshop on Types for Proofs and Programs* (2002), Springer, p. 200–219.
- [55] LI, Y., SUN, M. Modeling and verification of component connectors in coq. *Science of Computer Programming* 113 (2015), 285–301.
- [56] LI, Y., ZHANG, X., JI, Y., SUN, M. Capturing stochastic and real-time behavior in reo connectors. In *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings* (2017), p. 287–304.
- [57] LI, Y., ZHANG, X., JI, Y., SUN, M. A formal framework capturing real-time and stochastic behavior in connectors. *Science of Computer Programming* (2019).
- [58] LOVELAND, D. W. *Automated Theorem Proving: a logical basis*. Elsevier, 2014.
- [59] MARTIN-LÖF, P., SAMBIN, G. *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.
- [60] MASSACCI, F. Strongly analytic tableaux for normal modal logics. In *International Conference on Automated Deduction* (1994), Springer, p. 723–737.
- [61] MOUSAVI, M. R., SIRJANI, M., ARBAB, F. Formal semantics and analysis of component connectors in reo. *Electronic Notes in Theoretical Computer Science* 154, 1 (2006), 83–99.

- [62] NAVIDPOUR, S., IZADI, M. Linear temporal logic of constraint automata. In *Advances in Computer Science and Engineering*. Springer, 2008, p. 972–975.
- [63] NAWAZ, M. S., SUN, M. Reo2pvs: Formal specification and verification of component connectors. In *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018*. (2018), p. 391–390.
- [64] NIPKOW, T., PAULSON, L. C., WENZEL, M. *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.
- [65] OSTRO, J. S. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software* 18, 1 (1992), 33–60.
- [66] OWRE, S., RUSHBY, J., SHANKAR, N., OTHERS. PVS specification and verification system. *URL: pvs.csl.sri.com* (2001).
- [67] PAPAZOGLU, M. P. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on* (2003), IEEE, p. 3–12.
- [68] PAPAZOGLU, M. P., VAN DEN HEUVEL, W.-J. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal* 16, 3 (2007), 389–415.
- [69] PAULIN-MOHRING, C. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications* (1993), Springer, p. 328–345.
- [70] PAULIN-MOHRING, C. Introduction to the calculus of inductive constructions, 2015.
- [71] PFENNING, F., PAULIN-MOHRING, C. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics* (1989), Springer, p. 209–228.
- [72] POURVATAN, B., SIRJANI, M., HOJJAT, H., ARBAB, F. Automated analysis of reo circuits using symbolic execution. *Electronic Notes in Theoretical Computer Science* 255 (2009), 137–158.
- [73] PRATT, V. R. A near-optimal method for reasoning about action. *Journal of Computer and System Sciences* 20, 2 (1980), 231–254.
- [74] SLIND, K., NORRISH, M. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics* (2008), Springer, p. 28–32.
- [75] SOZEAU, M. Subset coercions in coq. In *International Workshop on Types for Proofs and Programs* (2006), Springer, p. 237–252.
- [76] STIRLING, C. Modal and temporal logics for processes. In *Logics for concurrency*. Springer, 1996, p. 149–237.
- [77] SUN, M., LI, Y. Formal modeling and verification of complex interactions in e-government applications. In *Proceedings of the 8th International Conference on Theory and Practice of Electronic Governance* (2014), ACM, p. 506–507.

-
- [78] TASHAROFI, S., SIRJANI, M. Formal modeling and conformance validation for ws-cdl using reo and casm. *Electronic Notes in Theoretical Computer Science* 229, 2 (2009), 155–174.
 - [79] ZHANG, X., HONG, W., LI, Y., SUN, M. Reasoning about connectors in coq. In *International Workshop on Formal Aspects of Component Software* (2016), Springer, p. 172–190.
 - [80] ZHANG, X., HONG, W., LI, Y., SUN, M. Reasoning about connectors using coq and z3. *Science of Computer Programming* 170 (2019), 27–44.