

UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Erick Simas Grilo

Compiling certified Reo code

Niterói-RJ

2018

Ficha catalográfica automática - SDC/BEE

G858c Grilo, Erick Simas
Compiling certified Reo code / Erick Simas Grilo ; Bruno
Lopes, orientador. Niterói, 2018.
190 f. : il.

Trabalho de Conclusão de Curso (Graduação em Ciência da
Computação)-Universidade Federal Fluminense, Escola de
Engenharia, Niterói, 2018.

1. Assistentes de prova. 2. Reo. 3. Constraint Automata. 4.
Produção intelectual. I. Título II. Lopes, Bruno,
orientador. III. Universidade Federal Fluminense. Escola de
Engenharia. Departamento de Ciência da Computação.

CDD -

Erick Simas Grilo

Compiling certified Reo code

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Bruno Lopes

Niterói-RJ

2018

Erick Simas Grilo

COMPILING CERTIFIED REO CODE

Trabalho submetido ao Curso de
Ciência da Computação da Universidade
Federal Fluminense como requisito parcial
para a obtenção do título de Bacharel em
Ciência da Computação.

Aprovado em Dezembro de 2018

BANCA EXAMINADORA




Prof. Bruno Lopes, D.Sc. — Orientador

UFF



Profa. Aline Marins Paes Carvalho, D.Sc.

UFF



Prof. Edward Hermann Haeusler, D.Sc.

PUC-Rio



Prof. Mario Benevides, Ph.D.

UFRJ

Niterói-RJ

2018

Dedicado à todos que permitem que possamos enxergar mais longe ao apoiarmos-nos em seus ombros.

Acknowledgments

First of all I thank to my family, which has always given unconditional support in many situations along this way, specially to my brother, with whom I have shared the first two years into University almost exclusively. I also thank Kara, the best Labrador dog that exists, for many laughing situations and sometimes for her bad manners.

To my advisor Bruno Lopes, who after two years became a great friend. for the opportunity to work in this project, for every tip, advice and any other help he has offered me.

To the friends that I could share (mainly) laughs all over this way.

To all whom have helped me in any way in this project, specially the Coq-club mailing list members. I also thank the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for supporting this work.

Abstract

Critical systems require high reliability and are present in many domains. In other words, 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 graphical modelling coordination language which focuses on model such interaction by taking advantage of natural properties in distributed systems, such as remote function calls and message passing. Constraint Automata are defined as the most basic formal semantic for Reo. Therefore Constraint Automata provide formalisms to reason and certify properties regarding Reo connectors.

This work describes the constructive formalization of Constraint Automata in Coq proof assistant, including a compositional operation. The results regarding the obtained framework are discussed, along with the implemented theory and usage examples.

Keywords: Proof Assistants, Reo, Constraint Automata

Resumo Estendido

Sistemas críticos são aqueles nos quais a falha pode resultar em perda de vida, destruição significativa, alta perda financeira ou dano ambiental [1]. Em suma, sistemas que precisam de um alto nível de confiabilidade. Há diversos exemplos de sistemas críticos aplicados a uma ampla gama de áreas, de dispositivos médicos a sistemas nucleares.

As técnicas padrão de engenharia de *software* não são projetadas para lidar com sistemas não tolerantes a falhas. Em muitos domínios, tais sistemas precisam de uma maneira de garantir sua segurança, a fim de garantir que o sistema realmente atenda à confiabilidade exigida. Os sistemas formais compõem um *background* teórico e implementado (e.g. *software*) capaz de modelar e raciocinar sobre sistemas, garantindo (matematicamente) que os requisitos são atendidos e que os sistemas se comportam conforme o esperado.

Sistemas formais foram usados para certificar a linha 1 do metrô de Paris, França [2], levando-a a um sistema de metrô totalmente automatizado, eliminado a necessidade de construir uma nova linha de metrô e consequentemente economizando milhões de euros. A Airbus utiliza métodos formais para certificar sistemas de controle de aviação de suas famílias de aeronaves A318 e A340-500/60 [3, 4].

A ausência de tal certificação em sistemas críticos pode levar a cenários catastróficos: entre 1985 e 1987, o acelerador de elétrons médico Therac-25 esteve envolvido em (pelo menos) seis ocorrências de overdoses de radiação [5]. Este episódio levou várias pessoas à morte e feriu muitas outras. Isso aconteceu devido a uma combinação de fatores, nos quais podemos citar o excesso de confiança dos engenheiros de *software* e à falta de certificação [6]. Em março de 2018, a iniciativa de um carro sem motorista da Uber estava envolvida em um acidente que matou um pedestre em Tempe, Arizona¹.

¹<https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html>

Muitos sistemas modernos estão se tornando críticos. Perda financeira e até mortes podem resultar de suas falhas [1]. Assim, desde o final dos anos 80 e início dos anos 90, pesquisas foram direcionadas para a aplicação de métodos formais para sistemas críticos [7, 1, 8].

Uma abordagem comum consiste em modelar tais sistemas como Sistemas Ciber-Físicos. Garantir a segurança e a confiabilidade dos Sistemas Ciber-Físicos é um desafio atual [9]. Modelá-los em sistemas baseados em lógica permite o uso de uma ampla estrutura teórica e de *software* para certificar que propriedades requeridas por tais sistemas são satisfeitas.

O uso de sistemas lógicos para modelar e raciocinar sobre os Sistemas Ciber-Físicos parece ser uma abordagem promissora [10]. Assistentes de prova [11], como Coq [12] e Isabelle [13], levam à possibilidade de automatizar a verificação de tais sistemas e fornecer código certificado. Seu design é feito sob medida para automatizar muitos (quando possível, todos) os passos das provas. A base teórica leva a ver provas como programas e programas como provas o que permite transformar uma prova de que os requisitos são atendidos em um modelo em um código (i.e. um programa) certificado.

Exemplos industriais do uso de assistentes de prova já estão presentes em empresas como Mistubishi [14] e NASA². Compiladores certificados desenvolvidos usando assistentes de prova também encontram-se em uso [15].

O Coq é um dos mais proeminentes assistentes de prova. Ele lida com uma linguagem de alto nível para modelar, provar e fornecer código certificado automaticamente, além de possuir uma linguagem de táticas usada no processo de prova facilmente extensível.

Uma linguagem formal também com interpretação gráfica para a modelagem e verificação de sistemas é Reo. Reo [16] é uma linguagem gráfica baseada em coordenadas com canais e conectores para a modelagem e verificação de sistemas. Dados seus componentes básicos (e.g. canais capazes de modelar o fluxo de dados, filtros, filas, decompositores, sumidouro, sincronizadores etc.); ela se adequa de forma bastante natural à arquitetura orientada a serviços.

Constraint Automata [17] é o formalismo mais básico que denota semântica formal para Reo. Trata-se de um formalismo semelhante à autômatos finitos, onde as transições

²<https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/>

são dependentes de proposições lógicas acerca de dados vistos nas portas do autômato, que por sua vez denotam nós em Reo, representando pontos dos conectores Reo pelos quais passam fluxos de dados. Cada nó em Reo é visto (a grosso modo) como uma instância de um software modelado. Para cada conector Reo canônico há um *constraint automaton* associado [18].

O presente trabalho trata da formalização de *Constraint Automata* no Coq de forma a obter meios para verificação formal de conectores Reo por meio de *Constraint Automata*. São formalizados as principais definições referentes a este formalismo, a operação produto que compõe autômatos a partir de outros autômatos e os autômatos referentes aos conectores Reo também são formalizados.

Palavras-chave : Assistente de Provas, Reo, *Constraint Automata*

Contents

Abstract	vii
Resumo Estendido	viii
List of Figures	xiii
List of Tables	xiv
1 Introduction	1
2 Related Logic Formalisms	4
2.1 Classical Propositional logic	4
2.2 Calculus of Inductive Constructions	5
3 Coq	9
3.1 Proof Assistants	9
3.2 Coq	10
4 Reasoning About Reo	18
4.1 Reo	18
4.2 Constraint Automata	22
5 Constraint Automata in Coq	33
5.1 Constraint Automata	33
5.2 Product Automata	53
6 Usage Examples	66
6.1 A two-bounded FIFO connector	66
6.2 Alternating Bit Protocol	70

	xii
7 Conclusions and Further Work	73
A An example of certified Coq code	75
B Code for Usage Examples	77
B.1 Two-Bounded FIFO Code	77
B.2 Alternating Bit Protocol	86
C Constraint Automata Definitions	109
C.1 General Helpers	109
C.2 Core Definitions	110
C.3 Product Automata	149
References	173

List of Figures

3.1	A screenshot of CoqIde	11
4.1	Canonical Reo connectors as provided by [16]	21
4.2	Modelling of the Alternating Bit Protocol in Reo	21
4.3	A graphical representation of a two-bounded FIFO Constraint Automaton with no data constraints	23
4.4	Constraint automaton obtained by the resulting product of Reo connectors depicted in Figure 4.2	32
6.1	A graphical representation of the resulting two-bounded FIFO connector . .	66

List of Tables

4.1	Table containing basic Reo channels and their respective constraint automata.	29
4.2	Table the product operations for the constraint automaton of the Reo Connector in Figure 4.2	31
5.1	Constraint Automata formalized in Coq for the canonical Reo connectors (i)	48
5.2	Constraint Automata formalized in Coq for the canonical Reo connectors (ii)	49
6.1	Formalization of constraint automata in Coq regarding the Reo connector depicted in Figure 4.2	72

Chapter 1

Introduction

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 in order for this set of software components interacting together to produce a complete system. Software built following this idea require a way to coordinate how these components interacts with each other.

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 its safety in order to guarantee that the system 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.

Critical systems are systems in which failure may result in loss of life, significant destruction, high financial loss or environmental damage [1]. In short, systems that need a high level of reliability. There are many examples of critical systems applied in a wide range of areas, from medical devices to nuclear systems.

Formal systems were used to certify Paris Metro line 1 (Paris's subway system, in France) [2], leading it a fully automated subway system, consequently eliminating the need of building another subway line, saving millions of dollars. Airbus uses formal methods in order to certify avionics control systems of its families of aircrafts A318 and A340-500/60 [3, 4].

The absence of such certification in critical systems may lead to catastrophic sce-

narios: between 1985 and 1987, the Therac-25 medical electron accelerator was involved in (at least) six radiation overdoses occurrences [5]. This episode led several people to death and injured many others. This happened due to a combination of factors, in which we can cite overconfidence of the software engineers and the lack of certification [6]. In March 2018, a Uber’s driverless car initiative was involved in an accident that killed a pedestrian in Tempe, Arizona¹.

Many modern systems are becoming safety-critical. Financial loss and even deaths can result from their failure [1]. Hence, since the end of the 80’s and early 90’s, researches have applied formal methods for critical systems [7, 1, 8]. By about the same time, software production has shifted from building large, static systems from the ground to building complete systems by reusing existing pieces of software whenever possible.

Proof assistants [11], like Coq [12] and Isabelle [13], lead to the possibility of automatize the verification of such systems and provide certified code. They are computational tools that usually implement logic systems. Their design is tailored to automatize many (when possible all) steps of proofs. The theoretical background leads to see proofs as programs and programs as proofs.

The application of proof assistants can be seen in a wide variety of areas: among many examples, the usage of proof assistants to model and reason about (Cyber-Physical) Systems seems to be a promising approach [10]. Industrial examples of the usage of proof assistants are already present in companies as Mistubishi [14] and NASA². Certified compilers developed using proof assistants are also in use [15].

Reo [16] is a prominent channel-based coordination model which provides modeling mechanisms that takes advantage of distributed systems’ characteristics like remote function calls and asynchronous message passing. Reo’s primary objective is to provide means to produce the integration code of heterogeneous software components that are part of a system as advocated by Component Based Software Engineering.

Based on channels which describes interaction between software components (called “ports” in Reo context), Reo enforces the compositional modeling of component based software by providing canonical Reo connectors and a product operation that composes more complex connectors out of simpler ones.

¹<https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html>

²<https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/>

Constraint Automata [17] are introduced as the most basic formal semantics for Reo. Therefore constraint automata are formalisms that enable formal reasoning on Reo connectors, providing mechanisms to reason about data flow through Reo connectors and how channels behave.

There are many works on the usage of proof assistants to verify coordination models: [19] Implements Reo connectors directly in Coq, while Verdi provides a framework to implement and formally verify distributed systems [20].

As far as the author knows, the only work focused on formalizing Reo semantics in Coq is [19]. This work formalizes directly Reo connectors in Coq, enabling direct reasoning of only the formalized canonical Reo connectors. Therefore [19] lacks the notion of Constraint Automata as formal semantics for Reo connectors and the product construction proposed by [17] that produces more complex Constraint Automata compositionally out of simpler ones, also not taking advantage of Constraint Automata, formalizing their own Reo semantics based on the formalized channels' behavior.

This work introduces the usage of Coq to model instances of Reo channels by means of Constraint Automata. The proposed framework comprises formalization of basic aspects of Constraint Automata theory and the product operation that enables compositional construction of Reo channels. Canonical constraint automata for basic Reo connectors introduced in [18] are also provided in Coq.

The structure of the work is organized as follows. Chapter 2 provides a brief overview of the logical formalisms behind the used system, Chapter 3 introduces Coq by means of a simple usage example, Chapter 4 discuss the main aspects of Reo, Constraint Automata and its core notions, Chapter 5 discuss the approach hereby used to formalize the aspects aforementioned, Chapter 6 presents usage examples of the proposed framework while Chapter 7 ends the discussion, concluding the present work and pointing out possible future directions.

Chapter 2

Related Logic Formalisms

The present work proposes a logic based approach upon formalizing Constraint Automata in Coq. In this chapter the main aspects of the underlying logic theory present in Coq are recovered, with focus on the logic formalisms that Coq employs.

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 in order to produce more complex sentences out of simpler ones [21]. As a branch of logic that studies methods in order to reason about relationship and/or how to compose propositions out of other propositions, Classical Propositional Logic provides the necessary apparatus to deal with such study.

Being a relatively simple model, Classical Propositional Logic allows everyday situations to be modelled such as “Carlos is a car salesman” as a letter called a ‘propositional symbol’. Therefore, $A \equiv$ “Carlos is a car salesman” denotes the attribution to a the proposition mentioned the same way letters are used to represent numbers in mathematics.

Its roots date back to the third century B.C., with Aristotle’s logical works composing the earliest form of logic’s formal study registered. By around the second century B.C., an early version of what is known as Propositional Logic was formalized by Chrysippus, an Greek philosopher [22]. The Sophists, and later Plato (early 4th century B.C.) displayed an interest in sentence analysis, truth and fallacies by means that no one could have any doubt about these properties on sentences. Propositional Logic was later then

reinvented by Peter Abelard, a French philosopher in the twelfth century B.C. [23].

Propositions in this logic can be separated in two categories: atomic propositions, propositions with no logical connectives, and molecular propositions, which are propositions built from other propositions by means of logical connectives. Logical connectives can be found in the spoken language as adverbs such as “or”, “and” and “not”. Such connectives leads to the possibility of reasoning about connected molecular propositions.

In order to formally reason about these propositions, a formal system is defined containing a space to translate written sentences into propositional sentences (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 about 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 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).

Therefore, a well formed formula in Classical Propositional Logic is expressed by the following grammar, with $P, Q \in \Phi$: $P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \neg P$

2.2 Calculus of Inductive Constructions

The Calculus of Constructions (CoC) is proposed 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 Intuitionist Type Theory (as proposed by Martin-Löf [24]): by removing types the result is a pure λ -expression with its associated algorithm. Calculus of Constructions therefore leads to a

high-level functional programming language which let their users to formalize definitions and prove properties about them in the same computational environment [25].

The idea behind Calculus of Constructions is a formalism that relies heavily on the Curry-Howard Correspondence among proposition as 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.

Calculus of Constructions was conceived to show how powerful the Curry-Howard Correspondence is to Computer Science. In Calculus of Constructions, every term is a λ -expression. All expressions in CoC have types: 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 Calculus of Constructions holds 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 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 an 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), a result of research on extraction of programs from proofs in the Calculus of Constructions. An important aspect for this extension relies on the idea that inductive defined propositions and data types play a core role in any application [26]. Calculus of Inductive Constructions is the current logical formalism behind Coq system. This extension enables inductively defined data types as well as principles such as proofs by induction and recursively defined function.

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

- (i) the types **Prop**, **Set** and **Type** (the well-founded infinite hierarchy is hereby implicit) are terms.
- (ii) variables and constants are terms,
- (iii) if x is a variable, with T and U terms, then $\forall x: T, U$ is also a term.
- (iv) 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.
- (v) 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 .

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

$$\frac{\Gamma \vdash (MN : P) \quad \Gamma \vdash M \cong M_1 \quad \Gamma \vdash N \cong N_1}{\Gamma \vdash (MN) \cong (M_1N_1)}$$

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, where M' is M with $[N/x]$.

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

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 [25] provide a full overview on the rules and the general aspects of this system.

Chapter 3

Coq

This chapter covers in some detail the main aspects regarding the Coq Proof Assistant. Coq is a widely used proof assistant in many projects around the world [10, 15, 27]. 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 computational tools that aids their uses in the process of formalizing propositions (consisting on what one would like to prove) and the process of proving properties about formalized propositions [28].

The advent of such systems has led to a new way of proving theorems, by reducing the human effort involved in the whole process, by reducing human errors that could be introduced in a proof, such as typos and false steps erroneously applied. Therefore, by using a proof assistant, such errors are easily avoided, making the task of building up a proof easier to perform.

Their popularity began to rise in the mid-90's, when Intel faced the infamous "Pentium Bug", which resulted in the recall of millions of buggy chips and a loss of approximately USD 475 millions [29]. 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 industry as it was dubbed "impractical" [30] by people who judged the required mathematical skills as "impossibly difficult" and claims that its usage extends software

development lifecycle [31].

Nowadays the usage of such systems has transcended the academic environment, making its way to the software development industry in a variety of areas, in order to ensure properties about softwares or certain programs/functions: SiFive¹ is a startup based in San Mateo, California (in the Silicon Valley), which aims to use Coq in order to certify processors and other core pieces of software while Bella et. al. [32] uses a proof assistant to certify an electronic payment protocol.

Unlike Automated Theorem Provers [11, 33], systems that try to prove theorems automatically (sometimes with minimum help from a human being), Proof Assistants aim to be tools that are used in the process of constructing proofs about theorems, therefore relying on heavy user manipulation (although some of them lets their users to automatize some, if not all steps, of some proofs) [12].

3.2 Coq

Coq [12] is a proof assistant based on an implementation of Calculus of Inductive Constructions, a type theory created by Christine Pauling [34] as an extension of Calculus of Constructions, a type theory created by Thierry Coquand in 1985 [25]. It is a powerful system which aims at representing both functional programs and proofs in higher-order logic using only one programming language named Gallina [35], Terms of Gallina can represent programs as well as properties of these programs and proofs of these properties.

The main purpose of Coq is to offer a system that lets one develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct with respect to their specification, all within a single environment that also lets their users to define their own proof tactics [36], a powerful mechanism which other proof assistants lack [37].

Coq is a proof assistant similar to other proof assistants, namely HOL systems, a family of interactive theorem provers based on Church's higher-order logic including Isabelle/HOL [13], HOL4 [38] and PVS [39], which are proof assistants based on Church's higher order logic [40], a higher order logic based on Church's type theory.

As a proof assistant, Coq has been used for many purposes: to name a few,

¹<https://www.sifive.com/>

Gonthier [41] presents a machine checked proof of the four color theorem, while Leroy et. al. certifies a compiler in Coq [15] and Grilo & Lopes uses Coq to formalize a framework based on finite state machines to model and reason about Smart Cities software interaction [10]. Fig. 3.2 shows an overview of CoqIde, Coq’s official IDE.

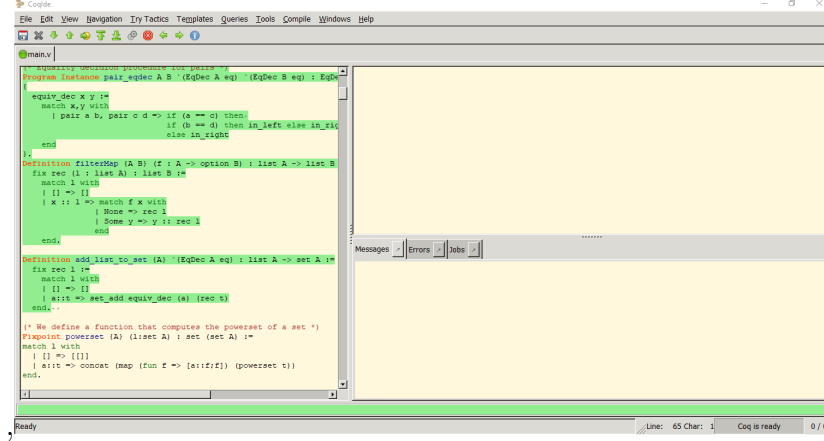


Figure 3.1: A screenshot of CoqIde

Besides Coq having an active community across the world and a (relatively) easy-to-follow documentation, one of its key aspects is the capability of converting certified Coq code to external languages such as OCaml or Scheme [27]. As far as the author is concerned, this is the only proof assistant that contains this functionality. Such aspects are the ones that directed us to choose Coq in the present work.

Coq’s language lets their users to formalize definitions and proofs in the same environment. This means that the definition of objects (integers, sets, trees, functions, programs...), making statements (using basic predicates and logical connectives) and (finally) writing proofs about the written statements happens in the same system.

Coq is structured through a two-level architecture : a small kernel based on a language with few primitive constructions (that comprehends functions and (co)-inductive definitions, product types, sorts, universes) and a small number of rules used for type-checking and computation and, on top of this kernel, lies all the apparatus Coq has to offer to its users: user extensible notations, tactics for proof automation, libraries, a mechanism that enables the possibility of one write its own proof tactics, among many other functionalities.

Therefore, types such as booleans, integers or strings which may be commonly found as primitive types in other programming languages are formalized as inductive

types in Coq. Such definitions are written as ordinary user code, although such types are provided by Coq’s standard library.

Another key property of Coq is that its language contains dependent types, which can be roughly described as types which 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 verify 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 [42].

All expressions formalized in Coq are named terms and all terms have a type. In other words, 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 is 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`, in order to avoid inconsistencies [43].

Sorts in Coq are categorized as an infinite well-founded hierarchy, beginning with the sorts `Prop` and `Set`. The sort `Prop` is the type of propositions. Any object of type `Prop` is said to be a proposition. An object p belonging to P with type `Prop` represents the class of terms denoting proofs of P . Therefore, p is said to be a witness of the provability of P . The sort `Set` is the sort of small sets, containing data types such as small user-defined data types, booleans and natural numbers, but also products, subsets, and function types over these data types.

The rest of this infinite hierarchy of sorts is organized through sorts of types `Type (i)`. `Type (i)` contain data types such as the ones that `Set` contains, as well as the sorts `Type (j)`, $\forall i < j$. The main reason in organizing sorts in Coq this way is to avoid paradoxes caused by the assumption that `Set` has type `Set`.

Since Coq’s theoretical foundation is based upon a variant of typed lambda calculus, it follows the functional programming paradigm. Gallina (which is Coq’s specification language name) lets one define functions and programs in a style that is very similar to other functional programming languages such as Haskell, allowing the definition of anonymous functions and compositions of functions [44].

In order to use Coq to formalize definitions and prove properties about them, Coq provides a built-in language to work with definitions and the proof process. In what

follows some of the keywords used in Coq are shown.

Lemma *id* : *type* denotes the binding of the type of a proposition (of type *type*) to the variable *id*, enabling proving of *id* interactively done using tactics. Other similar commands are **Theorem**, **Remark**, **Fact**, **Corollary** and **Proposition**. These keywords may have different meanings in real world but for Coq’s kernel they behave all the same. Definitions started with any of those keywords can be either closed by keywords **Qed** or **Defined**.

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** defines the proof term as a transparent term, being able to be unfolded in posterior programs. Recall that unfolding a constant means its replacement by its definition.

Inductive *ident* : *type* := {*ident* : *type*} defines an inductive type whose constructors are defined in {*ident* : *type*}, with a minimum of one constructor. The type of *ident* is *type* (which specification can be omitted, Coq’s type checker is capable of deducting the term’s type from its constructors).

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 in the environment named *id*, given that *id* has not already been used.

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. It follows the notion of fixed-point constructions as defined by the Knaster–Tarski theorem [45]. These definitions need to meet syntactical criteria on an argument called decreasing argument. This argument is needed in order to guarantee that the definition always terminate. Thus, the idea of the criteria is to have a structure that tells Coq such definitions always terminate. By using *struct id* where *id* is one of the parameters provided in *param* tells Coq which is the function’s decreasing argument. If not specified, Coq takes as the decreasing argument the leftmost provided parameter.

Section *id* opens a section named *id*. Sections are mechanisms used to structure proofs/programs in structured sections. Sections may be closed by the corresponding **End**

id. This sectioning mechanism eases the generalization of variables defined within a section, with respect to the variables and local definitions they are attached in the section.

Module *id* structures the program in a Module the same way a package groups similar elements together in Object-Oriented programming. A module may contain several definitions, sections and proofs and is closed by the **End** *id* keyword.

Instance *id class_id binders : type := { id := term }* declares a class identified by *id*, with non-obligatory parameters *binders* and the fields declared within the scope of $\{id := term\}$, having type *type*.

Extraction *id* enables the extraction of definition *id* to one either Haskell, OCaml or Scheme as target languages. 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).

Coq also comes packed with a canonical set of **tactics** which helps the user in the process of proving propositions. These tactics can also be extended by user action in order to adapt basic tactics to a certain domain. A tactic can be defined as an intermediate between the user and Coq’s language in order to deal with individual parts of a proof, enabling the possibility to adapt or automate a given part of a proof [36].

At each stage of a proof development, one has a list of goals to prove. Initially, the list consists only in the theorem itself. After having applied some tactics, the list of goals contains the subgoals generated by the tactics. Depending on the applied tactic, it may either reduce the number of goals (proving a sub-goal of the focused proof) or generating new sub-goals by using information from the proof context.

In a proof context, there are several hypothesis associated to a given subproof context, called the local context of a goal. Usually, it contains the local variables and hypothesis of the current section, as well as user-introduced constants in the context by means of keywords such as **Context**.

Tactics can only be used in the proof context. By applying a tactic on the focused goal, it may yield integers, tactics or even pattern matched cases against the term in

which the tactic has been applied to. At the end of the proof process, proof objects are formalized in Coq the same way as regular programs thanks to the Curry-Howard correspondence [46]. In what follows some examples of tactics are shown.

intros introduces in the proof context data related the the proven goal : if the goal is of type $\forall x : T, U$, *intros* saves in the proof context a hypothesis for each quantified variable $x : T$. For goals of type $T \rightarrow U$ introduces in the proof context a hypothesis $Hn : T$, if T has type **Prop** or **Set** or $Xn : T$, if T has type **Type**. Quantifiers in this case are treated as if the goal have type $\forall x : T, U$. Variant `intros $a_1 \dots a_n$` introduces hypothesis with names ranging from a_1 to a_n

destruct *term* is applied to any goal which is (co)-inductive, generating new subgoals, each for each constructor of *term*. If there is an variable being quantified in the goal, `destruct` also introduces these variables in the proof context by means of `intros`.

simpl aims to transform a term by reducing it to some intermediate term which is not fully normalized. In other words, `simpl` converts a term by reducing a definition based on pattern matching or unfolding a fixpoint definition. This tactic may be applied to goals and to hypothesis in the proof context.

reflexivity solves goals of the form $x = y$ if and only if it is able to transform x to y and vice-versa.

induction *term* start a proof by induction on *term*. This *term* must be an inductive type. This tactic creates a goal for each constructor of *term*. When the generated subgoals are proved, an induction hypothesis is added to the proof context automatically with identifier Hx , x being the type's identifier.

rewrite x replaces all occurrences of x 's left hand side in the current goal, replacing x by its right hand side. Note that x must occur in the proof context as an equality between two values. Then, x as $A = B$ replaces all occurrences of A with B in the goal. Variant `rewrite $\leftarrow x$` replaces occurrences of x 's right hand side in the current goal with x 's left hand side.

A simple yet representative usage example is introduced as follows: recall that Coq lets their users to define their own data definitions and programs, enabling the proof of

properties about user defined formalizations. 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* with a weekday returns the next day according with the current calendar. Definition *nextDay* (*day* : *weekdays*) :=

```
match day with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ saturday
| saturday ⇒ sunday
| sunday ⇒ monday
end.
```

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

Lemma *nextDayMonday* : $\forall \text{ day} : \text{weekdays}, \text{nextDay day} = \text{monday} \leftrightarrow \text{day} = \text{sunday}$.

Proof.

split.

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

- intros. rewrite *H*. reflexivity.

Defined.

One might also use Coq's defined data types in order to prove desired properties, such as properties on natural numbers. The function *plus2* defines the adds a natural number *n* with 2.

```
Definition plus2 (n:nat) := n + 2.
```

And also proofs about this function can be formalized, such as the following, which states that *plus2* returns *two* if the provided natural number *n* is *zero*.

Lemma *plus2Zero* : $\forall n, n = 0 \rightarrow \text{plus2 } n = 2$.

Proof.

intros. rewrite *H*. reflexivity. **Defined.**

The extraction of these definitions to other programming languages relies on 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* as depicted by Appendix A.

Chapter 4

Reasoning About Reo

This chapter addresses the main aspects of Reo, a graphic modelling coordination language, and Constraint Automata, defined as formal semantics for Reo. It also discusses in some detail Reo’s background, how Reo works, the functioning of Constraint Automata and how Constraint Automata denotes Reo’s formal semantics.

4.1 Reo

Since the nineties, software developers all around the world have researched new ways of how to produce software. New technologies and techniques have emerged since then, such as service-oriented computing [47] and model-driven development [48], where the first one promulgates the idea of composing software out of other software and the latter, developing software based on previous models.

Reo [16, 17] is a graphic-based coordination model based on channels where complex coordinators are compositionally built from simpler ones. These complex coordinators are called connectors and compose the very heart of Reo modelling language. Reo has as main goal to act as a “glue language”, a language that connects (“glues together”) instances of different components that act together in a component based system.

The usage of channel-based models is considered an alternative to advocate model-driven development of such “glue code”. A channel is an entity that connects two distinct ends with its own unique behavior. Channels can be seen as primitives for modelling concurrent systems. The usage of channel-based models brings a number of advantages by efficiently modelling primitives of concurrent systems (remote function calls, message

passing and shared memory, to name a few), enabling the capture of properties such as efficiency of how messages are exchanged and safety of how data can be inadvertently used by other entities in other modelling paradigms (such as shared data space modelling) by the model, a situation that does not occur in point-to-point communication.

As a coordination model, Reo focuses on connectors, their composition and how they behave, not focusing on the entities that are connected, communicate and work together through those connectors. Therefore, these entities may be modules of sequential codes, objects, agents, processes, web services and any other software component [16]. Such entities are called component instances in Reo.

The notion of Reo as an integrator between different instances of software component is further explored in the following idea: in Component Based Software Engineering, software components are expected to be independent from each other and more adapted to the environment they are meant to act on. Nowadays, software development is shifting from rather big, single instances of a system to reusable services, in order to produce a full application. This is the very heart of service-oriented computing, where such services can be modules, systems, web services or any other self-contained piece of loosely coupled software.

However, if the software development process considers the integration of such services by means of data exchange between their interfaces, the core of this data exchange (the specification of how such exchange happens) must be left out of each component in order to maintain software construction by (re)using seamless integration of existing services. Therefore a specific "glue-code" must be written for each built software.

Theoretically, service-oriented computing is analogue to exogenous coordination, where the coordination is done "externally" from the system's point of view, advocating the idea of separating the computation offered by integration of different, multiple services from how they integrate (the coordination of its data exchange) with each other [49]. Reo propagates such principles by providing out-of-the-box concepts and tools to develop the "glue-code" that acts as the coordinator of services' integration, given how their interface interacts with its outside world.

A system in Reo is composed by instances of software component that interacts with each other by means of Reo connectors. Component instances are defined as a non-empty set P that denotes a set of entities involved in an instance (process, services,

actors) and a predefined set of I/O operations associated with those entities where the only means of performing such operations are through channel ends connected to this set. A software component is a software implementation which 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 own unique predefined behavior. Each channel in Reo has exactly two ends. Channels are used to compose more complex connectors, being possible to combine themselves and to combine with the canonical connectors with provided by [17], although it is possible to build connectors from user defined channels, combining them with the basic Reo channels. Channel ends Fig. 4.1 shows the basic set of connectors as seen in [18].

A node in Reo is defined as a logical organization denoting the structure of how channel ends are linked to each other in Reo connectors. Therefore nodes denote an organization point where channel ends coincide. 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 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 (not simultaneously).

Channel ends can be used by any entity to disperse/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 data passing or receiving (depending on which channel end the entity has access to).

The bound between an instance and a channel end is a logical connection which does not rely on properties such as location of the involved entities. Channels in Reo have the sole objective to enable the data exchange by means of 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. Both channels and software components can be mobile entities. This leads to the possibility of the configuration of Reo connectors to change dynamically.

The mobility of Reo channels and instances does not affect the connections of

instances and channels, which only depends on the instance's will to connect to (or disconnect from) channel ends. This idea reflects the idea that, unrelated with their location, a software instance may be connected to another software instance while moving or after moving from its original location to elsewhere, where the channel only provides the medium of how they interact with each other.

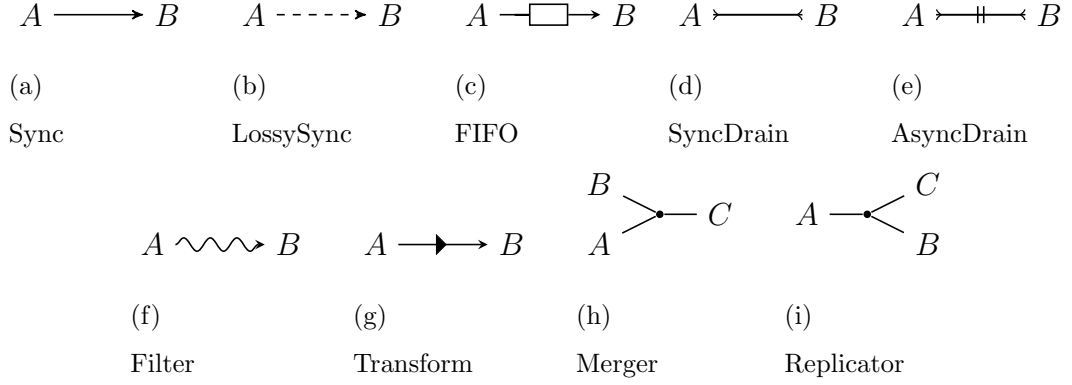


Figure 4.1: Canonical Reo connectors as provided by [16]

Reo therefore models many real world software interaction by means of its connectors. An example is a model of the Alternating Bit Protocol, a variant of the Sliding Window Protocol, which is a protocol implemented at the link layer, the second layer following the TCP/IP model of computer networking, where the size of the window is one bit. This protocol defines two communication channel between the parties involved, one for transmitting data and other for synchronization.

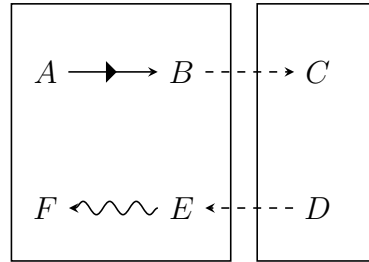


Figure 4.2: Modelling of the Alternating Bit Protocol in Reo

Figure 4.2 models the Alternating Bit Protocol in Reo, modelling the synchronization protocol channel. The left block denotes the sending entity while the right one denotes the receiving entity. The idea behind this connector is to synchronize both communicating parties if the same data item in B is seen in E , otherwise the synchronization is not achieved, also enabling data loss between these entities.

4.2 Constraint Automata

Constraint Automata [17] are defined as the most basic operational models for Reo, although there are many other formal semantics for Reo [50]. The present work focuses on Constraint Automata as proposed by Reo creators and is one of the most prominent formalism to reason about Reo connectors.

First proposed as a formalism to denote and reason about coordination models described as Reo models, Constraint Automata can be seen as a variation of Finite Automata where the transitions are influenced by ports containing data and data constraints over those ports, rather than depending only on the value seen in the input. These ports are Reo channel's ends, where the idea of establishing constraints over the data flow enables the possibility to enrich modelling scenarios, suiting well Reo functioning.

Therefore, Constraint Automata composes a basis on modeling and verifying the specification of such coordination mechanisms by usage of formal methods (e.g., by means of model checking against temporal-logic specifications [51]). By using Constraint Automata as formal semantics for Reo, automata states depicts the possible configurations of a channel (e.g., the data within a connector at a given time), while transitions of the automaton denotes how data in the connector flows and how it changes the configuration of the automaton.

Formally, Constraint Automata are defined as follows.

Definition 2 (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.

In what follows we recover and discuss the main concepts from Arbab et. al. [52, 53] on Constraint Automata. Fig. 4.2 shows a graphical description of a two-bounded Constraint Automaton with no data constraints over which data item can be enqueued

by the automaton. The idea of this automaton is to verify if the given data flow describes the behavior of a queue with exactly two data items (without any data restriction upon the data flow), namely data items flowing from port A to port B in a Reo connector.

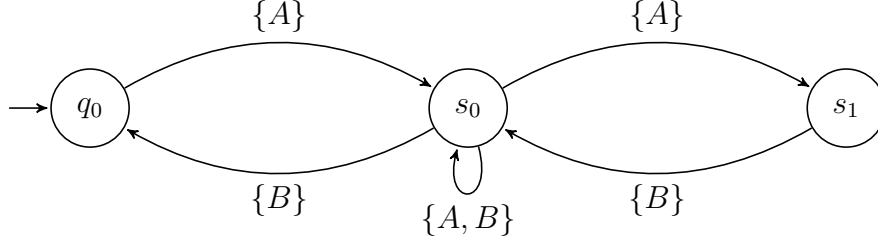


Figure 4.3: A graphical representation of a two-bounded FIFO Constraint Automaton with no data constraints

The behavior of Reo channels modelled as Timed Data Streams (introduced in Definition 3) as depicted in [53] introduces the notion of a channel node's behavior being a relation $R \subseteq TDS \times TDS$. 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. Therefore streams are used to model how data flows through a Reo connector by discriminating the flow by means of the relation R .

Streams are defined as a set A^ω containing all infinite sequences over A , where A can be any set. Hence, $A^\omega = \{\alpha \mid \alpha: \{0, 1, 2, \dots\} \rightarrow A\}$. 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 $\alpha' = \alpha(1), \alpha(2), \alpha(3), \dots$ with $\alpha^{(i)}$ denotes the i -th derivative, where $\alpha'(k) = \alpha(k+1)$ and $\alpha^{(i)}(k) = \alpha(i+k), \forall i, \forall k > 0$.

Constraint Automata are seen as Timed Data Stream acceptors. A Timed Data Stream (TDS) is a pair defining two streams, α as a stream on a set depicting the data domain of the automaton and a as other stream depicting the time instant a data element is in a port. The time stream a denotes, for each data element $\alpha(k)$ (k a natural number) the time instant $a(k)$ it happens in the port.

Formally, a TDS is defined as below.

Definition 3 (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.

In order to formalize the concept of input/output behavior of Constraint Automata by means of TDS, a set of names $Names$ is used, where $Names$ consists of a finite set of names A_1, A_2, \dots, A_n used to identify the input/output ports that connects different components or the whole system with the environment it is inserted. For each port $A_i \in Names$, a TDS is defined. Therefore, TDS^{Names} is defined as the set of all TDS tuples defined for each port $A_i \in Names$ formally as below.

Definition 4 (TDS^{Names}). *TDS^{Names} is a set containing a TDS for each port $A_i \in Names$ as*

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

When defining TDS^{Names} as above, it is assumed that $Names$ consists of a set of ports A_1, A_2, \dots, A_n (i.e., that there is a fixed enumeration of the ports $A_i \in Names$ such as the i -th timed data stream (α_i, a_i) is associated with the port A_i). However, it can be the case where such enumeration does not exist. In such cases it is used a notation $\theta = (\theta|_A)_{A \in Names}$ in order to map each TDS (α_i, a_i) to a port $A \in Names$.

A Data Assignment denotes which data element is in each port that belongs to a non-empty subset of ports $N \subseteq 2^{Names}$. Hence, a Data Assignment is defined as a function $\delta: N \rightarrow Data$.

Definition 5 (Data Assignment). *A Data Assignment is defined as*

$$\delta = [A \rightarrow \delta_a] : A \in N$$

Definition 5 describes the Data Assignment for any port $A_i \in N$, depicting the data item $\delta_a \in Data$ in the port.

By defining $\theta = ((\alpha_1, a_1), (\alpha_2, a_2), \dots, (\alpha_n, a_n)) : (\alpha_i, a_i) \in TDS^{Names}$, $\theta.time$ is defined as the time stream obtained by merging all timed streams a_1, a_2, \dots, a_n increasingly.

Definition 6 ($\theta.time(k)$). *The merging of time streams in increasing order denotes $\theta.time(k)$ as*

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

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

Therefore, $\theta.time(k)$ indicates the minimum time in where data starts to flow in a port, namely, the data item denoted by $\alpha(k)$. The next definition captures the idea of reuniting all ports that are in $\theta.time(k)$, $\theta.N = \theta.N(0), \theta.N(1), \theta.N(2), \dots$, a stream over 2^{Names} as

Definition 7 ($\theta.N(k)$). $\theta.N(k)$ denotes all ports that contains data in time instant $\theta.time(k)$:

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

By defining θ and $\theta.N(k)$, the derivative of θ is written θ' denoting 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 7, 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 containing the data assignments for each port $A_i \in \theta.N$. Intuitively, $\theta.\delta(k)$ holds all observed data flow at time $\theta.time(k)$ and is defined as follows.

Definition 8 ($\theta.\delta(k)$). 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 $a_i(l_i) = \theta.time(k)$.

The fact that Timed Data Streams $(\alpha_i, a_i) \in \theta$ where $\theta \in TDS^{Names}$ are required to be infinite (as depicted by Definition 3) denotes that, for any port $A_i \in Names$, there will be an infinite amount of indexes k where $A_i \in \theta.N(k)$. This requirement leads to the assumption of each port A_i there is an infinite data flow. Such assumption simplifies the modelling process but does not allow the modelling of situations such as a coordination mechanism blocking the data flow, configuring a deadlock situation.

It is interesting that Timed Data Streams as hereby defined does not distinguish between input and output data, only denoting data flow at a port. However, if necessary, it is possible to assume a fixed classification of ports as “input” and “output” ports and hence knowing whether a data is written or read. Alternatively, the data domain can discriminate its inhabitants in written or read values by denoting whether a data item denotes a “written value” or a “read value”.

A TDS language (for $Names$) denotes any subset of TDS^{Names} where, since for Reo circuits, its compositional semantics is provided by reasoning upon Timed Data Streams, TDS languages are also used as a formalism to describe the possible data flow for Constraint Automata.

Constraint Automata uses a finite set $Names$, where $Names$ can be a set as $\{A_1, A_2, \dots, A_n\}$ where the i -th port stands for a I/O port of a Reo connector or component. As depicted by Definition 2, transitions of Constraint Automata are labeled with pairs containing a non-empty subset $N \subseteq Names$ 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 that at port A the data item observed must be d , with $A \in Names$ and $d \in Data$.

Definition 9 (Data constraints). *A data constraint (DC) g is formally defined by the following grammar:*

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

The grammar described in Definition 9 suffices to express all propositions that can be modelled by means of the classical connectors from this logic (they can be derived from the grammar as well). In order to ease reading, a transition is denoted by $q \xrightarrow{N,g} p$ rather than $(q, N, g, p) \in \rightarrow$, \rightarrow being the transition relation defined in Definition 2, $q, p \in Q$ as states of the automaton, g a data constraint and $N \subseteq Names$. For each transition, it is required that $N \neq \emptyset$ and g is satisfiable by $\theta.\delta$ at that point (i.e., $\theta.\delta(k) \models g$, where \models stands for the classical satisfaction relation).

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 the automaton in configuration q_0 has data flow in ports $A \in N$ and this data flow meets the data constraint denoted by g , while in the other ports $Names \setminus N$ there must not have data flow.

By describing Constraint Automata as TDS acceptors, their behavior may be described as follows: given an input TDS-tuple $\theta \in TDS^{Names}$ as input to a Constraint Automaton \mathcal{A} , it 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 criteria for acceptance, all accepting runs are infinite runs if θ is infinite.

A run on \mathcal{A} may start in one of its initial states $q_0 \in Q_0$ and from this state, \mathcal{A} will wait for a data item happen in at least a port $A_i \in \text{Names}$. From q_0 , the automaton will check whether there is at least one transition from q_0 to any other state in which the ports involved satisfy the data constraint g attached to the transition. In other words, for example, given that data items flow in ports A_1 and A_2 ($A_1, A_2 \in \theta.N$), \mathcal{A} will check if from q_0 there is any transition labeled with $[A_1, A_2]$ where the data assignment depicted by $\theta.\delta$ here satisfies g . If a transition $q_0 \xrightarrow{[A_1, A_2], g} q_1$ ($q_1 \in Q$) which satisfies these conditions exists, then \mathcal{A} changes its configuration to q_1 , otherwise, \mathcal{A} rejects the input. In a broader sense, only transitions labeled by ports $A_i \in \theta.N$ with its data constraint g fulfilled may fire, where other transitions containing ports in which no data occurs may not.

Therefore, the condition of transitions being labeled by a non-empty subset $N \in \text{Names}$ stands for the requirement of automata transitions only firing if data takes place in ports $\{A_1, A_2, \dots, A_n\} \subseteq 2^{\text{Names}}$ where the condition of a data constraint g guarding the transition ensures that its firing depends only on the data seen in the present moment, not being affected by data that will occur somewhere in future.

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

Definition 10 (Runs in Constraint Automata). *Given a TDS-tuple $\theta \in TDS^{\text{Names}}$ as input, the set of infinite runs in a Constraint Automaton \mathcal{A} is denoted by the greatest set of streams $q = q_0, q_1, q_2, \dots$ over Q where:*

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

where (i) denotes that it is necessary to have at least one transition that can be fired from the actual state in the run, with (ii) as the requirement of no other ports other than the ones involved in a firing transition contains data, while (iii) states that the data on those ports must satisfy g and (iv) represents that the same behavior is expected in the rest of the run.

As discussed in Sect 4.1, [16] provides the canonical set of Reo connectors which may be used to compose more complex channels. Because Constraint Automata is a theory that provides formal semantics for Reo connectors, constraint automata for each canonical connector (depicted in Fig. 4.1) are also provided, each in accordance with its respective channel's behavior. Tab. 4.2 denotes the most basic channels provided by [16] and its representation by means of Constraint Automata. The label depicted in the edges between $\{\}$ are the ports that may “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 constraints for data in this transition.

As constraint automata serves as operational models for Reo connectors, [17] provides the most basic Constraint Automata for each basic Reo connector (as discussed in Chapter 4.1) in order to advocate compositional modeling (also being possible to use user defined basic Constraint Automaton).

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 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 [17].

Further exploring this idea, let two Reo circuits with nodes denoted by $Names_1$ and $Names_2$, respectively. The idea of the product operation is to join all common nodes $B \in Names_1 \cap Names_2$, while also maintaining nodes $A \in Names_1$ and $C \in Names_2$ where $A, C \notin Names_1 \cap Names_2$ and $B \in Names_1 \wedge B \in Names_2$. Recovering the example defined in [17], let $L_1(A, B)$ and $L_2(B, C)$, where the notation $L_i(X)$ stands for “ L_i is a TDS language for the name set X ”. The product of $Names_1 \cap Names_2$ is the resulting TDS Language $L_1 \cap L_2(A, B, C)$ formalized as

$$L_1 \cap L_2 = \{ ((\alpha, a), (\beta, b), (\gamma, c)) : ((\alpha, a), (\beta, b)) \in L_1 \text{ and } ((\beta, b), (\gamma, c)) \in L_2 \}.$$

The product automaton which encapsulates the TDS language acceptor for the resulting TDS language of the product operation is formally defined as

Definition 11 (Product Automata). *Given two Constraint Automata $A_1 = (Q_1, Names_1, \rightarrow_1, Q_{0,1})$ and $A_2 = (Q_2, Names_2, \rightarrow_2, Q_{0,2})$, the Product Automaton $A_1 \bowtie A_2$ is formally de-*

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

Table 4.1: Table containing basic Reo channels and their respective constraint automata.

defined as $A_1 \bowtie A_2 = (Q_1 \times Q_2, \text{Names}_1 \cup \text{Names}_2, \rightarrow, Q_{0,1} \times Q_{0,2})$, where \rightarrow is the resulting transition relation, defined as follows.

$$(i) \quad \frac{q_1 \xrightarrow{N_1, g_1} p_1, q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap \text{Names}_2 = N_2 \cap \text{Names}_1}{(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2)}$$

$$\begin{aligned}
(ii) \quad & \frac{q_1 \xrightarrow{N,g} p_1, N \cap \mathcal{N}ames_2 = \emptyset}{(q_1, q_2) \xrightarrow{N,g} (p_1, q_2)} \\
(iii) \quad & \frac{q_2 \xrightarrow{N,g} p_2, N \cap \mathcal{N}ames_1 = \emptyset}{(q_1, q_2) \xrightarrow{N,g} (q_1, p_2)}
\end{aligned}$$

Intuitively, the rules for constructing the resulting product automaton's transitions as the natural join of languages of both automata expresses this idea 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 the transition rules are built as defined, with the idea of

rule (i) stands for the join of both automata's transitions, in the sense of joining transitions being related with each other by respectively having their name set equal to the other automaton's name set. Considering the product of \mathcal{A}_1 and \mathcal{A}_2 , transitions from \mathcal{A}_1 with their name set intersecting the name set of \mathcal{A}_2 and transitions from \mathcal{A}_2 where the intersection of their name set with \mathcal{A}_1 where the result of both intersections are equal are then joined together as a single transition in the resulting product automaton.

rule (ii) and (iii) denotes the idea of preserving the transitions of \mathcal{A}_1 that are not related with \mathcal{A}_2 's transitions by means of the ports involved in each transition, i.e., these transitions does not have any port name in common with \mathcal{A}_2 's name set $\mathcal{N}ames_2$. Therefore, transitions from \mathcal{A}_1 that are not affected by \mathcal{A}_2 are maintained in the resulting product automaton (the same stands for transitions of \mathcal{A}_2).

An example of a constraint automaton is seen as follows. By recovering the Reo connector depicted in Figure 4.2, the constraint automaton for this connector is depicted in Figure 4.4. It is obtained by the product of the automata that compose the obtained Reo model. The sequence of operations that generates the resulting automata is denoted in Table 4.2.

The resulting constraint automaton is then obtained by doing the product of both constraint automaton in Table 4.2. Figure 4.4 contains the resulting automaton and its transitions.

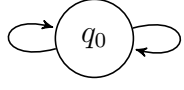
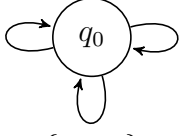
Reo	Constraint automaton
$A \twoheadrightarrow B$ $B \dashrightarrow C$	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> $\{A, B\}$ $f(d_A) = d_B$ </div> <div style="text-align: center;">  q_0 </div> <div style="text-align: center;"> $\{A, B, C\}$ $f(d_A) = d_B \wedge d_B = d_C$ </div> </div>
$D \dashrightarrow E$ $E \rightsquigarrow F$	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> $\{D\}$ </div> <div style="text-align: center;">  q_0 </div> <div style="text-align: center;"> $\{D, E, F\}$ $d_D = d_E \wedge P(d_E) \wedge d_E = d_F$ </div> </div> <div style="text-align: center; margin-top: 10px;"> $\{D, E\}$ $\neg P(d_E) \wedge d_D = d_E$ </div>

Table 4.2: Table the product operations for the constraint automaton of the Reo Connector in Figure 4.2

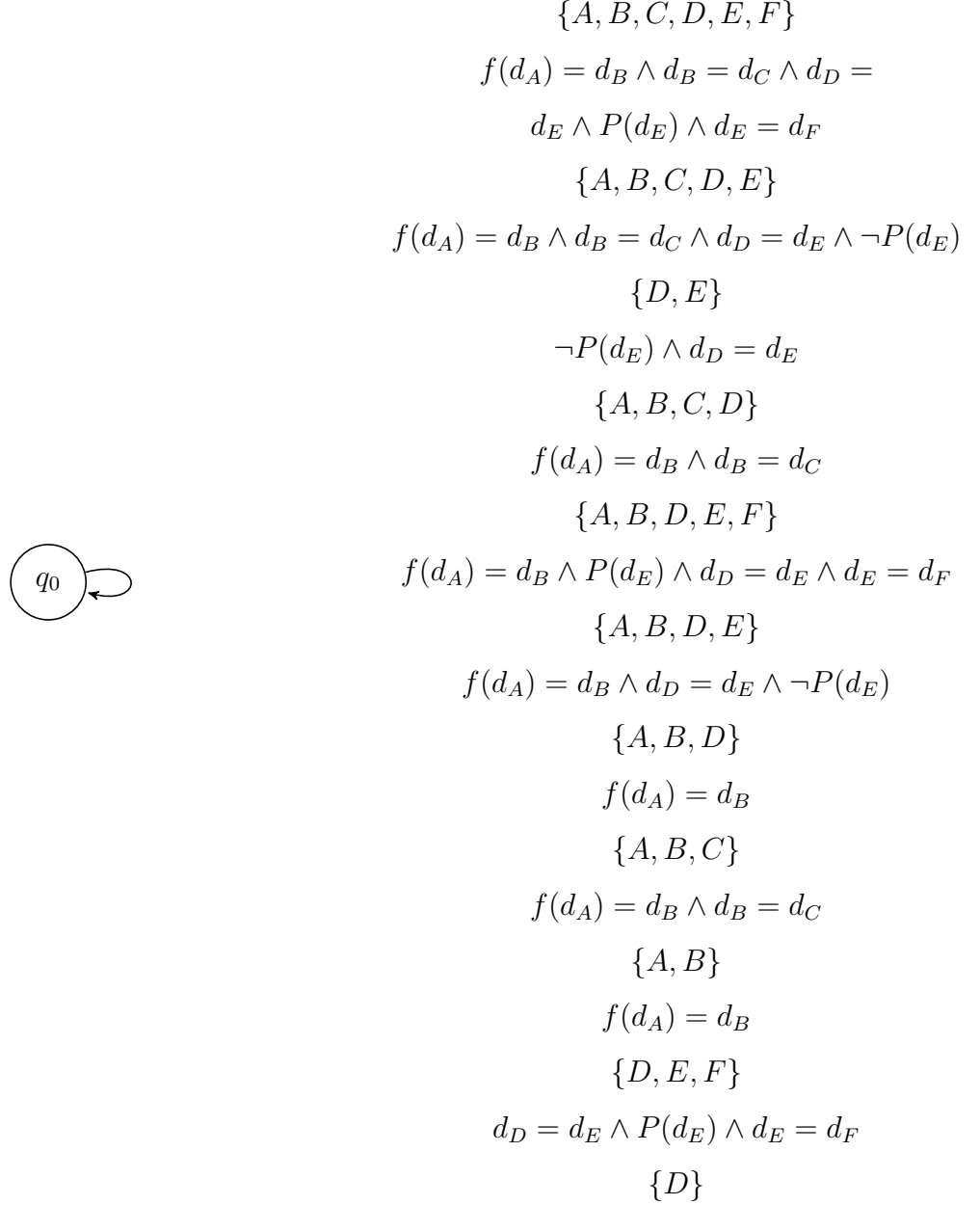


Figure 4.4: Constraint automaton obtained by the resulting product of Reo connectors depicted in Figure 4.2

Chapter 5

Constraint Automata in Coq

The adopted approach here is based upon Arbab et. al.’s definition of Constraint Automata [17] recovered in Chapter 4. In this Chapter we clarify the definitions in Coq used to define the main aspects of Constraint Automata hereby formalized.

The framework here presented was developed by using Coq 8.8 (by now the up-to-date version of the proof assistant) and its implementation can be found at <https://github.com/simasgrilo/CACoq>. Formalizing this theory in an environment such as Coq leads to the possibility of certifying instances of models which formal semantics relies on Constraint Automata according to its desired properties (namely Reo connectors) and consequently retrieve certified code in other programming languages by the discussed means in Section 3.2.

5.1 Constraint Automata

This formalization is done inside a Coq module defined like `Module ConstraintAutomata` which contains a section named `Section ConstraintAutomata`. The usage of `Module` stands for better organization of code (the same way similar structures of another programming languages, like Java packages, modularizes software development), while `Section` eases the developing task by generalizing its definitions with respect to the variables and local definitions they each depended on in the `Section`¹. Therefore, the type of objects hereby defined will depend on the type of its parameters as provided in runtime when instantiated.

¹<https://coq.inria.fr/distrib/current/refman>

Definition 2 is formalized in Coq as follows:

Definition 12 (Constraint Automata in Coq).

```

Context ‘{EqDec name eq} ‘{EqDec state eq} ‘{EqDec (option data) eq}.

Record constraintAutomata : Type := CA {
  Q : set state;
  N : set name;
  T : state → set (set (name) × DC × set(state));
  Q0 : set state;
}.

```

with variables *state* and *port* denoting the types of states and ports of the automaton. Its types are bounded to the automaton, since it is parametrized by both variables. All fields of the automaton depicts the fields that compose Definition 2, with *Q* being the set of states, *N* the set of port names (Reo ports), *T* denoting the transition relation of an automaton and *Q0* is the set containing the initial states of the automaton. The keyword *Type* denotes the most generic type in Coq, *Record* allows one to define records as done in other programming languages, creating an inductive type with only one constructor (namely an inductive type that expects as many parameters as fields defined) and the *set* variable stands for the implementation of sets as lists found in the library *ListSet*². Both variables *state* and *name* have type *Set*. The keyword *Context* require that equality decision procedures shall be provided for the types *name*, *state* and *option data*.

Let \mathcal{A} a constraint automaton as formalized in Definition 2. Therefore *constraintAutomata* is equivalent to \mathcal{A} : *Q* as a set of states stands for the set of states Q , *N* as a set of port names stands for the set of port names N , *Q0* as a set containing the initial states of *constraintAutomata* correspond to the set of initial states of \mathcal{A}

It is important to note that *T*'s definition here is slightly modified from the one presented in Definition 2: a transition relation *T* is modelled as a function that given a state q *T*(q) returns all possible transitions that leaves q , where each transition is denoted as a pair containing the set of names, the Data Constraint and resulting states associated with this transition, instead of modelling *T* as a function as depicted by Definition 2. The equivalence between *T* and T as depicted in Definition 2 lies on how these transitions are used in runs on Constraint Automata. Therefore this is done by means of Lemmas 21,

²<https://coq.inria.fr/library/Coq.Lists.ListSet.html>

22, 23 and 24.

Data Constraint (DC) here are formalized as an inductive type which encapsulates boolean behavior. This type will evaluate to Coq's boolean datatypes in runtime. Therefore, DCs as depicted by Definition 9 are defined as

Definition 13 (Data Constraint in Coq).

Inductive DC :=

- | *tDc : DC*
- | *dc : name → option data → DC*
- | *eqDc : name → name → DC*
- | *andDc : DC → DC → DC*
- | *orDc : DC → DC → DC*
- | *trDc : (option data → option data) → name → name → DC*
- | *negDc : DC → DC.*

where each constructor is:

tDc denotes a way to define *true* as a possible boolean value in the transition's Data Constraint,

dc stands for the definition of Data Constraints as $d_a = d$, where *name* is the identifier of the port that the data must be d ,

eqDc eases the formalization of data constraints " $d_a = d_b$ ", where *a* and *b* are port names. Such data constraints encapsulates the idea of two ports having the same data item at the same type,

andDc encapsulates boolean conjunction, considering as parameters two inhabitants of type *DC*,

orDc is the boolean disjunction prepared for *DC*, the same way *andDc* is the boolean disjunction

trDc encapsulates the idea of a data item transformed in a port to be equal to the data item in another port, as depicted by the Transform connector.

negDc denotes the boolean negation.

By defining Data Constraints as denoted by Definition 13, a way to evaluate DC to Coq's boolean data type is needed to evaluate whether a given port d_a has a data item d (the propositional formula $d_a = d$) and the composite Data Constraints. This is provided as follows.

```

Fixpoint evalCompositeDc (s:set port) (dc: DC) : bool :=
match dc with
| tDc ⇒ true
| dc a b ⇒ evalDC (retrievePortFromInput s a) (b)
| eqDc a b ⇒ eqDataPorts a b s
| andDc a b ⇒ evalCompositeDc s a && evalCompositeDc s b
| orDc a b ⇒ evalCompositeDc s a || evalCompositeDc s b
| trDc transform a b ⇒ transformDC transform a b s
| negDc a ⇒ negb (evalCompositeDc s a)
end.

```

Hence, *evalCompositeDc* provides boolean semantics for the inductive type *DC* which matches the grammar proposed in Definition 9, evaluating directly to its boolean counterpart in Coq. The symbol && is a notation for Coq's boolean “and” function and || is a notation for Coq's boolean “or” function.

The evaluation of Data Constraints formalized by means of *DC* uses *retrievePortFromInput*, a function that with a set of ports *s* and a port identifier *n*, returns the port $p \in s$ which *id* equals *n*, and *evalDC*, a function that given a port *po* and a data item *p* returns *true* if the data item calculated in *index* is *p* and false otherwise Both functions are defined as follows.

```

Fixpoint retrievePortFromInput (s:set port) (n: name) :=
match s with
| [] ⇒ None
| a::t ⇒ if (n == id a) then Some a else retrievePortFromInput t n
end.

Definition evalDC (po: option port) (d : option data) : bool :=
match po with
| Some p ⇒ if (dataAssignment p(index p) == d) then true else false
| None ⇒ false

```

end.

Definition *evalDC* then provides a way to evaluate whether a given port *po* has the data item *d*. The reason for *po* to be of type *option port* is that Coq needs a way to ensure that every function always terminate. Although its usage here there will not be the case of the set of ports provided to *retrievePortFromInput* being empty, since *retrievePortFromInput* iterates over a set, there must be an valid value to return in the case it is empty, which is one of the data type's constructor. . The notation *::* denotes the operation of concatenating elements inside a list as defined in Coq's standard library. Because *ListSet* extends *list*, one can apply operations bound to *list* in elements which type is *set*.

The evaluation of *eqDc* by means of *evalCompositeDc* relies on the idea of verifying in the TDS given as input if the ports in *eqDc* contains the same data item at the k-th step in a run on constraint automata. Therefore *eqDataPorts* comes as a function that with two port names *n1*, *n2* and a set of ports *s* returns *true* if both ports *n1* and *n2* has the same data item at the same moment.

```

Definition eqDataPorts (n1: name) (n2: name) (s: set port) :=
match (retrievePortFromInput s n1) with
| Some a ⇒ match (retrievePortFromInput s n2) with
            | Some b ⇒ if (dataAssignment a(index a)) == (dataAssignment
                        b(index b)) then true else false
            | None ⇒ false
end
| None ⇒ false
end.

```

Also, DCs formalized by means of *trDc* are evaluated by *transformDC*, a function that with a transformation function *transform*, two port names *n1* *n2* and a set of ports *s* also relies on *retrievePortFromInput* in order to retrieve the data item in TDSs denoting ports *n1* and *n2*, returning *true* if the data item flowing in *n2* is equal to the data item in *n1* the same time, transformed by *transform* and false otherwise.

```

Definition transformDC (transform: option data → option data) (n1: name)
(n2: name) (s:set port) :=
match (retrievePortFromInput s n1) with

```

```

| Some a  $\Rightarrow$  match (retrievePortFromInput s n2) with
  | Some b  $\Rightarrow$  if transform((dataAssignment a(index a))) ==
    (dataAssignment b(index b)) then true else false
  | None  $\Rightarrow$  false
end
| None  $\Rightarrow$  false
end. .

```

The variable *port* stands for a record defined in Coq which denotes a port of a Reo connector employed in Constraint Automata, describing the behavior of a single TDS (as denoted by Definition 3). Therefore, a port is defined as follows:

Definition 14 (Port in Coq).

```

Record port := mkport {
  id : name;
  dataAssignment : nat  $\rightarrow$  option data;
  timeStamp : nat  $\rightarrow$  QArith_base.Q ;
  portCond :  $\forall$  n:nat, Qle (timeStamp n) (timeStamp (S n));
  index : nat
}.

```

where *id* denotes the port's identifier, *dataAssignment* denotes a stream over *option data*, in which a port may contain a data item or no data at all, *timeStamp* stands for a stream over \mathbb{Q} , denoting a time stream of a port, *portCond* is a proof needed (which can be bypassed, if the user is safe to proceed without such guarantee, by considering it as an axiom in Coq) in order to define a port that the time stream is always crescent, the time stream denoted by *timeStamp* is a time stream over the set of rational numbers \mathbb{Q} as formalized in *QArith_base.Q*³, present in Coq's standard library since numbers in \mathbb{R} are only defined as theory in Coq (by means of lemmas and theorems), not being data types such as naturals. The field *index* denotes the index in which the port may be evaluated, used to calculate a TDS's derivative as discussed in Sect. 4. Following this definition, *port* then composes the input given to a constraint automaton being a set of ports of type *ports*: the notion of $\theta \in TDS^{Names}$ as depicted in Chap. 4 is then covered as the *set port* hereby used, with each port describing the behavior desired to that port to verify.

³https://coq.inria.fr/library/Coq.QArith.QArith_base.html

It is interesting to point that *mkport* is an alias for the constructor of this record. Therefore, one can defined a record of type *port* by using *mkport* with the required parameters.

The implementation aims to formalize the main concepts presented in Chapter 4. Therefore, the first step is to formalize $\theta.time$ as depicted by Definition 6. In order to do so, we start by defining *returnSmallerNumber* as a function that given a set of rational numbers and a rational number *m* returns the smallest rational number *k* that is in the set *l* given as parameter where $k < m$:

```

Fixpoint returnSmallerNumber (m:QArith_base.Q) (l:set QArith_base.Q)
:=
match l with
| []  $\Rightarrow$  m
| a::t  $\Rightarrow$  if ((a <? m)) then
    returnSmallerNumber a t else returnSmallerNumber m t
end.

```

Where $<?$ is a user-defined notation (by using the command *Notation*) for *Qle_bool*, a boolean function that with two numbers *a* and *b* returns *true* if *b* is greater than *a* and false otherwise. This function is used in order to retrieve the minimum a_i that will compose $\theta.time$ as stated by Definition 6. Therefore, *hasData* is defined as a function that given a port *p* and a natural number *k* returns whether the port contains data in time(*k*).

```

Definition hasData (p:port) (k:nat) :=
match (dataAssignment p(k)) with
| Some a  $\Rightarrow$  true
| None  $\Rightarrow$  false
end.

```

Next the objective is to formalize a way to retrieve the possible values from each port's time stream that is a candidate of $\theta.time$. The first step towards this functionality, the function *getThetaTimeCandidate* returns the current time stamp of a port *p* calculated on its index.

```

Definition getThetaTimeCandidate (p:port) := [timeStamp p(index(p))].

```

Therefore *getAllThetaTimes* is defined as a function that with a set of ports *s*

returns the $\theta.time$ candidates for the current step, retrieving all time stamps of each port $p \in s$ by means of *getThetaTimeCandidate*.

```

Fixpoint getAllThetaTimes (s: set port) :=
  match s with
  | [] => []
  | a::t => getThetaTimeCandidate a ++ getAllThetaTimes t
end.

```

Before formalizing $\theta.time$, *getNextThetaTime* returns the smallest number in the set of $\theta.time$ candidates l by means of *returnSmallerNumber*.

```

Definition getNextThetaTime (l: set QArith_base.Q) :=
  returnSmallerNumber (1000000#1).

```

The formalization of $\theta.time$ is achieved with *thetaTime*, a function that given a set of ports s and a natural number k , it returns the smallest number among all time streams from each port $p \in s$ as defined by Definition 6.

```

Definition thetaTime (s:set port) (Definition thetaTime (s:set port)
(k:nat) := getNextThetaTime(getAllThetaTimes s).

```

The next step is to formalize $\theta.N$ as represented by Definition 7. The idea starts with the definition of *timeStampEqThetaTime*, a function that with a set of ports s , a natural number k and a a port returns *true* if the port has its current time stamp in $\theta.time(k)$. In other words, it returns *true* if the port has its current index equal to $\theta.time(k)$.

```

Definition timeStampEqThetaTime (s:set port) (k:nat) (a:port) :=
  if ((timeStamp a(index a) ==? thetaTime (s) (k)) == true) then true else
  false.

```

thetaN implements the idea of $\theta.time$ as defined by Arbab. Given a set of ports ca , a natural number k and another set of ports s it returns a set containing all port identifiers that contains data at time $\theta.time(k)$. In order to do so, it iterates over s to evaluate all ports and ca as the full set of ports to calculate $\theta.time(k)$ within *thetaN*. It is later called with both s and ca as the input TDS.

```

Fixpoint thetaN (ca: set port) (k:nat) (s:set port) : set name :=
  match s with
  | a::t => if (hasData a k == true) then

```

```

    if (timeStampEqThetaTime ca k a == true) then
      id a :: thetaN ca k t
    else thetaN ca k t
  else thetaN ca k t
| [] ⇒ []
end.

```

The last basic definition on Constraint Automata is $\theta.\delta$. The function *portsWithData* implements the idea behind $\theta.\delta$, using a similar idea described by *thetaN*'s implementation, but returning pairs (portName, portData) instead of only the port identifiers. This will be used in order to calculate $\theta.\delta(k)$.

```

Fixpoint portsWithData (ca: set port) (k:nat) (s:set port) : set((name ×
option data)) :=
match s with
| a::t ⇒ if (hasData a k == true) then
  if (timeStampEqThetaTime ca k a == true) then
    ((id a) , (dataAssignment a(index(a)))) :: portsWithData ca k t
  else portsWithData ca k t
else portsWithData ca k t
| [] ⇒ []
end.

```

thetaDelta denotes the formalization of $\theta.\delta$ by means of *portsWithData*. Therefore, *thetaDelta* calls *portsWithData* with a natural number k denoting the current step and a set of ports po for both instances of po . It iterates over the same set of ports used to calculate $\theta.N(k)$.

```

Definition thetaDelta (ca:constraintAutomata) (k : nat) (po: set port)
:= portsWithData po k po.

```

After formalizing the basic definitions required to formalize Constraint Automata theory, the development aims to formalize the remaining functionalities for Constraint Automata, such as runs. In order to do so, the notion of derivative formalized here calculates the derivative as proposed in [17] slightly different: instead of calculating the derivative upon a stream, the notion of derivative is applied to a port. Hence a derivative of a port p is p with its index incremented by 1. The function *derivative* returns a port

with its updated index.

Definition *derivative* (p : *port*) := *mkport* (*id* p) (*dataAssignment* p)
 (*timeStamp* p) (*portCond* p) (*S* (*index* p)).

This definition will later be used in order to calculate the derivative of ports $p \in \theta$ used in each transition during a run. Hence, *derivativePortInvolved* with a set of names s and a port a verifies whether a 's *id* field matches one of the names in s .

Definition *derivativePortInvolved* (s :set *name*) (a : *port*) :=
match s with
 | [] \Rightarrow [a]
 | $x::t \Rightarrow$ *if* $x == id\ a$ *then* [*derivative*(a)]
 else *derivativePortInvolved* $t\ a$
end.

Then, *allDerivativesFromPortsInvolved* is a function that extends the behavior presented in *derivativePortInvolved* to a set of ports *port*. Hence, *allDerivativesFromPortsInvolved* with a set of names *names* and a set of ports *ports* applies *derivativePortInvolved* for each port $a \in port$.

Definition *allDerivativesFromPortsInvolved* (*names*: set *name*) (*ports*:set *port*) : set *port* := *flat_map* (*derivativePortInvolved* *names*) *ports* .

The function *flat_map* is a function within Coq's standard library which applies a function of type $A \rightarrow \text{list } B$ on all elements of a list of type list A , returning a list of elements of type list B . This function is common in another programming languages, mainly functional ones (such as Haskell or Scheme).

Another requirement that is related with runs in Constraint Automata consists on verifying whether when a transition is able to fire, it only fires if the ports involved with the transition are the only ports with data at the moment the transition is firing. The first step towards this formalization is *portsOutsideTransition*, a function that with a port p and a set of port names N verifies whether p 's *id* equals to a port name in N . The idea behind this definition is that the set of names that will be supplied is the names bound to a transition. If the port has its name in this set, then it is not outside the transition.

Fixpoint *portsOutsideTransition* (*input*: *port*) (*ports* : set *name*) :=
match *ports* with
 | [] \Rightarrow *true*


```

|  $a::t \Rightarrow \text{if } (id \text{ input} \neq a) \text{ then } portsOutsideTransition \text{ input } t \text{ else false}$ 
end.

```

The aforementioned idea is extended to a set of ports P (denoted by *input* in the following definition), by means of function *retrievePortsOutsideTransition* which applies *portsOutsideTransition* to a set of ports *input*. The idea behind is to retrieve all ports that are in s with its port name not in *ports*, where *ports* as a name set denotes the port names that are associated with a transition and *input* stands for the TDS $\theta \in TDS^{Names}$, which is given as input for the constraint automaton.

```

Fixpoint retrievePortsOutsideTransition (input: set port) (ports: set name)
:=
match input with
| []  $\Rightarrow$  []
|  $a::x \Rightarrow \text{if } (portsOutsideTransition a ports) == \text{true} \text{ then}$ 
       $a::retrievePortsOutsideTransition x ports$ 
     $\text{else } retrievePortsOutsideTransition x ports$ 
end.

```

Therefore, the last definition that encapsulates this functionality is *onlyPortsInvolvedContainsData*, which is a function that with a set of names *names*, two natural numbers k and l (which denotes the actual step of the run and the upper bound of valid values of the function, respectively) and a set of ports P .

```

Definition onlyPortsInvolvedContainsData (ports : set name)
(k : nat) (input : set port) :=
checkPorts (retrievePortsOutsideTransition (input) ports) (thetaDelta (k) (input)).

```

The idea behind this definition is to verify whether the ports that are not referenced in the current transition does not contain data at the moment, i.e., these ports does are not active in $\theta.delta(k)$, which is called as a parameter for *checkPorts*, a function that with a set of ports t and a set of pairs (*name*, *option data*) which is the type of return of *thetaDelta*, where *name* denotes a port name and *option data* the data item the port denoted by *name* has in $\theta.delta(k)$, returns *true* if there is no port $p \in t$ that has data in $\theta.delta(k)$. In what follows, let any occurrence of k and l has the same meaning denoted here, unless stated otherwise.

```

Fixpoint checkPorts (t:set port) (thetadelta: set (name  $\times$  option data)) :=

```

```

match  $t$  with
| []  $\Rightarrow$  true
|  $a::x \Rightarrow$  if (negb (hasDataInThetaDelta  $a$  thetadelta) == true) then
    checkPorts  $x$  thetadelta
    else false
end.

```

This definition uses *negb*, boolean negation as defined in Coq's standard library in order to invert the value returned by *hasDataInThetaDelta*, a function that with a port p and a set of pairs (*name*, *option data*) (just as defined in *checkPorts*) verifies if the set has a pair corresponding to p 's data item at $\theta.\text{delta}(k)$ as defined below. Definitions *fst* and *snd* hereby used are standard Coq functions for pairs where the first returns the first element of the pair and the latter returns the second element of the pair.

```

Fixpoint hasDataInThetaDelta ( $p$ : port) (thetadelta: set ( $\text{name} \times \text{option data}$ )) :=
match thetadelta with
| []  $\Rightarrow$  false
|  $a::t \Rightarrow$  if ((id  $p$  == (fst( $a$ )))) then
    if snd( $a$ )  $\neq$  None then true
    else hasDataInThetaDelta ( $p$ ) ( $t$ )
    else hasDataInThetaDelta  $p$   $t$ 
end.

```

All definitions so far were implemented following [17]. These definitions are core topics in Constraint Automata theory. Such definitions culminate in runs on Constraint Automata. Definitions regarding this functionality can now be formalized, starting with *retrievePortsFromThetaN*, a function that with natural numbers k , l and a set of ports p calculates $\theta.N(k)$ by means of *thetaN*.

```

Definition retrievePortsFromThetaN ( $k$  : nat) (input: set port) :=
thetaN (input) ( $k$ ) (input).

```

The idea is to incrementally define a run by means of auxiliary definitions. The first definition regarding this sequence of definitions is *step* that with natural numbers k regarding the i -th index of the current run, a set of ports *input* and a set of port names *ports* and a set of tuples (*name*, *DC*, *set(state)*) denoting the automaton's transition

relation returns a set containing all possible states reachable in the current step.

The idea of *step'* is to store all possible paths from a given state at a given point in the run. It defines how a single step in a run functions: exploring all transitions departing from a state, for all k steps, *step'* will verify whether the following requirements are fulfilled:

- the transition has its name set equal to the name set given as parameter (which will later be supplied with the name set returned by *thetaN*, denoting the name of ports with data in $\theta.N(k)$);
- the ports pointed by this transition are the only ones with data in time $\theta.time(k)$ by means of *onlyPortsInvolvedContainsData*;
- the data constraint depicted in the transition is satisfied by the available data at the ports in this moment.

The aforementioned topics are the necessary in order to a transition to be fired according to [17]. Hence, *step'* is defined as below.

```

Fixpoint step' (k: nat) (input : set port) (ports: set name)
(s: set (set name × DC × set(state))) :=
match s with
| [] ⇒ []
| a::t ⇒ if (set_eq (ports)((fst(fst(a))))) &&
           (onlyPortsInvolvedContainsData ca (fst(fst(a))) k input)
           && (evalCompositeDc (input) (snd(fst(a)))) then
           snd(a)++step' ca k input ports t
           else step' ca k input ports t
end.

```

The definition of *stepAux* as an supporting function that encapsulates the usage of *step'* with the name set as the Constraint Automaton's transition set. Therefore, *stepAux* is a function that with a Constraint Automaton *ca*, the index k , a set of ports *input*, a set of port names *ports* and a state *s* of *ca* calls *step'* with all transitions departing from *s*, by applying *T* as the transition function of *ca* with *s*.

```

Definition stepAux (ca:constraintAutomata) (k: nat) (input:set port)
(ports:set name) (s: state) := step' k input (T ca s) ports.

```

Next is defined *stepa* as a function which all parameters as *stepAux* but instead of *s* as a single state, *stepa* expects a set of state Q (denoted by *s*) in order to apply *stepAux* in every state achieved in the following configuration. In other words, *stepa*'s objective is to enable the run to evaluate all possible steps that can be achieved in moment *k*.

Definition *stepa* (*ca*:*constraintAutomata*) (*s*: set *state*) (*k*: *nat*)
 (*input*:set *port*) (*ports*: set *name*) :=
 (*ports*, *flat_map* (*stepAux* *ca* *k* *input* *ports*) *s*).

With *stepa* defined as above, a single step comprising the automaton's current configuration is defined as *step*, which given a Constraint Automaton *ca*, a set of states *s* which denotes the automaton's current configuration, index *k* and a set of ports *input*, applies *stepa* with these parameters, supplying the name set required by *stepa* the name set of ports $ports \subseteq \theta.N(k)$.

Definition *step* (*ca*:*constraintAutomata*) (*s*: set *state*) (*k*: *nat*)
 (*input*:set *port*) := *stepa* *ca* *s* *k* *input* (*retrievePortsFromThetaN* *k* *input*).

The run itself is achieved after formalizing *run'* as a function that iterates *k* steps over the set of ports given as parameter. In order to do so, *run'* is defined as a function which takes as arguments a Constraint Automaton *ca* a set of ports *input*, a set of natural numbers *k* ranging from 0 to *k*, a set of states *acc* and a set of set of states *resp* (an accumulator which holds the states during a single step) it returns all states which the automaton has passed through during the run.

The idea behind *run'*'s implementation lies in iterating over an ordered set from 0 to *k*. In each iteration, it applies *step* with the same parameters provided to *run*, storing the resulting states obtained with *step* in *resp*, calculating the derivatives of all ports involved in the transition fired by means of *derivativePortsInvolved* before recursively calling *run*.

Definition *run'* (*ca*:*constraintAutomata*) set *port* → list *nat* →
 set *state* → set (set *state*) → set (set *state*) :=
 fix rec *input* *k* *acc* *resp* :=
 match *k* with
 | [] ⇒ *resp*
 | *a*::*t* ⇒ *resp* ++ [*snd* (*step* *ca* *acc* *a* *input*)]
 |> *rec*

```

      (flat_map(derivativePortInvolved(fst((step ca acc a input))))
      input) t (snd (step ca acc a input))

end.

```

This functionality is encapsulated by *run* with a constraint automaton *ca*, a port set *input* and natural numbers *k* calls *run'* with the respective parameters, supplying *k* to *count_into_list*, a function that given a natural number *k* returns an ordered set from 0 to *k*, where the expected set of natural numbers by *run'* is therefore (*count_into_list* (*k*)), the set of states denoted by *acc* with *ca*'s initial states, which are the ones where the run may start and *resp*, which stands for the resulting set of set of states denoting the trace of a run in a constraint automaton with a set initially containing the set of *ca*'s initial states.

```

Definition run (ca:constraintAutomata) (input: set port) (k : nat) :=
run' ca input (count_into_list k) Q0 ca [Q0 ca]

Fixpoint count_into_list (n:nat) :=
  match n with
  | 0 => 0::nil
  | S n => count_into_list n ++ [S n]

end.

```

It is important to point that the notion of runs in constraint automata formalized here is slightly different from the one discussed in Definition 10: as [17] defines, runs in constraint automata are infinite. Roughly speaking, a run is an accepting run if the given TDS $\theta \in TDS^{Names}$ always fire at least one transition in the given automaton during the run and it is a rejecting run otherwise (if it reaches a state where no transitions can be fired). By algorithmic aspects, *run* formalizes the notion of a finite run bounded to *k* steps.

This notion of run is equivalent to the one presented by Definition 10 except that it implements finite run, but strictly implementing how a run works the same way Definition 10 introduces. Such equivalence is given by Lemmas 21, 22, 23 and 24.

By formalizing Constraint Automata in Coq by the aforementioned means, one can reason about Reo connectors by their constraint automata. In other words, given a Reo connector, it is possible to formalize in Coq its correspondent constraint automaton in order to verify desired properties.

The formalization of constraint automata of the canonical Reo connectors by means of CACoq can be obtained as follows, where Table 5.1 and Table 5.1 depicts constraint automata for canonical Reo connectors and shortly after their transition relation is introduced.

Reo Channel	Constraint automaton in Coq
Sync	$\text{Definition } \text{syncCA} := \{ $ $\text{ConstraintAutomata}.Q := [X];$ $\text{ConstraintAutomata}.N := [E;F];$ $\text{ConstraintAutomata}.T := \text{syncCaBehavior};$ $\text{ConstraintAutomata}.Q0 := [X]$ $ \}.$
LossySync	$\text{Definition } \text{lossySyncCA} := \{ $ $\text{ConstraintAutomata}.Q := [q0];$ $\text{ConstraintAutomata}.N := [A;B];$ $\text{ConstraintAutomata}.T := \text{lossySyncCaBehavior};$ $\text{ConstraintAutomata}.Q0 := [q0]$ $ \}.$
FIFO	$\text{Definition } \text{oneBoundedFIFOCA} := \{ $ $\text{ConstraintAutomata}.Q := [q0F;p0F;p1F];$ $\text{ConstraintAutomata}.N := [AF;BF];$ $\text{ConstraintAutomata}.T := \text{oneBoundedFIFOrel};$ $\text{ConstraintAutomata}.Q0 := [q0F]$ $ \}.$
SyncDrain	$\text{Definition } \text{SyncDrainCA} := \{ $ $\text{ConstraintAutomata}.Q := [q1D];$ $\text{ConstraintAutomata}.N := [AD;BD];$ $\text{ConstraintAutomata}.T := \text{syncDrainCaBehavior};$ $\text{ConstraintAutomata}.Q0 := [q1D]$ $ \}.$

Table 5.1: Constraint Automata formalized in Coq for the canonical Reo connectors (i)

It is important to point that the transitions of the automaton relies, among others,

Reo Channel	Constraint automaton in Coq
AsyncDrain	<p>Definition <i>aSyncDrainCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [q1A];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [AA;BA];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>aSyncDrainCaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [q1A]</p> <p style="padding-left: 40px;"> }.</p>
Filter	<p>Definition <i>filterCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [q1F];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [C;D];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>filterCaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [q1F]</p> <p style="padding-left: 40px;"> }.</p>
Trasform	<p>Definition <i>transformCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [q1T];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [AT;BT];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>transformCaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [q1T]</p> <p style="padding-left: 40px;"> }.</p>
Merger	<p>Definition <i>mergerCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [q1M];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [AM;BM;CM];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>mergerCaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [q1M]</p> <p style="padding-left: 40px;"> }.</p>
Replicator	<p>Definition <i>replicatorCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [q1R];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [AR;BR;CR];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>replicatorCaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [q1R]</p> <p style="padding-left: 40px;"> }.</p>

Table 5.2: Constraint Automata formalized in Coq for the canonical Reo connectors (ii)

the data type of the data to be observed by the ports. A brief explanation on the behavior of the aforementioned automata is introduced. To what follows the data type *nat* denoting naturals in Coq is used as the data domain.

Sync The idea behind Sync channel is to synchronize the data between two ports in Reo. Therefore *syncCaBehavior* is a definition that with an state of this automaton returns transitions as depicted by Definition 12. The transition relation is then defined as

```
Definition syncCaBehavior (s: syncState) :=
  match s with
  | X ⇒ [[E;F] , ConstraintAutomata.eqDc nat E F, [X]]
  end
```

where the idea behind *syncCaBehavior* is that both ports may have the same data object at the same time. The constructor of the inductive type denoting data constraints *DC* named *eqDc* encapsulates the formalization of such data constraint.

LossySync The idea behind LossySync channel is to enable the synchronization between two Reo ports, in the sense both ends of the channel must have the same data item simultaneously, but also letting only the source node of the connector to receive data (semantically denoting that the data item the ports should be synchronized to was lost when on its way to the sink node). This is the idea behind *lossySyncCaBehavior* which is

```
Definition lossySyncCaBehavior (s: lossySyncStates) :=
  match s with
  | q0 ⇒ [[A;B] , ConstraintAutomata.eqDc nat A B, [q0]];
        ([A], (ConstraintAutomata.tDc lossySyncPorts nat), [q0])]
  end.
```

FIFO The presented FIFO connector has capacity 1. The idea of this connector is to act like there is a memory area that stores an incoming data coming through the source node before passing it to the sink node. By establishing that the data domain of the data that flows within the connector is $[0, 1]$, the transition of the corresponding automaton can be formalized as follows.

```
Definition oneBoundedFIFOrel (s:FIFOStates) :=
  match s with
```



```

|  $q0F \Rightarrow [([AF], (ConstraintAutomata.dc\ AF\ (Some\ 0)), [p0F]) ;$ 
     $([AF], (ConstraintAutomata.dc\ AF\ (Some\ 1)), [p1F])]$ 
|  $p0F \Rightarrow [([BF], (ConstraintAutomata.dc\ BF\ (Some\ 0)), [q0F])]$ 
|  $p1F \Rightarrow [([BF], (ConstraintAutomata.dc\ BF\ (Some\ 1)), [q0F])]$ 
end.

```

SyncDrain The channel denoted by SyncDrain has the objective to model the idea of a channel which function is to drain data. In other words, let data go through the channel with no constraints on the data observed. Hence the transition relation is defined with a single transition as

```

Definition syncDrainCaBehavior (s: syncDrainState) :=
  match s with
  |  $q1D \Rightarrow [([AD;BD], ConstraintAutomata.tDc\ syncDrainPorts\ nat, [q1D])]$ 
  end.

```

AsyncDrain The idea of AsyncDrain channel is similar to the one SyncDrain denotes, but asynchronously, meaning that both ends of the channel can have data flowing but not necessarily at the same time. This behavior is captured in Coq as

```

Definition aSyncDrainCaBehavior (s: aSyncDrainState) :=
  match s with
  |  $q1A \Rightarrow [([AA], ConstraintAutomata.tDc\ aSyncDrainPorts\ nat, [q1A]);$ 
     $([BA], ConstraintAutomata.tDc\ aSyncDrainPorts\ nat, [q1A])]$ 
  end.

```

Filter The filter channel filters the data that flows through it: data flows from the source node to the sink node if the required condition is met, where the sink node must have the same data item the source node has or either the data stays in the source node, if the condition required is not met. For instance, suppose the required condition is that the data seen in the source node of this channel must equal three. This behavior is formalized as follows.

```

Definition filterCaBehavior (s: filterState) :=
  match s with
  |  $q1F \Rightarrow [([C;D], ConstraintAutomata.andDc\ (ConstraintAutomata.dc\ C$ 
     $(Some\ 3))\ (ConstraintAutomata.eqDc\ nat\ C\ D), [q1F]);$ 
     $([C], ConstraintAutomata.negDc\ (ConstraintAutomata.dc\ C$ 

```

$(\text{Some } 3)), [q1F]]]$

end.

Transform The transform channel introduces the idea of comparing whether the source node's data item transformed equals the data in sink node. In short, it transforms the data item in the source node to the one in the sink node by means of a user-supplied function *transformFunction*. This behavior can be expressed as below.

Definition *transformCaBehavior* ($s: \text{transformState}$) :=

match s with

| $q1T \Rightarrow [([AT;BT], \text{ConstraintAutomata.trDc transformFunction } AT \ BT, [q1T])]$

end.

where *transformFunction* can be any function of type *option data* \rightarrow *option data*.

Merger The merger channel acts like a multiplexer, where it lets data flow from one of its source node only if exactly one of them is active simultaneously to the sink node. The idea is to merge data respecting the idea of the channel must be used by at most one software component connected to it. Therefore the mentioned behavior is captured as follows.

Definition *mergerCaBehavior* ($s: \text{mergerState}$) :

set (set *mergerPorts* \times *ConstraintAutomata.DC mergerPorts nat* \times set *mergerState*) :=

match s with

| $q1M \Rightarrow [([AM;CM], \text{ConstraintAutomata.eqDc nat } AM \ CM, [q1M]); ([BM;CM], \text{ConstraintAutomata.eqDc nat } BM \ CM, [q1M])]$

end.

Replicator The idea behind the Replicator channel is to replicate data seen in its source node to all its sink nodes. In order to capture this behavior, the corresponding constraint automaton's transition relation is then formalized as follows.

Definition *mergerCaBehavior* ($s: \text{mergerState}$) :

set (set *mergerPorts* \times *ConstraintAutomata.DC mergerPorts nat* \times set *mergerState*) :=

match s with

| $q1M \Rightarrow [([AM;CM], \text{ConstraintAutomata.eqDc nat } AM \ CM, [q1M])];$

```

([BM;CM] , ConstraintAutomata.eqDc nat BM CM, [q1M]))
end.

```

5.2 Product Automata

CACoq also defines the construction of product automata as presented in Definition 11, implementing the necessary rules in order to produce all possible rules of product automata as specified by [17]. This implementation is done within a section defined as `Section ProductAutomata` within module `Module ProductAutomata` the same way Constraint Automata were formalized.

Product Automata are also Constraint Automata, therefore the same *record* defined for Constraint Automata will be the resulting automata's type. Recall from Definition 11 that a Product Automata is a tuple $\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, Names_1 \cup Names_2, \rightarrow, Q_{0,1} \times Q_{0,2})$. In what follows, let \mathcal{A}_1 and \mathcal{A}_2 be constraint automata formalized in Coq.

The resulting set of states as $Q_1 \times Q_2$ is formalized by means of *resultingStatesSet*, a function that given two constraint automata \mathcal{A}_1 and \mathcal{A}_2 generates the product set of \mathcal{A}_1 with \mathcal{A}_2 with *list_prod*, a function from Coq's standard library that returns the product of two lists.

```

Definition resultingStatesSet (a1:ConstraintAutomata.constraintAutomata
state name data) (a2:ConstraintAutomata.constraintAutomata state2 name data)
:= list_prod (ConstraintAutomata.Q a1) (ConstraintAutomata.Q a2).

```

$Names_1 \times Names_2$ as the resulting set of names is obtained with *resultingNameSet* as a function that with two constraint automata \mathcal{A}_1 and \mathcal{A}_2 returns the union of the set of names of \mathcal{A}_1 with the set of names of \mathcal{A}_2 by means of *set_union*, a function that denotes the union of two sets in `coqdocListSet`.

```

Definition resultingNameSet (a1:ConstraintAutomata.constraintAutomata
state name data) (a2:ConstraintAutomata.constraintAutomata state2 name data)
:= set_union equiv_dec (ConstraintAutomata.N a1) (ConstraintAutomata.N a2).

```

The product set denoted by $Q_{0,1} \times Q_{0,2}$ is defined as *resultingInitialStatesSet*, a function that with \mathcal{A}_1 and \mathcal{A}_2 as two constraint automata returns the product set of \mathcal{A}_1 's initial states set with \mathcal{A}_2 's initial state set.

```

Definition resultingInitialStatesSet (a1: constraintAutomata

```

```

state name data) (a2: constraintAutomata constraintAutomata state2 name data)
:= list_prod (ConstraintAutomata.Q0 a1) (ConstraintAutomata.Q0 a2).

```

The resulting transition relation is quite complex if compared to the other parts of the tuple that composes a product automaton. Its implementation is split in some functions, starting with *evaluateConditionsFirstRule* which with two transitions *t1* and *t2* and two names set *names1* and *names2* (denoting the names set of \mathcal{A}_1 and \mathcal{A}_2 respectively) returns *true* if the intersection of the set of names associated with *t2* and *names1* equals the intersection of the name set *t1* with *names2*. This first function is part of the implementation of the first rule depicted in Definition 11.

```

Definition evaluateConditionsFirstRule (t1 : (set(name) × DC × set(state)))
(t2 : (set(name) × DC × set(state2))) (names1 : set name)
(names2: set name) :=
if set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1)
(fst(fst(t2)))) then true else false.

```

Because transition relations are defined as $T: state \rightarrow set(set(name) \times DC \times set(state))$, from the first rule presented in Definition 11, states in the resulting automaton will be of type (q_1, q_2) , $q_1 \in Q_1$ and $q_2 \in Q_2$, Q_1, Q_2 as the set of states of \mathcal{A}_1 and \mathcal{A}_2 , respectively. This means that if a transition that satisfies *evaluateConditionsFirstRule* needs to have its resulting *set state* iterated in order to produce the correct resulting states of a transition. This is achieved by *buildResultingTransitionFromStatesRule1* with a state *p1* and a set of states *p2* returns pairs of states $(p1, a) \forall a \in p2$.

```

Fixpoint buildResultingTransitionFromStatesRule1 (p1: state) (p2: set state2)
:=
match p2 with
| [] ⇒ []
| a::t ⇒ (p1,a)::buildResultingTransitionFromStatesRule1 p1 t
end.

```

Therefore *buildResultingTransitionFromStatesRule1*'s behavior is extended to be applied to both sets of states of \mathcal{A}_1 and \mathcal{A}_2 respectively, *p1* and *p2* by means of *buildResultingTransitionFromStatesBothTransitionsRule1* as follows.

```

Fixpoint buildResultingTransitionFromStatesBothTransitionsRule1 (p1: set state)
(p2: set state2) :=

```

```

match p1 with
| [] => []
| a::t => buildResultingTransitionFromStatesRule1 a p2++
      buildResultingTransitionFromStatesBothTransitionsRule1 t p2
end.

```

The creation of a single transition is therefore captured by *buildResultingTransitionFromSingleStateRule1* with *Q1* and *Q2* states, *transition1* and *transition2* denoting a single transition of A_1 and A_2 respectively *names1* and *names2* denoting the name set of A_1 and A_2 , respectively, returns a transition as defined by the first rule in Definition 11.

```

Definition buildResultingTransitionFromSingleStateRule1
(Q1: state) (Q2: state2) (transition1: (set (name) × DC × (set(state))))
(transition2: (set (name) × DC × (set(state2))))
(names1 : set name) (names2: set name) :
(set (state × state2 × ((set name × DC) × set (state × state2)))) :=
if (evaluateConditionsFirstRule (transition1) (transition2) (names1) (names2))
== true then
  [((Q1,Q2),(((set_union equiv_dec (fst(fst(transition1)))
(fst(fst(transition2))))),ConstraintAutomata.andDc
(snd(fst(transition1))) (snd(fst(transition2))))),
(buildResultingTransitionFromStatesBothTransitionsRule1
(snd(transition1)) (snd(transition2)))))]
else [].

```

The next definition follows the idea of applying rule (i) from Definition 11 with a set of transitions by means of *buildResultingTransitionFromSingleStateRule1*: *buildTransitionFromMoreTransitionsRule1* applies *buildResultingTransitionFromSingleStateRule1* with its parameters, but instead of expecting *transition2* as a single transition, *transition2* now is a set of transitions on which *buildResultingTransitionFromSingleStateRule1* is applied.

```

Fixpoint buildTransitionFromMoreTransitionsRule1 (Q1: state) (Q2: state2)
(transition1: (set (name) × DC × (set(state))))
(transition2: set (set (name) × DC × (set(state2))))
(names1 : set name) (names2: set name) :=

```

```

match transition2 with
| [] ⇒ []
| a::t ⇒ (buildResultingTransitionFromSingleStateRule1 Q1 Q2 transition1 a
          names1 names2)++(buildTransitionFromMoreTransitionsRule1 Q1 Q2
          transition1 t names1 names2)

```

Then *buildTransitionFromMoreAllTransitionsSingleState* is the function that joins the behavior of *buildTransitionFromMoreTransitionsRule1* to a set of transitions. The idea is to verify two sets of transitions, namely the transitions of \mathcal{A}_1 and \mathcal{A}_2 which transitions satisfies the first rule as denoted by Definition 11. Then *buildTransitionFromMoreTransitionsRule1* is a function that with two states *Q1* and *Q2*, two sets of transitions *transition1* and *transition2* and two names set *names1* and *names2* it applies *buildTransitionFromMoreTransitionsRule1* upon *transition1*.

```

Fixpoint buildTransitionFromMoreAllTransitionsSingleState (Q1: state) (Q2:
state2) (transition1: set (set (name) × DC × (set(state))))
(transition2: set (set (name) × DC × (set(state2))))
(names1 : set name) (names2: set name) :=
match transition1 with
| [] ⇒ []
| a::t ⇒ (buildTransitionFromMoreTransitionsRule1 Q1 Q2 a transition2
          names1 names2)++(buildTransitionFromMoreAllTransitionsSingleState
          Q1 Q2 t transition2 names1 names2)
end.

```

Now that there is a method to verify the set of transitions that contains all transitions of a given state, a procedure to scan all states of both automata is required, in order to retrieve all resulting transitions of the product automaton. This formalization starts with *iterateOverStatesBuildingTransitionsOne*, a function with *Q1* being a state, *Q2* being a set of states (denoting the set of states of \mathcal{A}_2), *transition1* and *transition2* with the same type of *T*, being functions with support $state \rightarrow \text{set}(\text{set}(\text{name}) \times DC \times \text{set}(state))$ and *names1* and *names2* the names set used for comparison (as the rule depicts), *iterateOverStatesBuildingTransitionsOne* will iterate over *Q2* using *buildTransitionFromMoreAllTransitionsSingleState* with each state $a \in Q2$.

```

Fixpoint iterateOverStatesBuildingTransitionsOne (Q1: state) (Q2: set state2)

```

```

(transition1: state → set (set (name) × DC × (set(state))))
(transition2: state2 → set (set (name) × DC × (set(state2))))
(names1 : set name) (names2: set name) :=
match Q2 with
| [] ⇒ []
| a::t ⇒ (buildTransitionFromMoreAllTransitionsSingleState Q1 a
          (transition1 Q1) (transition2 a) names1 names2)++
          (iterateOverStatesBuildingTransitionsOne Q1 t transition1 transition2
          names1 names2)
end.

```

The algorithm also needs to scan the set of states of the first automaton. The function *buildAllTransitionsRule1* uses *iterateOverStatesBuildingTransitionsOne*, only instead of expecting *Q1* as a states, *Q1* now is a set of states that denotes the states of \mathcal{A}_1 , with all the other parameters as defined in *iterateOverStatesBuildingTransitionsOne*.

```

Fixpoint buildAllTransitionsRule1 (Q1: set state) (Q2: set state2)
(transition1: state → set (set (name) × DC × (set(state))))
(transition2: state2 → set (set (name) × DC × (set(state2))))
(names1 : set name) (names2: set name) :=
match Q1 with
| [] ⇒ []
| a::t ⇒ (iterateOverStatesBuildingTransitionsOne a Q2 transition1
          transition2 names1 names2) ++ (buildAllTransitionsRule1 t Q2
          transition1 transition2 names1 names2)
end.

```

All the apparatus to build the transitions rules with respect to the rule (i) as stated in Definition 11 is complete. The next definition is the top-level function used to build this set with data incoming from the automata involved in the process. Therefore *transitionsRule1* with two constraint automata *a1* and *a2* returns the set of transitions regarding the aforementioned rule.

```

Definition transitionsRule1 (a1: constraintAutomata)
(a2: constraintAutomata) := buildAllTransitionsRule1
(ConstraintAutomata.Q a1) (ConstraintAutomata.Q a2)

```

$$\begin{aligned}
& (\textit{ConstraintAutomata.T } a1) (\textit{ConstraintAutomata.T } a2) \\
& (\textit{ConstraintAutomata.N } a1) (\textit{ConstraintAutomata.N } a2).
\end{aligned}$$

The formalization of the first rule is now completed. In order to correctly build the transition relation of the resulting product automaton, a procedure to build transitions according to the second and third rule are also required. The first step towards this formalization is achieved by *intersectionNAndNames* with *tr* being a single transition of type $(\text{set } (\textit{name}) \times \textit{DC} \times \text{set}(\textit{state}))$ and *names2* as the name set of \mathcal{A}_2 , returning *true* if the intersection of *names2* with the name set associated with *tr* is \emptyset and *false* otherwise.

Definition *intersectionNAndNames* (*tr*: $\text{set } (\textit{name}) \times \textit{DC} \times \text{set}(\textit{state})$)
(*names2*: $\text{set } \textit{name}$) :=
if (*set_inter_equiv_dec* (*fst*(*fst*(*tr*))) *names2*) == *nil* *then true else false*.

The next definition comprises the notion of constructing a resulting transition's origin state as denoted by Definition 11. The states as shown by the second rule stands for a transition from a state q_1 to p_1 will have the resulting outcome state as (q_1, q_2) and (p_1, q_2) , $\forall q_2 \in Q_2$ where Q_2 is \mathcal{A}_2 's states set. Therefore, because the product automata is depicted as $\mathcal{A}_1 \bowtie \mathcal{A}_2$, in order to build the resulting transition rule as depicted by Definition 11, for each state $q_1 \in Q_1$ (with Q_1 as the states set of \mathcal{A}_1) that the rule (ii) is being applied to, it needs to recover all states $q_2 \in Q_2$, Q_2 the states set of \mathcal{A}_2 .

The function *createSingleTransition* is conceived as the definition that has the discussed idea behind it. Hence *createSingleTransition* with a state *q1*, a set of transitions *transitions*, a set of states of Q_2 denoted by *q2* and a set of names *names2* that stands for \mathcal{A}_2 's names set, it returns a resulting transition as discussed by rule (ii) in Definition 11 by applying the rule to q_1 and, if the necessary criteria are met, iterating over Q_2 in order to retrieve the states that are part of the built transition.


```

Fixpoint createSingleTransition (q1:state) (transition : (set (name) × DC ×
(set(state)))) (a2States : set state2) (a2Names: set name)
: set (state × state2 × ((set name × DC) × set (state × state2))) :=
match a2States with
| [] ⇒ []
| q2::t ⇒ if (intersectionNAndNames (transition) (a2Names) == true) then
((q1,q2),((fst(transition)), (iterateOverOutboundStatesRule2
(snd(transition)) (q2))))::createSingleTransition q1 transition t a2Names
else createSingleTransition q1 transition t a2Names
end.

```

It is important to note (once again) that transitions of a constraint automaton as denoted by Definition 12 are defined as a function $T: \text{state} \rightarrow \text{set}(\text{set}(\text{name}) \times DC \times \text{set state})$. This means that in order to correctly build the resulting set of states, a way to retrieve all states in set state in order to build the resulting states as (a, q_2) where $a \in \text{set state}$ and $q_2 \in Q_2$. The resulting set of states of the product automaton may preserve the same format (as will be discussed shortly), with the resulting set of transitions as $T: \text{set}(\text{state} \times \text{state2} \times (\text{setname} \times DC \times \text{set}(\text{state} \times \text{state2})))$.

The definition used by *createSingleTransition* in order to build these states is *iterateOverOutboundStatesRule2*, a function that with a set of states $p1$ and a fixed state $q2$ returns pairs of state (a, q_2) , $\forall a \in p1$.

```

Fixpoint iterateOverOutboundStatesRule2 (p1: set state) (q2: state2) :=
match p1 with
| [] ⇒ []
| a::t ⇒ set_add equiv_dec ((a,q2))(iterateOverOutboundStatesRule2 t q2)
end.

```

Because *createSingleTransition* only creates a single transition by means of applying the second derivation rule of product automata's transitions with a single state and only one transition departing from it, a way to build all resulting transitions of a product automata from a single state q_1 by applying this rule with all transitions departing from q_1 is still required. By means of *createSingleTransition*, *createTransitionRule2* is a function that with a state $q1$, a set of transitions *transitions*, a set of state $q2$ and a set of names *names2* calls *createSingleTransition* with every transition $t \in \text{transitions}$, fulfilling

the requirement.

```

Definition createTransitionRule2 (q1:state) : set (set (name) × DC ×
(set(state))) → set state2 → set name
→ set (state × state2 × ((set name × DC) × set (state × state2))) :=
fix rec transitions q2 names2 :=
  match transitions with
  | [] ⇒ []
  | a::t ⇒ (createSingleTransition q1 a q2 names2)++(rec t q2 names2)
end.

```

Then the last procedure needed in the context of creating all transitions for the resulting product automaton by means of the second derivation rule is to iterate through the set of states of A_1 , Q_1 and verifying, for each state $q_1 \in Q_1$ with every transition departing from q_1 if the necessary criteria to build the resulting transitions with q_1 and its transitions are met, is a function that iterates through Q_1 . The definition *createTransitionRule2AllStates* achieves this requirement by iterating over a set of states Q_1 and a set of transitions *transitions*, applying *createTransitionRule2* for each $q_1 \in Q_1$ and with every transition associated with q_1 .

```

Fixpoint createTransitionRule2AllStates (Q1: set state) (transitions: state →
set (set (name) × DC × (set(state)))) (names2: set name) (a2States : set
state2) :=
match Q1 with
| [] ⇒ []
| a::t ⇒ (createTransitionRule2 a (transitions(a)) a2States names2)++
(createTransitionRule2AllStates t transitions names2 a2States)
end.

```

Therefore, *transitionsRule2* is the top level definition that with two constraint automata $a1$ and $a2$ builds all transitions as depicted by rule (ii) in Definition 11, by applying *createTransitionRule2AllStates* as the set of states Q_1 to be iterated being $a1$'s states, the set of transitions being $a1$'s transitions, the name set that is used in the comparison between the transition's set of names and the other automaton involved in the product operation as $a2$'s name set and the set of states that is used to build the resulting states (a, q_2) as discussed earlier is $a2$'s set of states.

```

Definition transitionsRule2 (a1: constraintAutomata)
(a2 : constraintAutomata) := (createTransitionRule2AllStates
  (ConstraintAutomata.Q a1) (ConstraintAutomata.T a1)
  (ConstraintAutomata.N a2) (ConstraintAutomata.Q a2)).

```

Because the third rule in Definition 11 is the symmetric of the second rule, its definition proceeds in quite a similar way, with some different functions in order to adjust the required parameters. The first is to deal with the fact that transitions in Constraint Automata formalized in Coq are $T: \text{state} \rightarrow \text{set}(\text{set}(\text{name}) \times DC \times \text{set state})$ the same way *iterateOverOutboundStatesRule2* solves this question for rule(ii). In order to create states of the resulting transitions as required by rule(iii) *iterateOverOutboundStatesRule3* is defined as a function that with a state $q2$ and a set of states $p1$ returns a set containing pairs of states $(q2, a)$, $\forall a \in p1$ with similar objective as of *iterateOverOutboundStatesRule2*.

```

Fixpoint iterateOverOutboundStatesRule3 (q1: state) (p2: set state2) :=
match p1 with
| [] => []
| a::t => set_add equiv_dec ((q1,a))(iterateOverOutboundStatesRule3 q1 t)
end.

```

Then *createSingleTransitionRule3* is a function that is bound to apply rule (iii) with a single state and a single transition: with a state $q2$, a transition denoted by *transition*, a state set *a1States* and name set *a1Names* (which respectively denotes the state of A_2 in which the transition being used by the rule departs from, the transition itself, the name set of A_1 and A_1 's set of states) returns the resulting transition of the product automata by means of rule (iii) if *transition* meet the requirements.

```

Fixpoint createSingleTransitionRule3 (q2:state) (transition : (set (name) ×
DC × (set(state)))) (a1States : set state) (a1Names: set name)
: set (state × state × ((set name × DC) × set (state × state))) :=
match a1States with
| [] => []
| q1::t => if (intersectionNAndNames2 (transition) (a1Names) == true)
then ((q1,q2),((fst(transition))), (iterateOverOutboundStatesRule3 (q1)

```

```

      (snd(transition))))):createSingleTransitionRule3 q2 transition t a1Names
    else createSingleTransitionRule3 q2 transition t a1Names
  end.

```

The required condition for transitions to be built by means of rule (iii) is formalized by `by` which is a function that with a transition `tr` of \mathcal{A}_2 and `names1` denoting the names set of \mathcal{A}_1 returns `true` if the intersection of the name set bound to the transition `tr` with `names1` equals \emptyset and false otherwise.

```

Definition intersectionNAndNames2 (tr: set (name) × DC × set(state2))
  (names1: set name) :=
  if (set_inter equiv_dec (fst(fst(tr))) names1) == nil then true else false

```

A way to evaluate all transitions of a single state is still needed, given that `createSingleTransitionRule3` only uses a single transition in order to apply rule (iii). The objective of `createTransitionRule3` is to iterate over a set of transitions with a single state in order to verify which are the transitions that satisfies this rule in order to build all the resulting transitions. Hence `createTransitionRule3` is a function that with a state `q2`, a set of transitions depicted by `transitions`, a set of states `q1` and a name set `names1` it returns all transitions that can be built from transitions departing from a single state by means of `createSingleTransitionRule3`.

```

Definition createTransitionRule3 (q2:state2): set (set (name) × DC ×
  (set(state2))) → set state → set name
→ set (state × state2 × ((set name × DC) × set (state × state2))) :=
fix rec transitions q1 names1 :=
  match transitions with
  | [] ⇒ []
  | a::t ⇒ (createSingleTransitionRule3 q2 a q1 names1)++(rec t q1 names1)
  end.

```

With `createTransitionRule3`, it is possible now to use it over the set of states of \mathcal{A}_2 in order to achieve the expected result by the usage of rule (iii). The function `createTransitionRule3AllStates` is a function that with a set of states `Q2`, which denotes \mathcal{A}_2 's set of states, a transition relation as depicted by Definition 12 `transitions`, a name set `names1` and a states set `a1States` returns a set containing all transitions that can be built by applying rule (iii) to every transition of every state in `Q2` by means of

createTransitionRule3.

```

Fixpoint createTransitionRule3AllStates (Q2: set state2) (transitions: state2
→ set (set (name) × DC × (set(state2))))
(names1: set name) (a1States : set state) :=
match Q2 with
| [] ⇒ []
| a::t ⇒ (createTransitionRule3 a (transitions(a)) a1States names1)++
          (createTransitionRule3AllStates t transitions names1 a1States)
end.

```

These definitions culminate on *transitionsRule3*, the top level function that with two constraint automata *a1* and *a2*, calls *createTransitionRule3AllStates* with *a2*'s set of states as the set of states to be iterated, *a2*'s transition relation as the transitions to be used with the states in *a2*'s set of states, *a1*'s name set as the set used for comparison by the rule (as depicted in *createSingleTransitionRule3*) and *a1*'s set of states is needed in order to *createSingleTransitionRule3* to create the resulting transition's states correctly.

```

Definition transitionsRule3 (a1: constraintAutomata)
(a2 : constraintAutomata2) := (createTransitionRule3AllStates
(ConstraintAutomata.Q a2) (ConstraintAutomata.T a2)
(ConstraintAutomata.N a1) (ConstraintAutomata.Q a1)).

```

The set of transitions of the product automata $\mathcal{A}_1 \bowtie \mathcal{A}$ is then produced by *buildTransitionRuleProductAutomaton*, which given two constraint automata *a1* and *a2* applies the three production rules by means of *transitionsRule1*, *transitionsRule2* and *transitionsRule3*.

```

Definition buildTransitionRuleProductAutomaton (a1: constraintAutomata)
(a2: constraintAutomata) :=
(transitionRule1 a1 a2)++(transitionsRule2 a1 a2)++(transitionsRule3 a1 a2).

```

Although the necessary definitions to formalize the definition of Product Automata as proposed by [17] are completed, a way to adapt Product Automata created by means of the presented methods to the definition of Constraint Automata in Definition 12 is required. Because transitions in Definition 12 are $T: state \rightarrow \text{set}(\text{set}(name) \times DC \times \text{set } state)$, where *state* is of type *Type*.

Recall that states in the resulting product automaton have type $(state \times state2)$.

Then, the resulting transition relation for product automata is $T_{A_1 \bowtie A_2} : (\text{state} \times \text{state2}) \rightarrow \text{set}(\text{set}(\text{name}) \times DC \times \text{set}(\text{state} \times \text{state2}))$. In this context, *recoverResultingStatesPA* is a function that with a pair of states *st* and a set of transitions *t* returns the set of transitions that origins in *st*.

```

Fixpoint recoverResultingStatesPA (st: (state × state2))
(t:set (state × state2 × ((set name × DC) × set (state × state2)))):=
match t with
| [] ⇒ []
| a::tx ⇒ if st == fst((a)) then (snd((a))::recoverResultingStatesPA st tx)
        else recoverResultingStatesPA st tx
end.

```

Then the actual transition relation of the resulting product automaton is denoted by *transitionPA*, a function that with a state (*state* × *state*) *s* returns all transitions that origins in *s*, where the set of transitions is achieved by *buildTransitionRuleProductAutomaton*.

```

Definition transitionPA (s: (state × state2)) :=
recoverResultingStatesPA s (buildTransitionRuleProductAutomaton a1 a2).

```

The product automaton is built as a constraint automaton by means of Definition 12, using *CA* as the alias of *constraintAutomata*. Therefore *buildPA* uses *CA* with two constraint automaton *a1* and *a2* in order to build a product automaton based on *resultingStatesSet*, *resultingNameSet*, *transitionPA* and *resultingInitialStatesSet*, which respectively builds the set of states, set of names, the transition relation and the set of initial states of the resulting automaton.

```

Definition buildPA := ConstraintAutomata.CA
(resultingStatesSet a1 a2) (resultingNameSet a1 a2) (transitionPA)
(resultingInitialStatesSet a1 a2).

```

Because product automata are also constraint automata, all tooling available for constraint automata are also available for product automata. This means that runs defined for constraint automata can also be applied to product automata. The formalization of product automata as hereby discussed enables the compositional construction of complex automata from simpler ones, as complex Reo connectors are built out of canonical connectors, enabling compositional reasoning on these connectors by means of product

automata.

Therefore the formalization of product automata by means of *buildPA* corresponds to the definition of Product Automata as depicted by Definition 11. Let \mathcal{A}_1 and \mathcal{A}_2 constraint automata following Definition 2. Hence the set of states of $\mathcal{A}_1 \bowtie \mathcal{A}_2$, $Q_1 \times Q_2$ as the set of states is equivalent to *resultingStatesSet a1 a2* by Lemma 25, $Names_1 \cup Names_2$ is equivalent to *resultingNameSet a1 a2* by Lemma 26, \rightarrow as the transition relation of $\mathcal{A}_1 \bowtie \mathcal{A}_2$ is equivalent to *transitionPA* (except for the details discussed shortly after Definition 12 regarding its return), which is the transition relation of product automata in Coq. Such equivalence holds from Lemma 52. The resulting set of initial states of $\mathcal{A}_1 \bowtie \mathcal{A}_2$ is equivalent to *resultingInitialStatesSet a1 a2* by Lemma 27.

Chapter 6

Usage Examples

The current chapter provides usage examples of the formalisms hereby implemented, from the user input code to the corresponding Coq code in order to prove properties about the formalized connector. Automatic code extraction of these examples are also introduced. Full code regarding the examples shown here can be found in Appendix C and in Sections B.1 and B.2.

6.1 A two-bounded FIFO connector

Reo advocates the idea of compositionally building more complex channels out of simpler ones. This simple (yet representative) example consists of formalizing a two-bounded FIFO connector by composing two one-bounded FIFO channels as introduced in Chapter 4. Figure 6.1 shows the graphical model denoting a two-bounded FIFO.

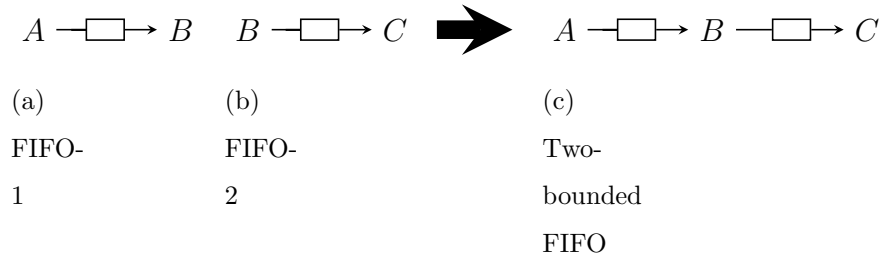


Figure 6.1: A graphical representation of the resulting two-bounded FIFO connector

This connector can be obtained by composing two FIFO connectors as discussed in Table 4.2, using the product operation hereby formalized. Therefore the idea is to join the sink end of the first FIFO with the source node of the second connector, obtaining a FIFO connector with capacity of two. Following this idea, both connectors must be

defined at first. Nodes are formalized as a single inductive type containing all nodes:

Inductive *fifoPorts* := *A* | *B* | *C*.

The states depicting states of the corresponding automaton also follows Table 4.2. Therefore, the following definition declares the automaton's states. For simplicity reasons, all states are defined within the same inductive definition. States *q0a*, *p0a* and *p1a* denotes the first connector being empty, with *0* in it and *1* in it respectively. The same holds for the second connector for the states ending with *b*

Inductive *fifoStates* := *q0a* | *p0a* | *p1a* | *q0b* | *p0b* | *p1b*.

From the first line of code within Definition 12, equality relation procedures must be provided for *state*, *name* and *option data* type by means of the Class *EqDec*¹. CACoq comes packed with a program that generates this decision procedure which results on the *EqDec* instances defined in Section B.1.

Therefore the first constraint automaton can be formalized as follows.

Definition *oneBoundedFIFOCA* := {
 ConstraintAutomata.Q := [*q0a*; *p0a*; *p1a*];
 ConstraintAutomata.N := [*A*; *B*];
 ConstraintAutomata.T := *oneBoundedFIFOrel*;
 ConstraintAutomata.Q0 := [*q0a*]
 }.

By simplicity reasons, let the data domain of both connectors be $[0, 1]$. Therefore the transition relation of the first automaton is defined as *oneBoundedFIFOrel* which is

Definition *oneBoundedFIFOrel* (*s*:*fifoStates*) :=
 match *s* with
 | *q0a* ⇒ [([*A*], (*ConstraintAutomata.dc A* (*Some* 0)), [*p0a*]) ;
 ([*A*], (*ConstraintAutomata.dc A* (*Some* 1)), [*p1a*])]
 | *p0a* ⇒ [([*B*], (*ConstraintAutomata.dc B* (*Some* 0)), [*q0a*])]
 | *p1a* ⇒ [([*B*], (*ConstraintAutomata.dc B* (*Some* 1)), [*q0a*])]
 | *q0b* | *p0b* | *p1b* ⇒ []
 end.

The second FIFO automaton may be formalized similarly. Therefore, *oneBoundedFIFOCA2* denoting this automaton is

¹<https://coq.inria.fr/library/Coq.Classes.EquivDec.html>

Definition *oneBoundedFIFOCA2* := {
 ConstraintAutomata.Q := [*q0b*; *p0b*; *p1b*];
 ConstraintAutomata.N := [*B*; *C*];
 ConstraintAutomata.T := *oneBoundedFIFOrel2*;
 ConstraintAutomata.Q0 := [*q0b*]
 }.

Hence *oneBoundedFIFOCA2*'s transition relation, *oneBoundedFIFOrel2* is defined in quite a similar way compared to *oneBoundedFIFOrel*

Definition *oneBoundedFIFOrel2* (*s:fifoStates*) :=
 match *s* with
 | *q0b* ⇒ [([*B*], (*ConstraintAutomata.dc B* (*Some* 0)), [*p0b*]) ;
 ([*B*], (*ConstraintAutomata.dc B* (*Some* 1)), [*p1b*])]
 | *p0b* ⇒ [([*C*], (*ConstraintAutomata.dc C* (*Some* 0)), [*q0b*])]
 | *p1b* ⇒ [([*C*], (*ConstraintAutomata.dc C* (*Some* 1)), [*q0b*])]
 | *q0a* | *p0a* | *p1a* ⇒ []
 end.

The resulting product automaton is then defined by means of *buildPA* definition which encapsulates the whole product operation defined in Section 5.2.

Definition *twoBoundedFifo* := *ProductAutomata.buildPA oneBoundedFIFOCA oneBoundedFIFOCA2*.

It is important to note that both *oneBoundedFIFOCA* and *oneBoundedFIFOCA2* remain valid constraint automata denoting one-bounded FIFO by themselves. This means that they are not consumed in the process of producing their product automata and therefore proofs and runs may be carried on them as desired.

Therefore *twoBoundedFifo* is a constraint automaton where its behavior encapsulates the sequentiality of two one-bounded FIFO linked together. Therefore properties about this newly produced automaton can be formalized. Let $\theta \in TDS^{Names}$ as the data flow given to this constraint automaton be denoted by the following ports, denoting a sequential data flow from source node *A* to sink node *C*. Let definitions *portAF*, *portBF* and *portCF* denote respectively *TDS* for port names *A*, *B* and *C*.

Therefore *realports* is a set grouping the *TDS* for *A*, *B* and *C*, being the *TDS* $\theta \in TDS^{Names}$ as input for the automaton. For both this and the example denoted in

Section B.2, code regarding the ports composing input for the corresponding constraint automaton are found respectively in Appendices B.1 and B.2.

Definition *realports* := $[portAF; portBF; portCF]$.

A possible run can be defined as *ru6*. Therefore *ru6* denotes a run with $\theta \in TDS^{Names}$ where $\theta = realports$ and with a total number of steps 6.

Definition *ru6* := `Eval` compute in *ConstraintAutomata.run* *twoBoundedFifo* *realports* 6.

Therefore *ru6* describes the behavior mentioned, where *ru6* unfolds to the following execution trace.

$[(q0a, q0b)]; [(p0a, q0b)]; [(q0a, p0b)]; [(q0a, q0b)]; [(p0a, q0b)]; [(q0a, p0b)]; [(q0a, q0b)]; [(p1a, q0b)]$.

Therefore properties about the automaton and a given data flow can be proven, for example, let the following Lemma denote that *ru6* describes a flow which is accepted by *twoBoundedFifo*: From Definition 10, an accepting run can be interpreted as a run where there is always a possible transition to be fired from a given state of the automaton. Therefore there will not be a rupture in the run trace (meaning that no transitions could have been fired).

Lemma *ru6Accept* : $\neg (In [] (ru6))$.

Proof.

`intros. unfold not. unfold ru6. simpl. intros. repeat
(destruct H ; inversion H). Defined.`

For another data flow, let the following *portA2* a TDS for *A*, *portB2* a TDS for *B* and *portC2* a TDS for *C*. In this data flow there is a moment that the buffer is full and now the run has 8 steps long.

Definition *ru62* := `Eval` compute in *ConstraintAutomata.run* *twoBoundedFifo* $[portA2; portB2; portC2]$ 8.

Therefore *ru62* is a definition comprising this run, where its execution trace is as follows.

$[(q0a, q0b)]; [(p0a, q0b)]; [(q0a, p0b)]; [(p0a, p0b)]; [(p0a, q0b)]; [(q0a, p0b)]; [(q0a, q0b)]; [(p1a, q0b)]; [(q0a, p1b)]; [(q0a, q0b)]$.

Where the Lemma *ru62Accept* also proves that *ru62* as a run for *twoBoundedFifo* is also an accepting run.

Lemma *ru62Accept* : $\neg (In [] (ru62))$.

Proof.

intros. unfold *not*. unfold *ru62*. simpl. intros.

repeat (destruct *H* ; inversion *H*). **Defined.**

One might also want to prove specific properties about the given data flow. Suppose that *ru62* must have at least a point in which the buffer is full, denoted by state (p0a;p0b). The following Lemma formalizes this idea.

Lemma *fullFifoWith0* : $In [(p0a,p0b)] ru62$.

Proof. simpl; auto. **Defined.**

6.2 Alternating Bit Protocol

The following example implements the Alternating Bit Protocol by recovering the example depicted in Figure 4.4. Because the protocol's behavior depends on the composition on smaller functionalities, the connectors composing it must be formalized. The definition of each connector is depicted in Table 6.2.

Each constraint automaton in Coq depicted in Table 6.2 formalizes the corresponding Reo behavior as expressed in Table 4.2. The resulting automaton can be obtained with the product of all automata in Table 6.2. Therefore, *filterLossySyncProduct* holds the product automaton obtained from the product of automata regarding port names *A*, *B* and *C* respectively denoted by definitions *transformCA* and *lossySyncCA*, while *filterLossySyncProduct2* is the resulting product of *lossySync2CA* and *filterCA* as the automata which has as port names *D*, *E* and *F*.

Definition *filterLossySyncProduct* := *ProductAutomata.buildPA transformCA lossySyncCA*.

Definition *filterLossySyncProduct2* := *ProductAutomata.buildPA lossySync2CA filterCA*.

The product of both *filterLossySyncProduct* and *filterLossySyncProduct2* as the automaton which represents the behavior of the Reo connector portrayed in Figure 4.2 is *resultingPaAbp*, which results in the constraint automaton shown in Figure 4.4.

Definition *resultingPaAbp* := *ProductAutomata.buildPA filterLossySyncProduct filterLossySyncProduct2*.

Now it is possible to reason about the Alternating Bit Protocol modelled by means of the Reo connector in Figure 4.2: proofs about the automaton as well as proofs regarding data input for the automaton can be formalized. An accepting stream can be formalized as follows.

Definition *steamAcce* := **Eval** compute in *ConstraintAutomata.run resultingPaAbp tdsRun1* 3.

Where *streamAccept8* is a Lemma regarding that a run in *resultingPaAbp* with *tdsRun1* is an accepting run with 3 steps.

Lemma *streamAccept8* : $\neg \text{In } ([\])$ (*steamAcce*).

Proof.

unfold *not*. unfold *steamAcce*. simpl. intros.

repeat (destruct *H* ; inversion *H*). **Defined.**

In a similar way, *onlyDHasData* depicts a run with an accepting input named *input2*, where only *D* has data for 5 steps. The Lemma *onlyDwithData* then stands for the proof that this run is an accepting one.

Definition *onlyDHasData* := **Eval** compute in *ConstraintAutomata.run resultingPaAbp input2* 5.

Lemma *onlyDwithData* : $\neg \text{In } ([\])$ (*onlyDHasData*).

Proof. unfold *not*. unfold *steamAcce*. simpl. intros.

repeat (destruct *H* ; inversion *H*). **Defined.**

A proof of a non-accepting run can also be carried on: *runAbpCAnotAc* is an example of a proof that a run with an input which execution trace is denoted by *runAbpCAnotAccept* as the result of a not accepting run with 6 iterations.

Definition *runAbpCAnotAccept* := **Eval** compute in *ConstraintAutomata.run resultingPaAbp input3* 6. **Lemma** *runAbpCAnotAc* : $\text{In } [\]$ *runAbpCAnotAccept*.

Proof. simpl;auto. **Defined.**

The developed framework also lets users to define proofs about their user-defined constraint automata as they wish, instead of also verifying data flow on constraint automata denoting Reo connectors. Although not shown here, the user may also extract the certified code in Coq to languages such as Scheme, Haskell or OCaml as discussed in Section 3.2.

Reo	Constraint automaton
$A \twoheadrightarrow B$	<p>Definition <i>transformCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [<i>q0ls</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [<i>A;B</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>transform-</i> <i>CaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [<i>q0ls</i>]</p> <p style="padding-left: 40px;"> }</p>
$B \dashrightarrow C$	<p>Definition <i>lossySyncCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [<i>q0</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [<i>B;C</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>lossySync-</i> <i>CaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [<i>q0</i>]</p> <p style="padding-left: 40px;"> }.</p>
$D \dashrightarrow E$	<p>Definition <i>lossySync2CA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [<i>q02</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [<i>D;E</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> :=</p> <p style="padding-left: 40px;"><i>lossySync2CaBehavior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [<i>q02</i>]</p> <p style="padding-left: 40px;"> }.</p>
$E \rightsquigarrow F$	<p>Definition <i>filterCA</i> := { </p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q</i> := [<i>q03</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.N</i> := [<i>E;F</i>];</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.T</i> := <i>filterCaBehav-</i> <i>ior</i>;</p> <p style="padding-left: 40px;"><i>ConstraintAutomata.Q0</i> := [<i>q03</i>]</p> <p style="padding-left: 40px;"> }.</p>

Table 6.1: Formalization of constraint automata in Coq regarding the Reo connector depicted in Figure 4.2

Chapter 7

Conclusions and Further Work

As software development shifts to more reusable, independent components that interact together towards a common objective, which is to compose a new system with already existing piece of software, new ways to deal with the need to orchestrate how these component interacts are needed.

Reo comes as a graphical language which takes advantage of distributed computing characteristics such as synchronous and asynchronous remote function calls, message passing and other common characteristics. Constraint Automata are Reo’s most basic formal semantics which enable formal reasoning on Reo connectors (by means of Constraint Automata).

Many applications and systems nowadays need a guarantee that certain requirements are met, especially critical systems. Tests are not enough to guarantee that a certain software meet necessary requirements. Formal methods provide ways to mathematically ensure that these requirements are met.

The formalization of a theory such as Constraint Automata in an environment like Coq enables the usage of an already consolidated system to verify and prove properties of Reo connectors by means of Constraint Automata. Coq specifically lets their users extract certified code to external languages as discussed in Section 3.2.

The work hereby presented culminates on CACoq, a Coq library that enables users to formalize constraint automata based on canonical constraint automata defined by [17], by means of the product operation, as well as user defined constraint automata. These user defined constraint automata may also be used in order to compose more complex automata with canonical ones or other user defined constraint automata.

Therefore formalisms defined in CACoq provide means to reason about constraint automata, their behavior and how input $\theta \in TDS^{\mathcal{Names}}$ are accepted (or rejected) by some automaton \mathcal{A} . Proofs about instances of such formalisms may also be carried on at the user's will and transformed in certified code.

Further work includes providing a Graphical User Interface to model Reo systems and making it transparent to the user the usage of Coq. A natural extension is the implementation of Timed Constraint Automata [18].

Appendix A

An example of certified Coq code

```
1 ;; This extracted scheme code relies on some additional macros
2 ;; available at http://www.pps.univ-paris-diderot.fr/~letouzey/scheme
3 (load "macros_extr.scm")
4
5
6 (define add (lambdas (n m) (match n
7                               ((O) m)
8                               ((S p) '(S ,( @ add p m))))))
9
10 (define weekdays_rect (lambdas (f f0 f1 f2 f3 f4 f5 w)
11   (match w
12     ((Monday) f)
13     ((Tuesday) f0)
14     ((Wednesday) f1)
15     ((Thursday) f2)
16     ((Friday) f3)
17     ((Saturday) f4)
18     ((Sunday) f5))))
19
20 (define weekdays_rec (lambdas (f f0 f1 f2 f3 f4 f5 w)
21   (match w
22     ((Monday) f)
23     ((Tuesday) f0)
```

```

24      ((Wednesday) f1)
25      ((Thursday) f2)
26      ((Friday) f3)
27      ((Saturday) f4)
28      ((Sunday) f5))))
29
30 (define nextDay (lambda (day)
31   (match day
32     ((Monday) '(Tuesday))
33     ((Tuesday) '(Wednesday))
34     ((Wednesday) '(Thursday))
35     ((Thursday) '(Friday))
36     ((Friday) '(Saturday))
37     ((Saturday) '(Sunday))
38     ((Sunday) '(Monday)))))
39
40 (define plus2 (lambda (n) (@ add n '(S ,'(S ,'(O))))))

```

Listing A.1: Certified code for the introduced example in Section 3.2

Appendix B

Code for Usage Examples

B.1 Two-Bounded FIFO Code

Require Import CaMain.

Inductive fifoStates := q0a | p0a | p1a | q0b | p0b | p1b.

Inductive fifoPorts := A | B | C | D.

Instance fifoStatesEq : EqDec fifoStates eq :=

{equiv_dec x y :=
match x, y with
| q0a, q0a ⇒ in_left
| p0a, p0a ⇒ in_left
| p1a, p1a ⇒ in_left
| q0b, q0b ⇒ in_left
| p0b, p0b ⇒ in_left
| p1b, p1b ⇒ in_left
| q0a, p0a ⇒ in_right
| q0a, p1a ⇒ in_right
| q0a, q0b ⇒ in_right
| q0a, p0b ⇒ in_right
| q0a, p1b ⇒ in_right
| p0a, q0a ⇒ in_right
| p0a, p1a ⇒ in_right
| p0a, q0b ⇒ in_right

```

| p0a,p0b ⇒ in_right
| p0a,p1b ⇒ in_right
| p1a,q0a ⇒ in_right
| p1a,p0a ⇒ in_right
| p1a,q0b ⇒ in_right
| p1a,p0b ⇒ in_right
| p1a,p1b ⇒ in_right
| q0b,q0a ⇒ in_right
| q0b,p0a ⇒ in_right
| q0b,p1a ⇒ in_right
| q0b,p0b ⇒ in_right
| q0b,p1b ⇒ in_right
| p0b,q0a ⇒ in_right
| p0b,p0a ⇒ in_right
| p0b,p1a ⇒ in_right
| p0b,q0b ⇒ in_right
| p0b,p1b ⇒ in_right
| p1b,q0a ⇒ in_right
| p1b,p0a ⇒ in_right
| p1b,p1a ⇒ in_right
| p1b,q0b ⇒ in_right
| p1b,p0b ⇒ in_right
end

```

}.

Proof.

all: congruence.

Defined.

Instance *fifoPortsEq* : *EqDec* *fifoPorts* *eq* :=

```

{equiv_dec x y :=
  match x, y with
  | A,A ⇒ in_left
  | B,B ⇒ in_left

```

```

|  $C, C \Rightarrow in\_left$ 
|  $D, D \Rightarrow in\_left$ 
|  $A, B \Rightarrow in\_right$ 
|  $A, C \Rightarrow in\_right$ 
|  $A, D \Rightarrow in\_right$ 
|  $B, A \Rightarrow in\_right$ 
|  $B, C \Rightarrow in\_right$ 
|  $B, D \Rightarrow in\_right$ 
|  $C, A \Rightarrow in\_right$ 
|  $C, B \Rightarrow in\_right$ 
|  $C, D \Rightarrow in\_right$ 
|  $D, A \Rightarrow in\_right$ 
|  $D, B \Rightarrow in\_right$ 
|  $D, C \Rightarrow in\_right$ 
end

```

```

}.

```

Proof.

all:congruence.

Defined.

Definition *dataAssignmentA* $n :=$

```

match  $n$  with
| 0  $\Rightarrow Some$  0
| 1  $\Rightarrow Some$  0
| 2  $\Rightarrow Some$  1
|  $S\ n \Rightarrow Some$  (0)
end.

```

Definition *dataAssignmentB* $n :=$

```

match  $n$  with
| 0  $\Rightarrow Some$  0
| 1  $\Rightarrow Some$  (0)
| 2  $\Rightarrow Some$  1

```

| $S\ n \Rightarrow \text{Some } 1$
end.

Definition *timeStampFIFOA*($n:\text{nat}$) : $Q\text{Arith_base}.Q :=$

match n with
| 0 $\Rightarrow 1\#1$
| 1 $\Rightarrow 4\#1$
| 2 $\Rightarrow 7\#1$
| 3 $\Rightarrow 10\#1$
| 4 $\Rightarrow 13\#1$
| 5 $\Rightarrow 15\#1$
| $S\ n \Rightarrow Z.\text{of_N } (N.\text{of_nat}(S\ n)) + 1\#1$
end.

Definition *timeStampFIFOB* ($n:\text{nat}$) : $Q\text{Arith_base}.Q :=$

match n with
| 0 $\Rightarrow 2\#1$
| 1 $\Rightarrow 5\#1$
| 2 $\Rightarrow 8\#1$
| 3 $\Rightarrow 11\#1$
| 4 $\Rightarrow 14\#1$
| 5 $\Rightarrow 17\#1$
| $S\ n \Rightarrow Z.\text{of_N } (N.\text{of_nat}(S\ n)) + 1\#1$
end.

Definition *dataAssignmentC* $n :=$

match n with
| 0 $\Rightarrow \text{Some } 0$
| 1 $\Rightarrow \text{Some } (0)$
| 2 $\Rightarrow \text{Some } 1$
| $S\ n \Rightarrow \text{Some } 0$
end.

Definition *timeStampFIFOC*($n:\text{nat}$) : $Q\text{Arith_base}.Q :=$

match n with
| 0 $\Rightarrow 3\#1$

```

| 1 ⇒ 6#1
| 2 ⇒ 9#1
| 3 ⇒ 12#1
| 4 ⇒ 15#1
| 5 ⇒ 18#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Lemma *timeStampTestFIFOAHolds* : $\forall\ n,$ $Qle\ (timeStampFIFOA\ n)\ (timeStampFIFOA\ (S\ n))$.

Proof.

```

induction n.
+ unfold timeStampTest. cbv. intros. inversion H.
+ unfold timeStampTest.   Admitted.

```

Lemma *timeStampTestFIFOBHolds* : $\forall\ n,$
 $Qle\ (timeStampFIFOB\ n)\ (timeStampFIFOB\ (S\ n))$.

Proof.

Admitted.

Lemma *timeStampTestFIFOC Holds* : $\forall\ n,$
 $Qle\ (timeStampFIFOC\ n)\ (timeStampFIFOC\ (S\ n))$.

Proof.

Admitted.

Definition *portAF* := { |
 ConstraintAutomata.id := *A*;
 ConstraintAutomata.dataAssignment := *dataAssignmentA*;
 ConstraintAutomata.timeStamp := *timeStampFIFOA*;
 ConstraintAutomata.portCond := *timeStampTestFIFOAHolds*;
 ConstraintAutomata.index := 0 | }.

Definition *portBF* := { |
 ConstraintAutomata.id := *B*;
 ConstraintAutomata.dataAssignment := *dataAssignmentB*;
 ConstraintAutomata.timeStamp := *timeStampFIFOB*;

ConstraintAutomata.portCond := *timeStampTestFIFOBHolds*;
ConstraintAutomata.index := 0 |}.

Definition *portCF* := {
ConstraintAutomata.id := *C*;
ConstraintAutomata.dataAssignment := *dataAssignmentC*;
ConstraintAutomata.timeStamp := *timeStampFIFO*;
ConstraintAutomata.portCond := *timeStampTestFIFOCHolds*;
ConstraintAutomata.index := 0 |}.

Definition *realports* := [*portAF*; *portBF*; *portCF*].

Definition *oneBoundedFIFOrel* (*s*:*fifoStates*)
:=
match s with
| *q0a* ⇒ [[*A*], (*ConstraintAutomata.dc A (Some 0)*), [*p0a*]] ;
 ([*A*], (*ConstraintAutomata.dc A (Some 1)*), [*p1a*])]
| *p0a* ⇒ [[*B*], (*ConstraintAutomata.dc B (Some 0)*), [*q0a*]]
| *p1a* ⇒ [[*B*], (*ConstraintAutomata.dc B (Some 1)*), [*q0a*]]
| *q0b* | *p0b* | *p1b* ⇒ []
end.

Definition *oneBoundedFIFOCA*:= {
ConstraintAutomata.Q := [*q0a*; *p0a*; *p1a*];
ConstraintAutomata.N := [*A*; *B*];
ConstraintAutomata.T := *oneBoundedFIFOrel*;
ConstraintAutomata.Q0 := [*q0a*]
|}.

Eval compute in *ConstraintAutomata.run oneBoundedFIFOCA realports* 10.

Definition *oneBoundedFIFOrel2* (*s*:*fifoStates*)
:=
match s with
| *q0b* ⇒ [[*B*], (*ConstraintAutomata.dc B (Some 0)*), [*p0b*]] ;
 ([*B*], (*ConstraintAutomata.dc B (Some 1)*), [*p1b*])]
| *p0b* ⇒ [[*C*], (*ConstraintAutomata.dc C (Some 0)*), [*q0b*]]


```

|  $p1b \Rightarrow [[C], (ConstraintAutomata.dc\ C\ (Some\ 1)), [q0b]]$ 
|  $q0a \mid p0a \mid p1a \Rightarrow []$ 
end.

```

```

Definition oneBoundedFIFOCA2 := {
  ConstraintAutomata.Q := [q0b;p0b;p1b];
  ConstraintAutomata.N := [B;C];
  ConstraintAutomata.T := oneBoundedFIFOrel2;
  ConstraintAutomata.Q0 := [q0b]
}.

```

```

Definition twoBoundedFifo := ProductAutomata.buildPA oneBoundedFIFOCA oneBound-
edFIFOCA2.

```

```

Eval compute in ConstraintAutomata.T twoBoundedFifo (p0a, q0b).
Eval compute in ConstraintAutomata.Q twoBoundedFifo.
Eval compute in ConstraintAutomata.xamboca2 twoBoundedFifo realports 5.
Eval compute in ConstraintAutomata.run twoBoundedFifo realports 6.

```

```

Definition ru6 := Eval compute in ConstraintAutomata.run twoBoundedFifo
realports 6.

```

```

Lemma ru6Accept :  $\neg (In\ ([]) (ru6))$ .

```

```

Proof.

```

```

intros. unfold not. unfold ru6. simpl. intros. repeat (destruct H ;
inversion H). Defined.

```

```

Definition dataAssignmentA2 n :=
match n with
| 0  $\Rightarrow Some\ 0$ 
| 1  $\Rightarrow Some\ 0$ 
| 2  $\Rightarrow Some\ 1$ 
|  $S\ n \Rightarrow Some\ (0)$ 
end.

```

```

Definition dataAssignmentB2 n :=
match n with
| 0  $\Rightarrow Some\ 0$ 

```

| 1 \Rightarrow *Some* (0)

| 2 \Rightarrow *Some* 1

| *S* *n* \Rightarrow *Some* 1

end.

Definition *timeStampFIFOA2*(*n:nat*) : *QArith_base.Q* :=

match *n* with

| 0 \Rightarrow 1#1

| 1 \Rightarrow 3#1

| 2 \Rightarrow 7#1

| 3 \Rightarrow 10#1

| 4 \Rightarrow 13#1

| 5 \Rightarrow 15#1

| *S* *n* \Rightarrow *Z.of_N* (*N.of_nat*(*S* *n*)) + 1#1

end.

Definition *timeStampFIFOB2* (*n:nat*) : *QArith_base.Q* :=

match *n* with

| 0 \Rightarrow 2#1

| 1 \Rightarrow 5#1

| 2 \Rightarrow 8#1

| 3 \Rightarrow 11#1

| 4 \Rightarrow 14#1

| 5 \Rightarrow 17#1

| *S* *n* \Rightarrow *Z.of_N* (*N.of_nat*(*S* *n*)) + 1#1

end.

Definition *dataAssignmentC2* *n* :=

match *n* with

| 0 \Rightarrow *Some* 0

| 1 \Rightarrow *Some* (0)

| 2 \Rightarrow *Some* 1

| *S* *n* \Rightarrow *Some* 0

end.

Definition *timeStampFIFOC2*($n:nat$) : *QArith_base.Q* :=

match n with

| 0 \Rightarrow 4#1

| 1 \Rightarrow 6#1

| 2 \Rightarrow 9#1

| 3 \Rightarrow 12#1

| 4 \Rightarrow 15#1

| 5 \Rightarrow 18#1

| $S\ n \Rightarrow Z.of_N\ (N.of_nat(S\ n)) + 1\#1$

end.

Lemma *timeStampTestFIFOA2Holds* : $\forall\ n, Qle\ (timeStampFIFOA2\ n)\ (timeStampFIFOA2\ (S\ n))$.

Proof.

induction n .

+ unfold *timeStampTest*. cbv. intros. inversion H .

+ unfold *timeStampTest*.

Admitted.

Lemma *timeStampTestFIFOB2Holds* : $\forall\ n,$

$Qle\ (timeStampFIFOB2\ n)\ (timeStampFIFOB2\ (S\ n))$.

Proof.

Admitted.

Lemma *timeStampTestFIFOC2Holds* : $\forall\ n,$

$Qle\ (timeStampFIFOC2\ n)\ (timeStampFIFOC2\ (S\ n))$.

Proof.

Admitted.

Definition *portA2* := { |

ConstraintAutomata.id := A ;

ConstraintAutomata.dataAssignment := *dataAssignmentA2*;

ConstraintAutomata.timeStamp := *timeStampFIFOA2*;

ConstraintAutomata.portCond := *timeStampTestFIFOA2Holds*;

ConstraintAutomata.index := 0 |}.

Definition *portB2* := { |
 ConstraintAutomata.id := *B*;
 ConstraintAutomata.dataAssignment := *dataAssignmentB2*;
 ConstraintAutomata.timeStamp := *timeStampFIFOB2*;
 ConstraintAutomata.portCond := *timeStampTestFIFOB2Holds*;
 ConstraintAutomata.index := 0 |}.

Definition *portC2* := { |
 ConstraintAutomata.id := *C*;
 ConstraintAutomata.dataAssignment := *dataAssignmentC2*;
 ConstraintAutomata.timeStamp := *timeStampFIFOC2*;
 ConstraintAutomata.portCond := *timeStampTestFIFOC2Holds*;
 ConstraintAutomata.index := 0 |}.

Eval compute in *ConstraintAutomata.run twoBoundedFifo* [*portA2*; *portB2*; *portC2*]

8.

Definition *ru62* := **Eval** compute in *ConstraintAutomata.run twoBoundedFifo* [*portA2*; *portB2*; *portC2*] 8.

Eval compute in *ConstraintAutomata.xamboca2 twoBoundedFifo* [*portA2*; *portB2*; *portC2*]

8.

Lemma *ru62Accept* : $\neg (In \ (\[]) \ (ru62))$.

Proof.

intros. unfold *not*. unfold *ru62*. simpl. intros. repeat (destruct *H* ;
 inversion *H*). **Defined.**

B.2 Alternating Bit Protocol

Inductive *transformState* := *q0ls*.

Instance *transformStateEq* : *EqDec transformState eq* :=

{*equiv_dec* *x y* :=
 match *x, y* with
 | *q0ls, q0ls* \Rightarrow *in_left*

end

}.

Proof.

all: congruence.

Defined.

Inductive *abpStates* := *s0* | *s1* | *r0* | *r1*.

Inductive *abpPorts* := *A* | *B* | *C* | *D* | *E* | *F*.

Instance *abpStatesEq* : *EqDec* *abpStates* *eq* :=

```
{equiv_dec x y :=
  match x, y with
  | s0,s0 => in_left
  | s1,s1 => in_left
  | r0,r0 => in_left
  | r1,r1 => in_left
  | s0,s1 => in_right
  | s0,r0 => in_right
  | s0,r1 => in_right
  | s1,s0 => in_right
  | s1,r0 => in_right
  | s1,r1 => in_right
  | r0,s0 => in_right
  | r0,s1 => in_right
  | r0,r1 => in_right
  | r1,s0 => in_right
  | r1,s1 => in_right
  | r1,r0 => in_right
end
```

}.

Proof.

all: congruence.

Defined.

Instance *abpPortsEq* : *EqDec* *abpPorts* *eq* :=

```
{equiv_dec x y :=
```

```

match x, y with
| A,A ⇒ in_left
| B,B ⇒ in_left
| C,C ⇒ in_left
| D,D ⇒ in_left
| E,E ⇒ in_left
| F,F ⇒ in_left
| A,B ⇒ in_right
| B,A ⇒ in_right
| F,A | F,B | B,F | A,F ⇒ in_right
| D,C | C, D | D,E | E,D ⇒ in_right
| A,C | C, A | B, C | C, B ⇒ in_right
| A,D | A, E | E,A | D,A ⇒ in_right
| B,D | B, E | E,B | D,B ⇒ in_right
| D,F | F,D | F, E | E,F | C,F | F,C | C, E | E,C ⇒ in_right
end

}.

```

Proof.

all:congruence.

Defined.

Definition *dataAssignmentA* *n* :=

```

match n with
| 0 ⇒ Some 0
| 1 ⇒ Some 0
| 2 ⇒ Some 1
| S n ⇒ Some (1)
end.

```

Definition *dataAssignmentB* *n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2

```

```

|  $S\ n \Rightarrow \text{Some } 2$ 
end.

```

Definition $\text{dataAssignmentC } n :=$

```

match  $n$  with
| 0  $\Rightarrow \text{Some } 1$ 
| 1  $\Rightarrow \text{Some } (1)$ 
| 2  $\Rightarrow \text{Some } 2$ 
|  $S\ n \Rightarrow \text{Some } 2$ 
end.

```

Definition $\text{dataAssignmentD } n :=$

```

match  $n$  with
| 0  $\Rightarrow \text{Some } 1$ 
| 1  $\Rightarrow \text{Some } (1)$ 
| 2  $\Rightarrow \text{Some } 2$ 
|  $S\ n \Rightarrow \text{Some } 2$ 
end.

```

Definition $\text{dataAssignmentE } n :=$

```

match  $n$  with
| 0  $\Rightarrow \text{Some } 1$ 
| 1  $\Rightarrow \text{Some } (1)$ 
| 2  $\Rightarrow \text{Some } 2$ 
|  $S\ n \Rightarrow \text{Some } 2$ 
end.

```

Definition $\text{dataAssignmentF } n :=$

```

match  $n$  with
| 0  $\Rightarrow \text{Some } 1$ 
| 1  $\Rightarrow \text{Some } (1)$ 
| 2  $\Rightarrow \text{Some } 2$ 
|  $S\ n \Rightarrow \text{Some } 2$ 
end.

```

Definition $\text{timeStampABPA}(n:\text{nat}) : \text{QArith_base.Q} :=$

```

match  $n$  with
| 0  $\Rightarrow$  2#1
| 1  $\Rightarrow$  5#1
| 2  $\Rightarrow$  8#1
| 3  $\Rightarrow$  11#1
| 4  $\Rightarrow$  14#1
| 5  $\Rightarrow$  17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPB* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0  $\Rightarrow$  2#1
| 1  $\Rightarrow$  5#1
| 2  $\Rightarrow$  8#1
| 3  $\Rightarrow$  11#1
| 4  $\Rightarrow$  14#1
| 5  $\Rightarrow$  17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPC* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0  $\Rightarrow$  2#1
| 1  $\Rightarrow$  5#1
| 2  $\Rightarrow$  8#1
| 3  $\Rightarrow$  11#1
| 4  $\Rightarrow$  14#1
| 5  $\Rightarrow$  17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPD* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0  $\Rightarrow$  2#1

```



```

| 1 ⇒ 5#1
| 2 ⇒ 8#1
| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPE* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0 ⇒ 2#1
| 1 ⇒ 5#1
| 2 ⇒ 8#1
| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPF* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0 ⇒ 2#1
| 1 ⇒ 5#1
| 2 ⇒ 8#1
| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Lemma *timeStampTestABPAHolds* : $\forall\ n, Qle\ (timeStampABPA\ n)\ (timeStampABPA\ (S\ n))$.

Proof.

Admitted.

Lemma *timeStampTestABPBHolds* : $\forall\ n,$

$Qle \ (timeStampABPB \ n) \ (timeStampABPB \ (S \ n)).$

Proof.

Admitted.

Lemma $timeStampABPCHolds : \forall \ n,$

$Qle \ (timeStampABPC \ n) \ (timeStampABPC \ (S \ n)).$

Proof.

Admitted.

Lemma $timeStampABPDHolds : \forall \ n,$

$Qle \ (timeStampABPD \ n) \ (timeStampABPD \ (S \ n)).$

Proof.

Admitted.

Lemma $timeStampABPEHolds : \forall \ n,$

$Qle \ (timeStampABPE \ n) \ (timeStampABPB \ (S \ n)).$

Proof.

Admitted.

Lemma $timeStampABPFHolds : \forall \ n,$

$Qle \ (timeStampABPF \ n) \ (timeStampABPF \ (S \ n)).$

Proof.

Admitted.

Definition $portA := \{ |$

$ConstraintAutomata.id := A;$

$ConstraintAutomata.dataAssignment := dataAssignmentA;$

$ConstraintAutomata.timeStamp := timeStampABPA;$

$ConstraintAutomata.portCond := timeStampTestABPAHolds;$

$ConstraintAutomata.index := 0 \ | \}$.

Definition $portB := \{ |$

$ConstraintAutomata.id := B;$

$ConstraintAutomata.dataAssignment := dataAssignmentB;$

$ConstraintAutomata.timeStamp := timeStampABPB;$

$ConstraintAutomata.portCond := timeStampTestABPBHolds;$

$ConstraintAutomata.index := 0 \ | \}$.

Definition *portC* := {
 ConstraintAutomata.id := *C*;
 ConstraintAutomata.dataAssignment := *dataAssignmentC*;
 ConstraintAutomata.timeStamp := *timeStampABPC*;
 ConstraintAutomata.portCond := *timeStampABPCHolds*;
 ConstraintAutomata.index := 0 |}

Definition *portD* := {
 ConstraintAutomata.id := *D*;
 ConstraintAutomata.dataAssignment := *dataAssignmentD*;
 ConstraintAutomata.timeStamp := *timeStampABPD*;
 ConstraintAutomata.portCond := *timeStampABPDHolds*;
 ConstraintAutomata.index := 0 |}

Definition *portE* := {
 ConstraintAutomata.id := *E*;
 ConstraintAutomata.dataAssignment := *dataAssignmentE*;
 ConstraintAutomata.timeStamp := *timeStampABPE*;
 ConstraintAutomata.portCond := *timeStampABPEHolds*;
 ConstraintAutomata.index := 0 |}

Definition *portF* := {
 ConstraintAutomata.id := *F*;
 ConstraintAutomata.dataAssignment := *dataAssignmentF*;
 ConstraintAutomata.timeStamp := *timeStampABPF*;
 ConstraintAutomata.portCond := *timeStampABPFHolds*;
 ConstraintAutomata.index := 0 |}

Definition *transformFunction* (*n*:*option nat*) :=
 match *n* with
 | *Some n* ⇒ *Some (n + 1)*
 | *None* ⇒ *None*
 end.

Definition *transformCaBehavior* (*s*:*transformState*) :=
 match *s* with

| $q0ls \Rightarrow [[A;B], \text{ConstraintAutomata.trDc transformFunction } A \ B, [q0ls]]$
end.

Definition *transformCA* := {
 ConstraintAutomata.Q := $[q0ls]$;
 ConstraintAutomata.N := $[A;B]$;
 ConstraintAutomata.T := *transformCaBehavior*;
 ConstraintAutomata.Q0 := $[q0ls]$
}.

Eval compute in *ConstraintAutomata.T transformCA*.

Inductive *lossySyncStates* := $q0$.

Instance *lossySyncStateEq*: *EqDec lossySyncStates eq* :=
{*equiv_dec x y* :=
 match x,y with
 | $q0, q0 \Rightarrow \text{in_left}$
 end }.

Proof.

reflexivity.

Defined.

Definition *dataAssignmentLossySyncBoth n* :=
 match n with
 | $0 \Rightarrow \text{Some } 0$
 | $1 \Rightarrow \text{Some } 1$
 | $S \ n \Rightarrow \text{Some } (1)$
end.

Definition *timeStampLossyA* ($n:nat$) : *QArith_base.Q* :=
 match n with
 | $0 \Rightarrow 1\#1$
 | $S \ n \Rightarrow Z.of_N (N.of_nat(S \ n)) + 1\#1$
end.

Definition *timeStampLossyB* ($n:nat$) : *QArith_base.Q* :=
 match n with

```

| 0 ⇒ 4#1
| S n ⇒ Z.of_N (N.of_nat(S n)) + 1#1
end.

```

Lemma *timeStampTestHoldsLossyA*: $\forall n$,
 $Qle \ (timeStampLossyA \ n) \ (timeStampLossyA \ (S \ n))$.

Proof. *Admitted.*

Lemma *timeStampTestHoldsLossyB*: $\forall n$,
 $Qle \ (timeStampLossyB \ n) \ (timeStampLossyB \ (S \ n))$.

Proof. *Admitted.*

Definition *lossySyncCaBehavior* (s : *lossySyncStates*) :=
 match s with
 | $q0 \Rightarrow [([B;C], ConstraintAutomata.eqDc \ nat \ B \ C, [q0]);$
 $([B], (ConstraintAutomata.tDc \ abpPorts \ nat), [q0])]$
 end.

Definition *lossySyncCA* := {
 $ConstraintAutomata.Q := [q0]$;
 $ConstraintAutomata.N := [B;C]$;
 $ConstraintAutomata.T := lossySyncCaBehavior$;
 $ConstraintAutomata.Q0 := [q0]$
 }.

Inductive *lossySyncStates2* := $q02$.

Instance *lossySyncState2Eq*: $EqDec \ lossySyncStates2 \ eq :=$
 {*equiv_dec* $x \ y :=$
 match x, y with
 | $q02, q02 \Rightarrow in_left$
 end }.

Proof.

reflexivity.

Defined.

Definition *lossySync2CaBehavior* (s : *lossySyncStates2*) :=
 match s with

```

|  $q02 \Rightarrow [[D;E], \text{ConstraintAutomata.eqDc nat } D \ E, [q02]]$ ;
      ( $[D], (\text{ConstraintAutomata.tDc abpPorts nat}), [q02]]$ )

end.

Definition lossySync2CA := {|
  ConstraintAutomata.Q :=  $[q02]$ ;
  ConstraintAutomata.N :=  $[D;E]$ ;
  ConstraintAutomata.T := lossySync2CaBehavior;
  ConstraintAutomata.Q0 :=  $[q02]$ 
|}.

Inductive filterStates3 := q03.

Instance filterState3Eq: EqDec filterStates3 eq :=
  {equiv_dec x y :=
    match x,y with
    |  $q03, q03 \Rightarrow \text{in\_left}$ 
    end }.

Proof.
  reflexivity.
Defined.

Definition filterCaBehavior (s: filterStates3) :=
  match s with
  |  $q03 \Rightarrow [[E;F], \text{ConstraintAutomata.andDc } (\text{ConstraintAutomata.eqDc nat } E \ B)$ 
      ( $\text{ConstraintAutomata.eqDc nat } E$ 
 $F), [q03]]$ ;
      ( $[E], \text{ConstraintAutomata.negDc } (\text{ConstraintAutomata.eqDc nat } E \ B),$ 
 $[q03]]$ )
  end.

Definition filterCA := {|
  ConstraintAutomata.Q :=  $[q03]$ ;
  ConstraintAutomata.N :=  $[E;F]$ ;
  ConstraintAutomata.T := filterCaBehavior;
  ConstraintAutomata.Q0 :=  $[q03]$ 
|}.

```

|}

Definition *filterLossySyncProduct* := *ProductAutomata.buildPA transformCA lossySyncCA*.

Definition *filterLossySyncProduct2* := *ProductAutomata.buildPA lossySync2CA filterCA*.

Definition *resultingPaAbp* := *ProductAutomata.buildPA filterLossySyncProduct filterLossySyncProduct2*.

Eval compute in *ConstraintAutomata.T filterLossySyncProduct (q0ls, q0)*.

Eval compute in *ConstraintAutomata.T filterLossySyncProduct2 (q02, q03)*.

Eval compute in *ConstraintAutomata.Q resultingPaAbp*.

Eval compute in *ConstraintAutomata.T resultingPaAbp (q0ls, q0, (q02, q03))*.

Definition *tdsRun1* := [*portA;portB;portC;portD;portE;portF*].

Definition *steamAcce* := **Eval** compute in *ConstraintAutomata.run resultingPaAbp tdsRun1* 3.

Lemma *streamAccept8* : $\neg \text{In } ([\])$ (*steamAcce*).

Proof. unfold *not*. unfold *steamAcce*. simpl. intros. repeat (destruct *H* ; inversion *H*). **Defined.**

Definition *dataAssignmentA2 n* :=

match *n* with

| 0 \Rightarrow *Some* 0

| 1 \Rightarrow *Some* 0

| 2 \Rightarrow *Some* 1

| *S n* \Rightarrow *Some* (1)

end.

Definition *dataAssignmentB2 n* :=

match *n* with

| 0 \Rightarrow *Some* 1

| 1 \Rightarrow *Some* (1)

| 2 \Rightarrow *Some* 2

| *S n* \Rightarrow *Some* 2

end.

Definition *dataAssignmentC2 n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentD2* n :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentE2* n :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentF2* n :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *timeStampABPA2*(n:nat) : *QArith_base.Q* :=

```

match n with
| 0 ⇒ 2#1
| 1 ⇒ 5#1
| 2 ⇒ 8#1

```



```

| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPB2* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0 ⇒ 2#1
| 1 ⇒ 5#1
| 2 ⇒ 8#1
| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPC2* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0 ⇒ 2#1
| 1 ⇒ 5#1
| 2 ⇒ 8#1
| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPD2* ($n:nat$) : $QArith_base.Q :=$

```

match  $n$  with
| 0 ⇒ 2#10
| 1 ⇒ 5#10
| 2 ⇒ 8#10
| 3 ⇒ 11#10
| 4 ⇒ 14#10

```

```

| 5 ⇒ 17#10
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPE2*($n:nat$) : $QArith_base.Q :=$

```

  match  $n$  with
  | 0 ⇒ 2#1
  | 1 ⇒ 5#1
  | 2 ⇒ 8#1
  | 3 ⇒ 11#1
  | 4 ⇒ 14#1
  | 5 ⇒ 17#1
  |  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
  end.

```

Definition *timeStampABPF2* ($n:nat$) : $QArith_base.Q :=$

```

  match  $n$  with
  | 0 ⇒ 2#1
  | 1 ⇒ 5#1
  | 2 ⇒ 8#1
  | 3 ⇒ 11#1
  | 4 ⇒ 14#1
  | 5 ⇒ 17#1
  |  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
  end.

```

Lemma *timeStampTestABPAHolds2* : $\forall\ n, Qle\ (timeStampABPA2\ n)\ (timeStampABPA2\ (S\ n))$.

Proof.

Admitted.

Lemma *timeStampTestABPBHolds2* : $\forall\ n,$
 $Qle\ (timeStampABPB2\ n)\ (timeStampABPB2\ (S\ n))$.

Proof.

Admitted.

Lemma *timeStampABPCHolds2* : $\forall n,$

$Qle (timeStampABPC2\ n) (timeStampABPC2\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPDHolds2* : $\forall n,$

$Qle (timeStampABPD2\ n) (timeStampABPD2\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPEHolds2* : $\forall n,$

$Qle (timeStampABPE2\ n) (timeStampABPB2\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPFHolds2* : $\forall n,$

$Qle (timeStampABPF2\ n) (timeStampABPF2\ (S\ n)).$

Proof.

Admitted.

Definition *portA2* := { |

ConstraintAutomata.id := *A*;

ConstraintAutomata.dataAssignment := *dataAssignmentA2*;

ConstraintAutomata.timeStamp := *timeStampABPA2*;

ConstraintAutomata.portCond := *timeStampTestABPAHolds2*;

ConstraintAutomata.index := 0 | }.

Definition *portB2* := { |

ConstraintAutomata.id := *B*;

ConstraintAutomata.dataAssignment := *dataAssignmentB2*;

ConstraintAutomata.timeStamp := *timeStampABPB2*;

ConstraintAutomata.portCond := *timeStampTestABPBHolds2*;

ConstraintAutomata.index := 0 | }.

Definition *portC2* := { |

ConstraintAutomata.id := *C*;

ConstraintAutomata.dataAssignment := *dataAssignmentC2*;

$\text{ConstraintAutomata.timeStamp} := \text{timeStampABPC2};$
 $\text{ConstraintAutomata.portCond} := \text{timeStampABPCHolds2};$
 $\text{ConstraintAutomata.index} := 0 \mid \}.$

Definition $\text{portD2} := \{ \mid$
 $\text{ConstraintAutomata.id} := D;$
 $\text{ConstraintAutomata.dataAssignment} := \text{dataAssignmentD2};$
 $\text{ConstraintAutomata.timeStamp} := \text{timeStampABPD2};$
 $\text{ConstraintAutomata.portCond} := \text{timeStampABPDHolds2};$
 $\text{ConstraintAutomata.index} := 0 \mid \}.$

Definition $\text{portE2} := \{ \mid$
 $\text{ConstraintAutomata.id} := E;$
 $\text{ConstraintAutomata.dataAssignment} := \text{dataAssignmentE2};$
 $\text{ConstraintAutomata.timeStamp} := \text{timeStampABPE2};$
 $\text{ConstraintAutomata.portCond} := \text{timeStampABPEHolds2};$
 $\text{ConstraintAutomata.index} := 0 \mid \}.$

Definition $\text{portF2} := \{ \mid$
 $\text{ConstraintAutomata.id} := F;$
 $\text{ConstraintAutomata.dataAssignment} := \text{dataAssignmentF2};$
 $\text{ConstraintAutomata.timeStamp} := \text{timeStampABPF2};$
 $\text{ConstraintAutomata.portCond} := \text{timeStampABPFHolds2};$
 $\text{ConstraintAutomata.index} := 0 \mid \}.$

Definition $\text{input2} := [\text{portA2}; \text{portB2}; \text{portC2}; \text{portD2}; \text{portE2}; \text{portF2}].$

Definition $\text{onlyDHasData} := \text{Eval compute in ConstraintAutomata.run result-}$
 $\text{ingPaAbp input2 5.}$

Definition $\text{onlyDwithData} : \neg \text{In } [] \text{ onlyDHasData.}$

Proof. $\text{unfold not. unfold steamAcce. simpl. intros. repeat (destruct}$
 $H ; \text{inversion } H). \text{Defined.}$

Definition $\text{dataAssignmentA3 } n :=$

$\text{match } n \text{ with}$

$\mid 0 \Rightarrow \text{Some } 0$

$\mid 1 \Rightarrow \text{Some } 0$

```

| 2 ⇒ Some 1
| S n ⇒ Some (1)
end.

```

Definition *dataAssignmentB3* *n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentC3* *n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentD3* *n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentE3* *n* :=

```

match n with
| 0 ⇒ Some 1
| 1 ⇒ Some (1)
| 2 ⇒ Some 2
| S n ⇒ Some 2
end.

```

Definition *dataAssignmentF3* $n :=$

```

  match  $n$  with
  | 0  $\Rightarrow$  Some 1
  | 1  $\Rightarrow$  Some (1)
  | 2  $\Rightarrow$  Some 2
  | S  $n \Rightarrow$  Some 2
  end.

```

Definition *timeStampABPA3* ($n:nat$) : *QArith_base.Q* :=

```

  match  $n$  with
  | 0  $\Rightarrow$  2#1
  | 1  $\Rightarrow$  5#1
  | 2  $\Rightarrow$  8#1
  | 3  $\Rightarrow$  11#1
  | 4  $\Rightarrow$  14#1
  | 5  $\Rightarrow$  17#1
  | S  $n \Rightarrow$  Z.of_N (N.of_nat(S  $n$ )) + 1#1
  end.

```

Definition *timeStampABPB3* ($n:nat$) : *QArith_base.Q* :=

```

  match  $n$  with
  | 0  $\Rightarrow$  2#1
  | 1  $\Rightarrow$  5#1
  | 2  $\Rightarrow$  8#1
  | 3  $\Rightarrow$  11#1
  | 4  $\Rightarrow$  14#1
  | 5  $\Rightarrow$  17#1
  | S  $n \Rightarrow$  Z.of_N (N.of_nat(S  $n$ )) + 1#1
  end.

```

Definition *timeStampABPC3* ($n:nat$) : *QArith_base.Q* :=

```

  match  $n$  with
  | 0  $\Rightarrow$  2#1
  | 1  $\Rightarrow$  5#1
  | 2  $\Rightarrow$  8#1

```

```

| 3 ⇒ 11#1
| 4 ⇒ 14#1
| 5 ⇒ 17#1
|  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
end.

```

Definition *timeStampABPD3* ($n:nat$) : $QArith_base.Q :=$

```

  match  $n$  with
  | 0 ⇒ 2#1
  | 1 ⇒ 5#1
  | 2 ⇒ 8#1
  | 3 ⇒ 11#1
  | 4 ⇒ 14#1
  | 5 ⇒ 17#1
  |  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
  end.

```

Definition *timeStampABPE3* ($n:nat$) : $QArith_base.Q :=$

```

  match  $n$  with
  | 0 ⇒ 2#1
  | 1 ⇒ 5#1
  | 2 ⇒ 8#10
  | 3 ⇒ 11#1
  | 4 ⇒ 14#1
  | 5 ⇒ 17#1
  |  $S\ n \Rightarrow Z.of\_N\ (N.of\_nat(S\ n)) + 1\#1$ 
  end.

```

Definition *timeStampABPF3* ($n:nat$) : $QArith_base.Q :=$

```

  match  $n$  with
  | 0 ⇒ 2#1
  | 1 ⇒ 5#1
  | 2 ⇒ 8#1
  | 3 ⇒ 11#1
  | 4 ⇒ 14#1

```

| 5 \Rightarrow 17#1
 | $S\ n \Rightarrow Z.of_N\ (N.of_nat(S\ n)) + 1\#1$
 end.

Lemma *timeStampTestABPAHolds3* : $\forall\ n,$ $Qle\ (timeStampABPA2\ n)\ (timeStampABPA2\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampTestABPBHolds3* : $\forall\ n,$
 $Qle\ (timeStampABPB3\ n)\ (timeStampABPB3\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPCHolds3* : $\forall\ n,$
 $Qle\ (timeStampABPC3\ n)\ (timeStampABPC3\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPDHolds3* : $\forall\ n,$
 $Qle\ (timeStampABPD3\ n)\ (timeStampABPD3\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPEHolds3* : $\forall\ n,$
 $Qle\ (timeStampABPE3\ n)\ (timeStampABPE3\ (S\ n)).$

Proof.

Admitted.

Lemma *timeStampABPFHolds3* : $\forall\ n,$
 $Qle\ (timeStampABPF3\ n)\ (timeStampABPF3\ (S\ n)).$

Proof.

Admitted.

Definition *portA3* := {
 ConstraintAutomata.id := *A*;
 ConstraintAutomata.dataAssignment := *dataAssignmentA3*;
 ConstraintAutomata.timeStamp := *timeStampABPA3*;

ConstraintAutomata.portCond := *timeStampTestABPAHolds3*;
ConstraintAutomata.index := 0 |}.

Definition *portB3* := {|
ConstraintAutomata.id := *B*;
ConstraintAutomata.dataAssignment := *dataAssignmentB3*;
ConstraintAutomata.timeStamp := *timeStampABPB3*;
ConstraintAutomata.portCond := *timeStampTestABPBHolds3*;
ConstraintAutomata.index := 0 |}.

Definition *portC3* := {|
ConstraintAutomata.id := *C*;
ConstraintAutomata.dataAssignment := *dataAssignmentC3*;
ConstraintAutomata.timeStamp := *timeStampABPC3*;
ConstraintAutomata.portCond := *timeStampABPCHolds3*;
ConstraintAutomata.index := 0 |}.

Definition *portD3* := {|
ConstraintAutomata.id := *D*;
ConstraintAutomata.dataAssignment := *dataAssignmentD3*;
ConstraintAutomata.timeStamp := *timeStampABPD3*;
ConstraintAutomata.portCond := *timeStampABPDHolds3*;
ConstraintAutomata.index := 0 |}.

Definition *portE3* := {|
ConstraintAutomata.id := *E*;
ConstraintAutomata.dataAssignment := *dataAssignmentE3*;
ConstraintAutomata.timeStamp := *timeStampABPE3*;
ConstraintAutomata.portCond := *timeStampABPEHolds3*;
ConstraintAutomata.index := 0 |}.

Definition *portF3* := {|
ConstraintAutomata.id := *F*;
ConstraintAutomata.dataAssignment := *dataAssignmentF3*;
ConstraintAutomata.timeStamp := *timeStampABPF3*;
ConstraintAutomata.portCond := *timeStampABPFHolds3*;

ConstraintAutomata.index := 0 |}.

Definition *input3* := [*portA3*; *portB3*; *portC3*; *portD3*; *portE3*; *portF3*].

Definition *runAbpCAnotAccept* := **Eval** compute in *ConstraintAutomata.run* *resultingPaAbp input3* 6.

Lemma *runAbpCAnotAc* : *In [] runAbpCAnotAccept*.

Proof. **simpl**; **auto**. **Defined**.

Appendix C

Constraint Automata Definitions

C.1 General Helpers

Lemma 1 (Soundness of $s1_in_s2$). $\forall s1, \forall s2, s1_in_s2\ s1\ s2 = true \leftrightarrow s1 = [] \vee (\forall a, In\ a\ s1 \rightarrow set_mem\ equiv_dec\ a\ s2 = true)$. The function $s1_in_s2$ with two sets A and B returns true if and only if all elements of A are in B and false otherwise, where $equiv_dec$ here (and in what follows for the rest of the document) is the equality relation between elements provided by class $EqDec$, which is in Coq's standard library.

Proof. The proof is straightforward from $s1_in_s2$'s definition. By unfolding $s1_in_s2$, it is

```
match A with
| []  $\Rightarrow true$ 
|  $a::t \Rightarrow set\_mem\ equiv\_dec\ a\ B \ \&\&\ s1\_in\_s2\ t\ B$ 
end.
```

where set_mem is a function within `Module ListSet` that returns true if a given element a is in a given set B and false otherwise. The notation $\&\&$ stands for the boolean conjunction defined in Coq's standard library. By applying $set_mem\ a\ B$ for every element a in A as a conjunction between all executions of $set_mem\ a\ B$, we have that $s1_in_s2$ returns true iff all elements of A are in B , proving this Lemma. \square

Lemma 2 (Soundness of set_eq). The function set_eq returns true iff given two sets A and B , all elements of A are in B and all elements of B are in A .

Proof. The proof is straightforward from set_eq 's definition. It suffices to unfold set_eq 's

definition and verify that it returns *true* if both sets have the same cardinality and if all elements of *A* are in *B* (and vice-versa) which is verified by applying *s1_in_s2 A B* and *s1_in_s2 B A*. Therefore by Lemma 1, this Lemma is proved. \square

Lemma 3 (Soundness of *count_into_list*). *The function *count_into_list* with a natural number *k* returns an ordered set of natural numbers ranging from 0 to *k*. This function will later be used as the set of natural numbers used by *run* to enable bounded runs with *k* steps.*

Proof. The proof proceeds by induction on the natural number *k*. By unfolding *count_into_list*'s definition, it is

```
match k with
| 0  $\Rightarrow$  0::nil
| S k  $\Rightarrow$  count_into_list k ++ [S k]
end.
```

- (i) Induction Basis: Let *k* = 0. The aim is to prove that *count_into_list* returns an ordered set with elements ranging from 0 to 0. By *count_into_list*'s definition, with *k* = it returns an list containing only one element, namely, its return is [0]. Therefore the goal holds.
- (ii) Induction Hypothesis: Suppose *k* is a natural number different from zero. Therefore *count_into_list* with *k* returns an set with ordered elements ranging from 0 to *k*.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for a natural number denoted by *k* + 1. From *count_into_list*'s definition, the return of *count_into_list* with *k* + 1 is *count_into_list k* ++ [S *k*]. By applying the Induction Hypothesis, it is [0, 1, ..., *k*] ++ [S *n*] where *S* is the sucessor function. As *count_into_list* returns exactly a set containing natural numbers ordered from 0 to *k* + 1 the goal is proved.

\square

C.2 Core Definitions

Lemma 4 (Soundness of *evalDC*). $\forall po: \text{option port}, \forall d: \text{data}, \text{evalDC } po \ d = \text{true} \leftrightarrow \exists x, po = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. This Lemma states the soundness of *evalDC*, a function that with a option port *po* and a data item *d* returns true iff *po* is different from *None* (this denotes that the port is a valid port) and its *dataAssignment* field with its *index* value matches *d*.

Proof.

\Rightarrow

The proof is done by case analysis on *po*. This proof wants to show that *evalDC* returns true iff all ports given as parameter are valid ports and there is a data item in its data stream that matches the data item *d*.

Let $H: \text{evalDC } po \ d = \text{true}$. The goal is $\exists x, po = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. By destructing *po*, two subgoals are generated, one for each constructor of *po*'s type. Therefore let *po*:

- *Some n*. *H* now is $\text{evalDC } \text{Some } n \ d = \text{true}$. The first subgoal now is $\exists x, \text{Some } n = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. By simplification, *H*'s right hand side unfolds to *evalDC*'s definition, which states that “if *n*'s *dataAssignment* field with *n*'s index equals *d*, then it returns true, otherwise, false” = *true*. By destructing the equality relation between *n*'s *dataAssignment* field with *n* and *d*, another subgoal is generated, equal to the current subgoal. *H* now is “Let *n*'s *dataAssignment* field with *n*'s index be equal to *d*”. The current goal is $\exists x, \text{Some } n = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. Since *n* is a port, it can be instantiated in order to eliminate the existential quantifier in the goal. By replacing it in the goal, it now is $\text{Some } n = \text{Some } n \wedge \text{dataAssignment } p(\text{index } p) = d$ which, from *H*, holds; The other subgoals stands for the case that *H*'s right hand side be “Let *n*'s *dataAssignment* field with *n*'s index not equal to *d*”. *H* now is “Let *n*'s *dataAssignment* field with *n*'s index not equal to *d*” = *false* This is a contradiction, since from *evalDC*'s definition, it only returns *true* if the port given is a valid port and its *dataAssignment* with its index matches the data item *d*. The current subgoal is then discharged.
- *None*. *H* now is $\text{evalDC } \text{None} \ d = \text{true}$. It contradicts the definition of *evalDC*

that with *None* evaluates to *false*, discharging the current goal.

\Leftarrow

Let H a hypothesis such as $H: \exists x, po = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. The goal is $\text{evalDC } \text{Some } n \ d = \text{true}$. From H one can derive that there exists a given port x . H now is $po = \text{Some } x \wedge \text{dataAssignment } x(\text{index } x) = d$. From H it is possible to derive as another hypothesis $H2$ that $\text{dataAssignment } x(\text{index } x) = d$. By replacing po in the goal with $\text{Some } x$ it now is $\text{evalDC } \text{Some } x \ d = \text{true}$. By unfolding evalDC 's definition, the goal evaluates to “if x 's dataAssignment field with x 's index equals d ” then returns true, else returns $\text{false} = \text{true}$. From H and $H2$ the goal evaluates to $\text{true} = \text{true}$, which holds. \square

Lemma 5 (Soundness of *retrievePortFromInput*). $\forall s:\text{set port}, \forall n:\text{name}, \forall a, \text{retrievePortFromInput } s \ n = \text{Some } a \rightarrow \text{In } a \ s \wedge n = \text{id } a$. The function *retrievePortFromInput* is a function that retrieves all ports from a set of ports s that has its *id* value equal to the given name n . *In* is a predicate present in Coq's standard library that denotes an element in a list. As it is used sets implemented as lists in Coq, one can use this predicate for structures of type *set*.

Proof.

\Rightarrow

The proof proceeds by induction on the set of ports s . Therefore let

$H: \text{retrievePortFromInput } s \ n = \text{Some } a$ a hypothesis. The goal is to prove that $\text{In } a \ s \wedge n = \text{id } a$ holds. Let

- (i) Induction Basis: Let s be an empty set. Then $H: \text{retrievePortFromInput } [] \ n = \text{Some } a$ holds. The goal is $\text{In } a \ [] \wedge n = \text{id } a$. By *retrievePortFromInput*'s definition, H is a contradiction. Then, the goal holds.
- (ii) Induction Hypothesis: Suppose that the induction basis holds for an arbitrary nonempty set s . This means that $\text{retrievePortFromInput } s \ n = \text{Some } a \rightarrow \text{In } a \ s \wedge n = \text{id } a$ holds.
- (iii) Inductive Step: The objective is to prove $\text{In } a \ (a_0 :: s) \wedge n = \text{id } a$. In other words, the set supplied for *retrievePortFromInput* contains an element which is not in s . The hypothesis H now is $H: \text{retrievePortFromInput } (a_0 :: s) \ n = \text{Some } a$. Also,

let $IH: \text{retrievePortFromInput } s \ n = \text{Some } a \rightarrow \text{In } a \ s \wedge n = \text{id } a$ the induction hypothesis. By simplification on H it is $(\text{if } \text{equiv_dec } n \ (\text{id } a_0) \ \text{then } \text{Some } a_0 \ \text{else } \text{retrievePortFromInput } s \ n) = \text{Some } a$. Upon analyzing equiv_dec by destruction, two possible hypothesis are obtained, where for each of them the current goal must hold.

- (a) Let $n = \text{id } a_0$. A new hypothesis H_2 is added to the proof context, where $H_2: n = \text{id } a_0$. The goal is $\text{In } a \ (a_0 :: s) \wedge n = \text{id } a$. H is $H: \text{Some } a_0 = \text{Some } a$. From H a new hypothesis H_3 can be introduced as $H_3: a_0 = a$. By rewriting H_3 in the goal, replacing occurrences of a_0 with a , the goal is $\text{In } a \ (a :: s) \wedge n = \text{id } a$. Also, by rewriting H_3 in H_2 , it is $H_2: n = \text{id } a$. The left side of the conjunction in the goal holds by unfolding it, which is $a = a \vee \text{In } a \ s$. The right side of the conjunction in the goal holds by H_2 . Therefore the goal holds.
- (b) Let $n \neq \text{id } a_0$. A new hypothesis H_2 is added to the proof context, where $H_2: n \neq \text{id } a_0$. H is $H: \text{retrievePortFromInput } s \ n = \text{Some } a$. The goal is $\text{In } a \ (a_0 :: s) \wedge n = \text{id } a$. By applying IH on H , H is $H: \text{In } a \ s \wedge n = \text{id } a$. H allows one to derive two new hypothesis by eliminating the conjunction in it. Therefore let $H_3: \text{In } a \ s$ and $H_4: n = \text{id } a$. By simplification on the goal, it unfolds to $(a = a_0 \vee \text{In } a \ s) \wedge n = \text{id } a$, which holds from H_3 and H_4 .

□

Lemma 6 (Soundness of eqDataPorts). $\forall n1, \forall n2, \forall s, \text{eqDataPorts } n1 \ n2 \ s = \text{true} \leftrightarrow \exists a, \text{retrievePortFromInput } s \ n1 = \text{Some } a \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. The function eqDataPorts is employed by evalCompositeDc in order to provide semantics for the constructor eqDc of DC . Therefore eqDataPorts returns true if and only if both ports $n1$ and $n2$ contain the same data element at the same time. In other words, when evaluating both $n1$ and $n2$'s respective streams, they must have the same data item at the same time (which is $\theta.\text{time}(k)$ even if they have different index values).

Proof. Let $H: \text{eqDataPorts } n1 \ n2 \ s = \text{true}$. The objective is to prove $\exists a, \text{retrievePortFromInput } s \ n1 = \text{Some } a \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. By unfolding eqDataPorts 's definition in H , it is

```

match (retrievePortFromInput s n1) with
| Some a ⇒ match (retrievePortFromInput s n2) with
    | Some b ⇒ if (dataAssignment a(index a)) ==
        (dataAssignment b(index b)) then true else false
    | None ⇒ false
end
| None ⇒ false
end.

```

By case analysis on both *retrievePortFromInput s n1* and *retrievePortFromInput s n2*, four hypothesis are obtained where the current goal must hold.

- (i) let $H_2: \text{retrievePortFromInput } s \ n1 = \text{Some } p$ and $H_3: \text{retrievePortFromInput } s \ n2 = \text{Some } q$, with both p and q with type *port*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{Some } p = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. By rewriting both H_2 and H_3 in H , H now is $(\text{if equiv_dec } (\text{dataAssignment } p(\text{index } p)) (\text{dataAssignment } q(\text{index } q)) \text{ then true else false}) = \text{true}$. Upon destructing *equiv_dec*, two possibilities are generated:

- (a) let $(\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } q(\text{index } q))$. Let H_4 a new hypothesis introduced in the proof context, where $H_4: (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } q(\text{index } q))$. H is $H: \text{true} = \text{true}$. The goal is $\exists a, \text{Some } p = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. The existential quantifier over a and b can be eliminated by instantiating p as a and q as b , the goal is $\text{Some } p = \text{Some } p \wedge \text{Some } q = \text{Some } q \wedge (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } q(\text{index } q))$. The goal holds because $\text{Some } p = \text{Some } p$ and $\text{Some } q = \text{Some } q$ holds. The rightmost part of the goal holds from H_4 .
- (b) $(\text{dataAssignment } p(\text{index } p)) \neq (\text{dataAssignment } q(\text{index } q))$. H is $H: \text{false} = \text{true}$. The goal is $\exists a, \text{Some } p = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. As H is a contradiction, the current goal is discharged.

- (ii) let $H_2: \text{retrievePortFromInput } s \ n1 = \text{None}$ and $H_3: \text{retrievePortFromInput } s \ n2$

= *Some q*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{None} = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_2 in H , H is $H: \text{false} = \text{true}$. As H is a contradiction, the current goal is discharged.

(iii) let $H_2: \text{retrievePortFromInput } s \ n1 = \text{Some } p$ and $H_3: \text{retrievePortFromInput } s \ n2 = \text{None}$. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{Some } q = \text{Some } a \wedge \exists b, \text{None} = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_3 in H , H is $H: \text{false} = \text{true}$. As H is a contradiction, the current goal is discharged.

(iv) let $H_2: \text{retrievePortFromInput } s \ n1 = \text{None}$ and $H_3: \text{retrievePortFromInput } s \ n2 = \text{None}$. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{None} = \text{Some } a \wedge \exists b, \text{None} = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_2 in H , H is $H: \text{false} = \text{true}$. As H is a contradiction, the current goal is discharged.

\Leftarrow

Suppose $H: \exists a, \text{retrievePortFromInput } s \ n1 = \text{Some } a \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge (\text{dataAssignment } a(\text{index } a)) = (\text{dataAssignment } b(\text{index } b))$. The objective is to prove $\text{eqDataPorts } n1 \ n2 \ s = \text{true}$. From H , a port p can be obtained by introducing the existential quantifier over a . H is $H: \text{retrievePortFromInput } s \ n1 = \text{Some } p \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } b(\text{index } b))$. Two hypothesis can be derived from H , $H_2: \text{retrievePortFromInput } s \ n1 = \text{Some } p$ and $H_3: \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } b(\text{index } b))$. By introducing the existential quantifier in H_3 over b , a port q is introduced in the proof context. H_3 is $H_3: \text{retrievePortFromInput } s \ n2 = \text{Some } q \wedge (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } q(\text{index } q))$. H_3 can be split into two hypothesis, namely $H_3: \text{retrievePortFromInput } s \ n2 = \text{Some } q$ and $H_4: (\text{dataAssignment } p(\text{index } p)) = (\text{dataAssignment } q(\text{index } q))$. By unfolding eqDataPorts in the goal, it is

```
match (retrievePortFromInput s n1) with
| Some a  $\Rightarrow$  match (retrievePortFromInput s n2) with
| Some b  $\Rightarrow$  if (dataAssignment a(index a)) ==
```

```

      (dataAssignment b(index b)) then true else false
    | None ⇒ false
  end
| None ⇒ false
end.

```

By rewriting H_2 , H_3 and H_4 (`if equiv_dec (dataAssignment x0 (index x0)) (dataAssignment x0 (index x0)) then true else false`). As $(dataAssignment x0 (index x0)) = (dataAssignment x0 (index x0))$, the goal unfolds to `true = true` which holds. \square

Lemma 7 (Soundness of *transformDC*). $\forall transform, \forall n1, \forall n2, \forall s, transformDC\ transform\ n1\ n2\ s = true \leftrightarrow \exists a, retrievePortFromInput\ s\ n1 = Some\ a \wedge \exists b, retrievePortFromInput\ s\ n2 = Some\ b \wedge transform((dataAssignment\ a(index\ a))) = (dataAssignment\ b(index\ b))$.

Proof. This proof is quite similar of *eqDataPorts*'s soundness proof. Let $H: transformDC\ transform\ n1\ n2\ s = true$. The objective is to prove $\exists a, retrievePortFromInput\ s\ n1 = Some\ a \wedge \exists b, retrievePortFromInput\ s\ n2 = Some\ b \wedge transform((dataAssignment\ a(index\ a))) = (dataAssignment\ b(index\ b))$. By unfolding *eqDataPorts*'s definition in H , it is

```

match (retrievePortFromInput s n1) with
| Some a ⇒ match (retrievePortFromInput s n2) with
      | Some b ⇒ if equiv_dec (transform((dataAssignment a(index a)))
(dataAssignment b(index b))) then true else false
      | None ⇒ false
    end
| None ⇒ false
end.

```

end. By case analysis on both *retrievePortFromInput s n1* and *retrievePortFromInput s n2*, four hypothesis are obtained where the current goal must hold.

- (i) let $H_2: retrievePortFromInput\ s\ n1 = Some\ p$ and $H_3: retrievePortFromInput\ s\ n2 = Some\ q$, with both p and q with type *port*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, Some\ p = Some\ a \wedge \exists b, Some\ q = Some\ b \wedge transform((dataAssignment\ a(index\ a))) = (dataAssignment\ b(index\ b))$. By rewriting both H_2 and H_3 in H , H now is `(if equiv_dec transform ((dataAssignment p (index p))) (dataAssign-`

ment *q* (*index* *q*)) *then true else false*) = *true*. Upon destructing *equiv_dec*, two possibilities are generated:

(a) let *transform*((*dataAssignment* *p* (*index* *p*))) = (*dataAssignment* *q* (*index* *q*)).

Let H_4 a new hypothesis introduced in the proof context, where H_4 : *transform*((*dataAssignment* *p* (*index* *p*))) = (*dataAssignment* *q* (*index* *q*)). H is H : *true* = *true*. The goal is $\exists a, \text{Some } p = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$. The existential quantifier over *a* and *b* can be eliminated by instantiating *p* as *a* and *q* as *b*, the goal is *Some* *p* = *Some* *p* \wedge *Some* *q* = *Some* *q* \wedge *transform*((*dataAssignment* *a*(*index* *a*))) = (*dataAssignment* *b*(*index* *b*)). The goal holds because *Some* *p* = *Some* *p* and *Some* *q* = *Some* *q* holds. The rightmost part of the goal holds from H_4 .

(b) *transform*((*dataAssignment* *p* (*index* *p*))) \neq (*dataAssignment* *q* (*index* *q*)). H is H : *false* = *true*. The goal is $\exists a, \text{Some } p = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$.

As H is a contradiction, the current goal is discharged.

(ii) let H_2 : *retrievePortFromInput* *s* *n1* = *None* and H_3 : *retrievePortFromInput* *s* *n2* = *Some* *q*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{None} = \text{Some } a \wedge \exists b, \text{Some } q = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_2 in H , H is H : *false* = *true*. As H is a contradiction, the current goal is discharged.

(iii) let H_2 : *retrievePortFromInput* *s* *n1* = *Some* *p* and H_3 : *retrievePortFromInput* *s* *n2* = *None*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{Some } q = \text{Some } a \wedge \exists b, \text{None} = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_3 in H , H is H : *false* = *true*. As H is a contradiction, the current goal is discharged.

(iv) let H_2 : *retrievePortFromInput* *s* *n1* = *None* and H_3 : *retrievePortFromInput* *s* *n2* = *None*. By rewriting H_2 and H_3 in the goal, it now is $\exists a, \text{None} = \text{Some } a \wedge \exists b, \text{None} = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$. By rewriting H_2 in H , H is H : *false* = *true*. As H is a contradiction, the current goal is discharged.

\Leftarrow

Suppose $H: \exists a, \text{retrievePortFromInput } s \ n1 = \text{Some } a \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge \text{transform}((\text{dataAssignment } a(\text{index } a))) = (\text{dataAssignment } b(\text{index } b))$.

The objective is to prove $\text{transformDc transform } n1 \ n2 \ s = \text{true}$. From H , a port p can be obtained by introducing the existential quantifier over a . H is $H: \text{retrievePortFromInput } s \ n1 = \text{Some } p \wedge \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge \text{transform}((\text{dataAssignment } p(\text{index } p))) = (\text{dataAssignment } b(\text{index } b))$. Two hypothesis can be derived from H , $H_2: \text{retrievePortFromInput } s \ n1 = \text{Some } p$ and $H_3: \exists b, \text{retrievePortFromInput } s \ n2 = \text{Some } b \wedge \text{transform}((\text{dataAssignment } p(\text{index } p))) = (\text{dataAssignment } b(\text{index } b))$. By introducing the existential quantifier in H_3 over b , a port q is introduced in the proof context. H_3 is $H_3: \text{retrievePortFromInput } s \ n2 = \text{Some } q \wedge \text{transform}((\text{dataAssignment } p(\text{index } p))) = (\text{dataAssignment } q(\text{index } q))$. H_3 can be split into two hypothesis, namely $H_3: \text{retrievePortFromInput } s \ n2 = \text{Some } q$ and $H_4: \text{transform}((\text{dataAssignment } p(\text{index } p))) = (\text{dataAssignment } q(\text{index } q))$. By unfolding eqDataPorts in the goal, it is

```

match (retrievePortFromInput s n1) with
| Some a  $\Rightarrow$  match (retrievePortFromInput s n2) with
    | Some b  $\Rightarrow$  if (equiv_dec transform((dataAssignment a(index a))
(dataAssignment b(index b)))) then true else false
    | None  $\Rightarrow$  false
end
| None  $\Rightarrow$  false
end.

```

By rewriting H_2 , H_3 and H_4 on the goal, it is $(\text{if equiv_dec transform } ((\text{dataAssignment } p(\text{index } p))) (\text{dataAssignment } p(\text{index } p))) \text{ then true else false}$. As $\text{transform}((\text{dataAssignment } p(\text{index } p))) = (\text{dataAssignment } p(\text{index } p))$, the goal unfolds to $\text{true} = \text{true}$ which holds. \square

Lemma 8 (Soundness of $\text{returnSmallerNumber}$). $\forall m, \forall l, \text{returnSmallerNumber } m \ l \neq m \leftrightarrow l \neq [] \wedge \exists a, \text{In } a \ l \wedge a < m = \text{true}$. The function $\text{returnSmallerNumber}$ returns a number different from m iff there is a number a in the set of numbers provided such that $a < m$.

Proof.

\Rightarrow

The proof proceeds by induction on the set of rational numbers l . Let

- (i) Induction basis: Let l be an empty set. Then, let H be “*returnSmallerNumber* with m and $[]$ returns a number x which is different from m ”. The goal now is $[] \neq [] \wedge \exists a, \text{In } a [] \wedge a < m = \text{true}$. By *returnSmallerNumber*’s definition, H is a contradiction (*returnSmallerNumber* may return m if l is empty). Then, it holds.
- (ii) Induction Hypothesis: Suppose that the induction basis holds for an non-empty arbitrary set l . In other words, *returnSmallerNumber* with m and s returns a number which is different from m holds, then $l \neq [] \wedge \exists a, \text{In } a l \wedge a < m = \text{true}$ also holds.
- (iii) Inductive Step: The aim is to prove that $a::t \neq [] \wedge \exists a, \text{In } a a::t \wedge a < m = \text{true}$. The hypothesis H now is “*returnSmallerNumber* with m and $a :: s$ returns a number a which is different from m ”. By simplification upon H , by *returnSmallerNumber*’s definition, H is “If $a < m$ then *returnSmallerNumber* $a l$ holds, otherwise *returnSmallerNumber* $m l$ holds” is different from m . By case analysis on H , there are two possible goals:
 - (a) if $a < m$ holds, then H is *returnSmallerNumber* $a l$ is different from m also holds. The existential quantifier can be eliminated by instantiating the quantified variable with a . The goal now is $\text{In } a a::t \wedge a < m = \text{true}$, which holds.
 - (b) if $a < m$ does not hold, then H is *returnSmallerNumber* $m l$ is different from m holds. By applying the induction hypothesis on $H2$. $H2$ is transformed into $l \neq [] \wedge \exists a, \text{In } a l \wedge a < m = \text{true}$. By applying the elimination of conjunction, we have that $H2$ now is $l \neq []$ and $H3$ is a new hypothesis, $\exists a, \text{In } a l \wedge a < m = \text{true}$, which matches the goal.

\Leftarrow

The proof proceeds by induction on the set of rational numbers l .

- (i) Induction Basis: l be an empty set. Then let H be “ $[] \neq [] \wedge \exists a, \text{In } a [] \wedge a < m = \text{true}$ ”. The goal is “*returnSmallerNumber* with m and $[]$ returns a number x which is different from m ”. H is a contradiction because $[] \neq []$ does not hold so this goal is discharged.

- (ii) Induction Hypothesis: Suppose that the inductive basis holds for an nonempty arbitrary set l . This means that “ $(l \neq [] \wedge \exists a, \text{In } a \ l \wedge a < m = \text{true}) \rightarrow \text{returnSmallerNumber } m \ l \neq m$ ” holds.
- (iii) Inductive step: The aim is to prove that “ $\text{returnSmallerNumber}$ with m and $a :: s$ returns a number a which is different from m ”. H is $a::t \neq [] \wedge \exists a, \text{In } a \ a::t \wedge a < m = \text{true}$. By simplification on the goal, it is “If $a < m$ then $\text{returnSmallerNumber } a \ l <> m$ holds, otherwise $\text{returnSmallerNumber } m \ l <> m$ holds”. Therefore the goal is split in two
- (a) if $a < m$ holds then the goal is $\text{returnSmallerNumber } a \ l <> m$. By $\text{returnSmallerNumber}$ ’s definition it returns a number a different from m given as parameter if there is a number a in the set given as parameter. Therefore this goal holds.
- (b) if $a < m$ does not hold then the goal is $\text{returnSmallerNumber } m \ l <> m$. This holds by the induction hypothesis.

□

Lemma 9 (Soundness of hasData). $\forall p, \forall k, \text{hasData } p \ k = \text{true} \leftrightarrow \exists \text{data}, \text{dataAssignment } p(k) = \text{Some data}$. This Lemma states the soundness of hasData function, a function that with a port p and a natural number k returns true if p ’s data assignment in k equals Some data , where data is a data item of type data . The idea of this function is to return true if there is a valid data item at $\text{dataAssignment}(k)$.

Proof.

\Rightarrow

Let H a hypothesis, $H: \text{hasData } p \ k = \text{true}$, for a given p a port and k a natural number. The goal is to prove that $\exists \text{data}, \text{dataAssignment } p(k) = \text{Some data}$ holds. By unfolding hasData ’s definition, H is replaced by its definition, namely

$H: \text{match } (\text{dataAssignment } p(k)) \text{ with}$
 $\quad | \text{Some } a \Rightarrow \text{true}$
 $\quad | \text{None} \Rightarrow \text{false}$
 $\text{end} = \text{true}.$

By destructing dataAssignment , the current goal is split into two: each for each constructor of the return type of H ’s definition, option type . Therefore let H as

- (i) $true = true$. The goal is $\exists data0, Some\ a = Some\ data0$. It is important to denote that when destructing $dataAssignment$, in order to generate the first return as $Some\ a$, a was introduced in the proof context as a hypothesis $H1$: “let a be of type $data$ ”. By instantiating a as the existential quantifier \exists in the goal it is $Some\ a = Some\ a$ which holds.
- (ii) $false = true$. The goal is $\exists data0, None = Some\ data0$. As $false = true$ is a contradiction, this goal is discharged.

\Rightarrow

Let $H: \exists data, dataAssignment\ p(k) = Some\ data$. The goal is to prove that $hasData\ p\ k = true$ holds, given p is a port and k a natural number. From H , it is possible to derive that $H_a: dataAssignment\ p(k) = Some\ x$ holds, where x is a data item of type $data$. By unfolding $hasData$ in the current goal, it is

```
match (dataAssignment p(k)) with
| Some a  $\Rightarrow true$ 
| None  $\Rightarrow false$ 
end = true.
```

By rewriting H_a in the goal, $dataAssignment\ p(k)$ is replaced by $Some\ x$. The goal now is $true = true$ which holds.

□

Lemma 10 (Soundness of $evalCompositeDc$). $\forall s, \forall dca: DC, evalCompositeDc\ s\ dca = true \leftrightarrow dca = tDc \vee (\exists a, \exists b, dca = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, dca = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, dca = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, dca = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, dca = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, dca = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. The function $evalCompositeDc$ evaluates data constraints as formalized by the inductive type DC to boolean datatypes in Coq. This Lemma states that $evalCompositeDc$ returns true if one of the following is satisfied (let a being a port name and b a data item).

- dca is tDc , denoting a way to formalize data constraints that denotes $true$ (such data constraints are always satisfied),

- *dca* is *dc a b* and *evalDc (retrievePortFromInput s a b) = true*, denoting the idea of “the port *a* may have the data item *b*,”
- *dca* is *eqDc a b* and *eqDataPorts a b s = true*, denoting the idea of two ports having the same data item at the same time.
- *dca* is *andDc a b* and *evalCompositeDc s a && evalCompositeDc s b = true*, where && stands for the boolean conjunction formalized in Coq. This means that if the data constraint is a conjunction of data constraints formalized by means of *andDc*, both data constraints must evaluate to *true* in order to their conjunction also be *true*.
- *dca* is *orDc a b* and *evalCompositeDc s a || evalCompositeDc s b = true*. The notation *||* stands for Coq’s boolean disjunction. In other words, if the data constraint is a disjunction of other data constraint denoted by means of *orDc*, at least one of the data constraints that compose *orDc* may be true in order to their disjunction also be *true*.
- *dca* is *trDc tr a b*, where The aim of this data constraint is to encapsulate the notion of a data item of a port *a* with a function *tr* applied to it equals the data item in *b*.
- *dca* is *negDc a* and *negb (evalCompositeDc s a) = true*. The constructor *negDc* holds the idea of formalizing data constraint as the negation of other data constraint. As *negb* is Coq’s boolean negation, in order to *negDc* to be *true*, the boolean negation of the data constraint *negDc* encapsulates must be *true*.

Proof.

\Rightarrow

The proof is done by analysis on *dca*, destructing it on each of its possible constructors as defined by *DC*. Let $H: \text{evalCompositeDc } s \text{ dca} = \text{true}$. The goal is to prove $\text{dca} = \text{tDc} \vee (\exists a, \exists b, \text{dca} = \text{dc } a \text{ b} \wedge (\text{evalDC } (\text{retrievePortFromInput } s \text{ a}) \text{ b}) = \text{true}) \vee (\exists a, \exists b, \text{dca} = \text{eqDc } a \text{ b} \wedge \text{eqDataPorts } a \text{ b } s = \text{true}) \vee (\exists a, \exists b, \text{dca} = \text{andDc } a \text{ b} \wedge \text{evalCompositeDc } s \text{ a} \ \&\& \ \text{evalCompositeDc } s \text{ b} = \text{true}) \vee (\exists a, \exists b, \text{dca} = \text{orDc } a \text{ b} \wedge \text{evalCompositeDc } s \text{ a} \ || \ \text{evalCompositeDc } s \text{ b} = \text{true}) \vee (\exists a, \exists b, \exists tr, \text{dca} = \text{trDc } tr \text{ a b} \wedge \text{transformDC } tr \text{ a b } s = \text{true}) \vee (\exists a, \text{dca} = \text{negDc } a \wedge \text{negb } (\text{evalCompositeDc } s \text{ a}) = \text{true})$. By case analysis on *dca*, five different subgoals are generated, one for each constructor of *dca*’s inductive type. Let

- $dca = tDc$. The goal is $tDc = tDc \vee (\exists a, \exists b, tDc = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, tDc = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, tDc = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, tDc = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, tDc = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, tDc = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. The leftmost disjunction is $tDc = tDc$ which holds. Therefore the goal holds.
- $dca = dc\ n\ o$, where n is a port name and o is a data item of type *option data*. The goal is $dc\ n\ o = tDc \vee (\exists a, \exists b, dc\ n\ o = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, dc\ n\ o = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, dc\ n\ o = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, dc\ n\ o = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, dc\ n\ o = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, dc\ n\ o = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. As $dca = dc\ n\ o$, n is a port name and o is a data item, both within the proof context. By choosing the right part of the first disjunction, the goal is $(\exists a, \exists b, dc\ n\ o = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true)$. Therefore the existential variables referring to a and b in the goal can be discharged by instantiating a with n and b with o . The goal is $dc\ n\ o = dc\ n\ o \wedge (evalDC\ (retrievePortFromInput\ s\ n)\ o) = true$. From $dca = dc\ n\ o$, the hypothesis $H: evalCompositeDc\ s\ (dc\ n\ o) = true$. By simplification of H , it is $H: evalDC\ (retrievePortFromInput\ s\ n)\ o = true$. The goal being $dc\ n\ o = dc\ n\ o \wedge (evalDC\ (retrievePortFromInput\ s\ n)\ o) = true$ holds.
- $dca = eqDc\ n1\ n2$, where both $n1$ and $n2$ are port names of type *name*. The goal is $eqDc\ n1\ n2 = tDc \vee (\exists a, \exists b, eqDc\ n1\ n2 = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, eqDc\ n1\ n2 = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, eqDc\ n1\ n2 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, eqDc\ n1\ n2 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, eqDc\ n1\ n2 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, eqDc\ n1\ n2 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. As $dca = eqDc\ n1\ n2$ holds, both $n1$ and $n2$ in the proof context as port names also holds. By choosing the right part of the second disjunction, the

goal is $(\exists a, \exists b, eqDc\ n1\ n2 = eqDc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true)$. Therefore the existential variables referring to a and b in the goal can be discharged by instantiating a with $n1$ and b with $n2$. The goal is $eqDc\ n1\ n2 = eqDc\ n1\ n2 \wedge (evalDC\ (retrievePortFromInput\ s\ n)\ o) = true$. From $dca = eqDc\ n1\ n2$, the hypothesis H is $H: evalCompositeDc\ s\ (eqDc\ n1\ n2) = true$. By simplification of H , it is $H: eqDataPorts\ n1\ n2\ s = true$. The goal being $eqDc\ n1\ n2 = eqDc\ n1\ n2 \wedge (evalDC\ (retrievePortFromInput\ s\ n1)\ n2) = true$ which holds.

- $dca = andDc\ dca1\ dca2$, where both $dca1$ and $dca2$ are data constraint with type DC . The goal is $andDc\ dca1\ dca2 = tDc \vee (\exists a, \exists b, andDc\ dca1\ dca2 = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, andDc\ dca1\ dca2 = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, andDc\ dca1\ dca2 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\& \ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, andDc\ dca1\ dca2 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ || \ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, andDc\ dca1\ dca2 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, andDc\ dca1\ dca2 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. By choosing the proposition that is between the third and the fourth disjunction, the goal is $(\exists a, \exists b, andDc\ dca1\ dca2 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\& \ evalCompositeDc\ s\ b = true)$. H is $H: evalCompositeDc\ s\ (andDc\ dca1\ dca2) = true$, from $dca = andDc\ dca1\ dca2$. By simplifying H , it is $H: evalCompositeDc\ s\ dca1 \ \&\& \ evalCompositeDc\ s\ dca2 = true$. As $dca1$ and $dca2$ are valid constants of type DC , the quantifiers upon a and b can be eliminated with a being $dca1$ and b being $dca2$. Therefore, the goal is $andDc\ dca1\ dca2 = andDc\ dca1\ dca2 \wedge evalCompositeDc\ s\ dca1 \ \&\& \ evalCompositeDc\ s\ dca2 = true$. The left proposition of the conjunction $andDc\ dca1\ dca2 = andDc\ dca1\ dca2$ holds and the right proposition holds from H , proving the current goal.
- $dca = orDc\ dca1\ dca2$, both $dca1$ and $dca2$ are data constraint with type DC . The goal is $orDc\ dca1\ dca2 = tDc \vee (\exists a, \exists b, orDc\ dca1\ dca2 = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, orDc\ dca1\ dca2 = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, orDc\ dca1\ dca2 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\& \ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, orDc\ dca1\ dca2 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ || \ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, orDc\ dca1\ dca2 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s =$

$true) \vee (\exists a, orDc\ dca1\ dca2 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. By choosing the proposition that is between the fourth and the fifth disjunction, the goal is $(\exists a, \exists b, orDc\ dca1\ dca2 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true)$. H is $H: evalCompositeDc\ s\ (orDc\ dca1\ dca2) = true$, from $dca = orDc\ dca1\ dca2$. By simplifying H , it is $H: evalCompositeDc\ s\ dca1 \ \&\&\ evalCompositeDc\ s\ dca2 = true$. As $dca1$ and $dca2$ are valid constants of type DC , the quantifiers upon a and b can be eliminated with a being $dca1$ and b being $dca2$. Therefore, the goal is $orDc\ dca1\ dca2 = orDc\ dca1\ dca2 \wedge evalCompositeDc\ s\ dca1 \ \&\&\ evalCompositeDc\ s\ dca2 = true$. The left proposition of the conjunction $orDc\ dca1\ dca2 = orDc\ dca1\ dca2$ holds and the right proposition holds from H . Hence the goal holds.

- $dca = trDc\ o\ n\ n0$, where o is a function $option\ data \rightarrow option\ data$ and both n and $n0$ are port names of type $name$. The goal is $trDc\ o\ n\ n0 = tDc \vee (\exists a, \exists b, trDc\ o\ n\ n0 = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, trDc\ o\ n\ n0 = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b, trDc\ o\ n\ n0 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, trDc\ o\ n\ n0 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists a, \exists b, \exists tr, trDc\ o\ n\ n0 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists a, trDc\ o\ n\ n0 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. By choosing the proposition that is between the fifth and the sixth disjunction, the goal is $(\exists a, \exists b, \exists tr, trDc\ o\ n\ n0 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true)$. H is $H: evalCompositeDc\ s\ (trDc\ o\ n\ n0) = true$, from $dca = trDc\ o\ n\ n0$. By simplifying H , it is $H: transformDCConn0s = true$. As o is a function $option\ data \rightarrow option\ data$, n and $n0$ are valid port names, the quantifiers upon a and b can be eliminated with a being n and b being $n0$ and the quantifier over tr can be eliminated with o . Therefore, the goal is $trDc\ o\ n\ n0 = trDc\ o\ n\ n0 \wedge transformDC\ o\ n\ n0\ s = true$. The left proposition of the conjunction $trDc\ o\ n\ n0 = trDc\ o\ n\ n0$ holds and the right proposition holds from H . Hence the goal holds.
- $dca = negDc\ dca1$ where $dca1$ is a data constraint of type DC . Then H is $H: evalCompositeDcs(negDc\ dca1) = true$. The goal is $negDc\ dca1 = tDc \vee (\exists a, \exists b, negDc\ dca1 = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists a, \exists b, negDc\ dca1 = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists a, \exists b,$

$negDc\ dca1 = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true)$
 $\vee (\exists\ a, \exists\ b, negDc\ dca1 = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true)$
 $\vee (\exists\ a, \exists\ b, \exists\ tr, negDc\ dca1 = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true)$
 $\vee (\exists\ a, negDc\ dca1 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. By choosing the rightmost proposition at the last disjunction, the goal is $\exists\ a, negDc\ dca1 = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true$. By simplification of H , it is $H: negb(evalCompositeDc\ s\ dca1) = true$. The existential quantifier on a can be eliminated with $dca1$. The goal is $negDc\ dca1 = negDc\ dca1 \wedge negb\ (evalCompositeDc\ s\ dca1) = true$. The left side of the goal holds and the right side of the goal holds from H . Therefore the goal holds.

\Leftarrow

Suppose $H: dca = tDc \vee (\exists\ a, \exists\ b, dca = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true) \vee (\exists\ a, \exists\ b, dca = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true) \vee (\exists\ a, \exists\ b, dca = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true) \vee (\exists\ a, \exists\ b, dca = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true) \vee (\exists\ a, \exists\ b, \exists\ tr, dca = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true) \vee (\exists\ a, dca = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true)$. The goal is $evalCompositeDc\ s\ dca = true$. As H holds, at least one of the disjunction in H must hold. By analyzing H , let it be

- $dca = tDc$. The goal is $evalCompositeDc\ s\ dca = true$. By replacing dca with tDc in the goal, it is $evalCompositeDc\ s\ dca = true$ which holds by $evalCompositeDc$'s definition.
- $\exists\ a, \exists\ b, dca = dc\ a\ b \wedge evalDC\ (retrievePortFromInput\ s\ a)\ b = true$. The goal is $evalCompositeDc\ s\ dca = true$. From H it is possible to derive a port name a and a data item b where H holds. Then let a and b be a port name and a data item, respectively. H now is $dca = dc\ a\ b \wedge (evalDC\ (retrievePortFromInput\ s\ a)\ b) = true$. From H , let $H1: dca = dc\ a\ b$ and $H2: evalDC\ (retrievePortFromInput\ s\ a)\ b = true$. By rewriting $H1$'s right hand side in the goal, i.e., replacing dca with $dc\ a\ b$, it is $evalCompositeDc\ s\ (dc\ a\ b) = true$. By simplification, the goal is $evalDC\ (retrievePortFromInput\ s\ a)\ b = true$ which holds from $H2$.
- $\exists\ a, \exists\ b, dca = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true$. The goal is $evalCompositeDc\ s\ dca = true$. H allows one to derive two port names of type *port*, denoted by $a\ b$

- respectively. H now is $dca = eqDc\ a\ b \wedge eqDataPorts\ a\ b\ s = true$. By eliminating the conjunction in H , let $H1: dca = eqDc\ a\ b$ and $H2: eqDataPorts\ a\ b\ s = true$. By rewriting $H1$ in the goal, it is $evalCompositeDc\ s\ (eqDc\ a\ b) = true$. By simplification of the goal, it is $eqDataPorts\ a\ b\ s = true$ which holds from $H2$.
- $\exists\ a, \exists\ b, dca = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true$. The goal is $evalCompositeDc\ s\ dca = true$. H allows one to derive two data constraints of type DC denoted by $a\ b$ respectively. H now is $dca = andDc\ a\ b \wedge evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true$. By eliminating the conjunction in H , let $H1: dca = andDc\ a\ b$ and $H2: evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true$. By rewriting $H1$ in the goal, it is $evalCompositeDc\ s\ (andDc\ a\ b) = true$. By simplification of the goal, it is $evalCompositeDc\ s\ a \ \&\&\ evalCompositeDc\ s\ b = true$ which holds from $H2$.
 - $\exists\ a, \exists\ b, dca = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true$. The goal is $evalCompositeDc\ s\ dca = true$. H allows one to derive two data constraints of type DC denoted by a and b respectively. H now is $dca = orDc\ a\ b \wedge evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true$. By eliminating the conjunction in H , let $H1: dca = orDc\ a\ b$ and $H2: evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true$. By rewriting $H1$ in the goal, it is $evalCompositeDc\ s\ (orDc\ a\ b) = true$. By simplification of the goal, it is $evalCompositeDc\ s\ a \ ||\ evalCompositeDc\ s\ b = true$ which holds from $H2$.
 - $\exists\ a, \exists\ b, \exists\ tr, dca = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true$. The goal is $evalCompositeDc\ s\ dca = true$. H allows one to derive a function tr of type $option\ data \rightarrow option\ data$ and two port names of type $port$ denoted by a and b respectively. H now is $dca = trDc\ tr\ a\ b \wedge transformDC\ tr\ a\ b\ s = true$. By eliminating the conjunction in H , let $H1: dca = trDc\ tr\ a\ b$ and $H2: transformDC\ tr\ a\ b\ s = true$. By rewriting $H1$ in the goal, it is $evalCompositeDc\ s\ (trDc\ tr\ a\ b) = true$. By simplification of the goal, it is $transformDC\ tr\ a\ b\ s = true$ which holds from $H2$.
 - $\exists\ a, dca = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true$. The goal is $evalCompositeDc\ s\ dca = true$. From H , one can derive a data constraint a . Then H is $dca = negDc\ a \wedge negb\ (evalCompositeDc\ s\ a) = true$. By eliminating the conjunction

in H , let $H1$: $dca = \text{negDc } a$ and $H2$: $\text{negb } (\text{evalCompositeDc } s \ a) = \text{true}$. By rewriting $H1$, the goal is $\text{evalCompositeDc } s \ (\text{negDc } a) = \text{true}$. By simplification on the goal, it is $\text{negb } (\text{evalCompositeDc } s \ a) = \text{true}$ which holds from $H2$.

□

Lemma 11 (Soundness of *getAllThetaTimes*). $\forall s: \text{set port}, \text{getAllThetaTimes } s \neq [] \leftrightarrow \exists a, \text{In } a \ s$. The function *getAllThetaTimes* is a function that with a set of ports s returns all values of ports at a given step k that are candidate to be the $\theta.\text{time}(k)$. Therefore this function always return a nonempty set, unless s is empty.

⇒

Proof. The proof is done by destructing s , generating a goal for each constructor of the inductive type s belongs to. Suppose H : $\text{getAllThetaTimes } s <> []$. Hence let s be

- an empty set $[]$. H is H : $\text{getAllThetaTimes } [] <> []$. the goal is $\exists a, \text{In } a \ []$. By simplification on H , it is H : $[] <> []$, which is a contradiction. Therefore, this goal is discharged.
- a set with elements $(p :: s)$, where p is a port and s is a set of ports. The notation $::$ is the usual operation of appending an element to a set in Coq. H is H : $\text{getAllThetaTimes}(p :: s) <> []$. The goal is $\exists a, \text{In } a \ (p :: s)$. As p is a port, the existential quantifier on the variable a in the goal can be eliminated by instantiating p . The goal is $\text{In } p \ (p :: s)$, which by simplification unfolds to $p = p \vee \text{In } p \ s$ where the left side of this disjunction holds.

⇐

Let H : $\exists a, \text{In } a \ s$. The goal is $\text{getAllThetaTimes } s \neq []$. The proof is straightforward from *getAllThetaTimes*'s definition: given that there is an element in s , it is not empty. By *getAllThetaTimes*'s definition, it only returns an empty set if the given set s is empty. Therefore, the goal is proved.

□

Lemma 12 (Soundness of *timeStampEqThetaTimeSound*). $\forall s, \forall k, \forall a, \text{timeStampEqThetaTime } s \ k \ a = \text{true} \leftrightarrow ((\text{timeStamp } a(\text{index } a) =? \ \text{thetaTime } (s) \ (k)) = \text{true})$. The function *timeStampEqThetaTime* verifies whether a port denoted by a is in $\theta.\text{time}(k)$.

Proof.

\Rightarrow

Let $H: \text{timeStampEqThetaTime} \text{ ca } a = \text{true}$. The goal is to prove $((\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } (s) (k)) = \text{true})$. The proof proceeds by unfolding $\text{timeStampEqThetaTime}$ in H . H is $H: (\text{if equiv_dec } (\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } s k) \text{ true then true else false}) = \text{true}$. By analyzing equiv_dec in H , there are two possibilities where for each the current goal must hold.

- (i) let $(\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } s k) = \text{true}$. A new hypothesis $H_2: (\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } s k) = \text{true}$ is added to the proof context. H unfolds to $H: \text{true} = \text{true}$. The goal is $((\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } (s) (k)) = \text{true})$ which holds from H_2 .
- (ii) let $(\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } s k) \neq \text{true}$. H is $H: \text{false} = \text{true}$. The goal is $((\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } (s) (k)) = \text{true})$. As H is a contradiction, this goal is discharged.

\Leftarrow

Let $H: (\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } (s) (k)) = \text{true}$. The aim is to prove $\text{timeStampEqThetaTime } s k a = \text{true}$. By unfolding $\text{timeStampEqThetaTime}$ in the goal, it is $(\text{if equiv_dec } (\text{timeStamp } a(\text{index } a) =? \text{ thetaTime } s k) \text{ true then true else false}) = \text{true}$. By rewriting H in the goal it unfolds to $(\text{if equiv_dec } (\text{true}) \text{ true then true else false}) = \text{true}$. By simplification, the goal unfolds to $\text{true} = \text{true}$ which holds.

□

Lemma 13 (Soundness of thetaN). $\forall \text{ ca}, \forall k, \forall s, \text{thetaN } \text{ ca } k s \neq [] \leftrightarrow$

$(\exists a, \text{In } a s \wedge \text{hasData } a(k) = \text{true} \wedge \text{timeStampEqThetaTime } \text{ ca } k a = \text{true})$. The function thetaN builds $\theta.N(k)$ by verifying every port $p \in s$ has data $\theta.time(k)$, where s is a set of ports. Therefore, thetaN will not be an empty set if there is at least one port $p \in s$ that has data in $\theta.time(k)$ and its time stream associated with this data item is in $\theta.time(k)$.

Proof.

\Rightarrow

Suppose that $H: \text{thetaN } \text{ ca } k s \neq []$, where ca and s are port sets, k a natural number which is the current step of the run in the automaton. The objective is to prove

that $(\exists a, In\ a\ s \wedge hasData\ a\ (k) = true \wedge timeStampEqThetaTime\ ca\ k\ a = true)$ holds. The proof proceeds by induction on the set of ports s . Let

- (i) Induction Basis: let s be an empty set. H is $H: thetaN\ ca\ k\ [] \neq []$. The goal is $(\exists a, In\ a\ s \wedge hasData\ a\ (k) = true \wedge timeStampEqThetaTime\ ca\ k\ a = true)$. By simplification on H , it is $[] \neq []$. As H is a contradiction, this goal is discharged.
- (ii) Induction Hypothesis: suppose the Induction Basis holds for an arbitrary non empty s . This means to suppose that $thetaN\ ca\ k\ s \neq [] \rightarrow (\exists a, In\ a\ s \wedge hasData\ a\ (k) = true \wedge timeStampEqThetaTime\ ca\ k\ a = true)$.
- (iii) Inductive Step: The goal is to prove that $(\exists a_0, In\ a_0\ a::s \wedge hasData\ a_0(k) = true \wedge timeStampEqThetaTime\ ca\ k\ a_0 = true)$. In other words, the idea is to prove that the property holds for a set $a::s$, which is s with another element a added. Let IH be the Induction Hypothesis. H is $H: thetaN\ ca\ k\ a::s \neq []$. Upon simplification on H , it is $H: (if\ hasData\ a\ k\ then\ if\ equiv_dec\ (timeStampEqThetaTime\ ca\ k\ a)\ true\ then\ id\ a\ ::\ thetaN\ ca\ k\ s\ else\ thetaN\ ca\ k\ s\ else\ thetaN\ ca\ k\ s) \neq []$. By analyzing H , by destructing $hasData$, it generates two equal subgoals, each for each constructor of the return type of $hasData$ (which is *bool*):
 - (a) $hasData\ a\ k = true$, introducing a new hypothesis H_2 , $H_2: hasData\ a\ k = true$ is introduced. H is $H: (if\ equiv_dec\ (timeStampEqThetaTime\ ca\ k\ a)\ true\ then\ id\ a\ ::\ thetaN\ ca\ k\ s\ else\ thetaN\ ca\ k\ s) \neq []$. From H , by destructing $equiv_dec$, there are two possible cases, each associated with the same goal:
 - (1) $timeStampEqThetaTime\ ca\ k\ a = true$. A new hypothesis H_3 is introduced in the proof context, where $H_3: timeStampEqThetaTime\ ca\ k\ a = true$ is introduced. H is $H: id\ a\ ::\ thetaN\ ca\ k\ s \neq []$. The goal is $\exists a_0, In\ a_0\ a::s \wedge hasData\ a_0\ (k) = true \wedge timeStampEqThetaTime\ ca\ k\ a_0 = true$. The existential quantifier on the variable a_0 is eliminated by instantiating a . The goal is now $In\ a\ a::s \wedge hasData\ a\ (k) = true \wedge timeStampEqThetaTime\ ca\ k\ a = true$. The leftmost part of the goal holds because a is in the set given as predicate to In , the predicate that states that a element is inside a set. $hasData\ a\ (k) = true$ holds by H_2 , while $timeStampEqThetaTime\ ca\ k\ a = true$ holds from H_3 . Therefore the goal holds.

- (2) $\text{timeStampEqThetaTime } ca \ k \ a) \neq \text{true}$. H_3 is introduced now as H_3 : $\text{timeStampEqThetaTime } ca \ k \ a) \neq \text{true}$. H is $\text{thetaN } ca \ k \ s \neq []$. By using IH on H , H is $\exists a, \text{In } a \ s \wedge \text{hasData } a \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ a = \text{true}$. The goal is $(\exists a_0, \text{In } a_0 \ a::s \wedge \text{hasData } a_0(k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ a_0 = \text{true})$. From H , it is possible to deduce that exists $x, x \neq a$ a port in which $H: \text{In } x \ s \wedge \text{hasData } x \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ x = \text{true}$. Therefore the existential quantifier in the goal may be eliminated by x . The goal is $\text{In } x \ a::s \wedge \text{hasData } x \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ x = \text{true}$. As the predicate In in the leftmost part of the goal unfolds to $x = a \vee \text{In } x \ s$, the goal holds from H .
- (b) $\text{hasData } a \ k \neq \text{true}$. H is $\text{thetaN } ca \ k \ s \neq []$. By applying IH on H , it is $\exists a, \text{In } a \ s \wedge \text{hasData } a \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ a = \text{true}$. The goal is $(\exists a_0, \text{In } a_0 \ a::s \wedge \text{hasData } a_0(k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ a_0 = \text{true})$. From H , one can derive a port x where $H: \text{In } x \ s \wedge \text{hasData } x \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ x = \text{true}$. Therefore the existential quantifier on the variable a_0 in the goal can be eliminated with x . The goal is $\text{In } x \ a::s \wedge \text{hasData } x \ (k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ x = \text{true}$. As the predicate In in the leftmost part of the goal unfolds to $x = a \vee \text{In } x \ s$, the goal holds directly from H .

\Leftarrow

Suppose $H: (\exists a, \text{In } a \ a::s \wedge \text{hasData } a(k) = \text{true} \wedge \text{timeStampEqThetaTime } ca \ k \ a = \text{true})$. The objective is to prove that $\text{thetaN } ca \ k \ s \neq []$. With H , this proof is straightforward from thetaN 's definition in which, by unfolding thetaN , in order to thetaN not return an empty set, s must be non-empty and there is at least a port $a \in s$ where s is the port set given to thetaN that has data in $\theta.time(k)$ and its time stream at k is in $\theta.time(k)$.

□

Lemma 14 (Soundness of *derivative*). $\forall p, \text{derivative } p = \text{mkport } (id \ p) \ (\text{dataAssignment } p) \ (\text{timeStamp } p) \ (\text{portCond } p) \ (S(\text{index } p))$. The function *derivative* with a port p returns p 's derivative, which in the context of CACoq is p 's index plus 1.

Proof. The proof is straightforward from *derivative*'s definition. With any port p , *derivative* returns a port o where all fields of o are the same of p , only o 's index is p 's index plus 1. \square

Lemma 15 (Soundness of *derivativePortInvolved*). $\forall s, \forall a, \text{derivativePortInvolved } s \ a = [\text{derivative}(a)] \leftrightarrow (\exists x, \text{In } x \ s \wedge x = \text{id } a)$. The function *derivativePortInvolved* is a function that given a set of port names s and a port a returns a 's derivative (by means of *derivative*) if a 's name is in s , or returning a otherwise.

\Rightarrow

Proof. Suppose $H: \text{derivativePortInvolved } s \ a = [\text{derivative}(a)]$. The objective is to prove $\exists x, \text{In } x \ s \wedge x = \text{id } a$. The proof proceeds by induction on the port set s .

- (i) Inductive Basis: let s be an empty set. H is $H: \text{derivativePortInvolved } [] \ a = [\text{derivative}(a)]$. The goal is to prove $\exists x, \text{In } x \ s \wedge x = \text{id } a$. By *derivativePortInvolved*'s definition, H is a contradiction. Therefore the current goal is discharged.
- (ii) Induction Hypothesis: suppose the Inductive Basis holds for an arbitrary set of ports s , i.e., suppose a hypothesis $IH: \text{derivativePortInvolved } s \ a = [\text{derivative}(a)] \rightarrow \exists x, \text{In } x \ s \wedge x = \text{id } a$.
- (iii) Inductive Step: Suppose $H: \text{derivativePortInvolved } b::s \ a = [\text{derivative}(a)]$, where b is a port. The objective is to prove that $\exists x, \text{In } x \ b::s \wedge x = \text{id } a$ holds. By simplification on H , it is $H: (\text{if equiv_dec } b \ \text{id } a \ \text{then } [\text{derivative}(a)] \ \text{else } \text{derivativePortInvolved } s \ a) = [\text{derivative}(a)]$. By destructing *equiv_dec* in H , two equal goals are generated, each for each possible return of *equiv_dec*. Then, let *equiv_dec* yield
 - (a) $b = \text{id } a$. The goal is $\exists x, \text{In } x \ b::s \wedge x = \text{id } a$. A new hypothesis H_2 is introduced in the proof context as this result, namely, $H_2: b = \text{id } a$. H is $H: [\text{derivative}(a)] = [\text{derivative}(a)]$. The existential quantifier in the goal over the variable x can be eliminated by instantiating b . The goal is $\text{In } b \ b::s \wedge b = \text{id } a$. The left side of the conjunction in the goal holds by *In*'s definition, which by simplification evaluates to $b = b \vee \text{In } b \ s$, which holds. The right side of the disjunction is $b = \text{id } a$ which holds from H_2 . Therefore the goal holds.

(b) $b \neq id\ a$. The goal is $\exists x, In\ x\ b::s \wedge x = id\ a$. The hypothesis H is $H: derivativePortInvolved\ s\ a = [derivative(a)]$. By applying IH to H , H is $H: \exists x, In\ x\ s \wedge x = id\ a$. Upon destructing H , one can derive that exists a port name c by introducing the existential quantifier in H . H is $H: In\ c\ s \wedge c = id\ a$. Therefore the existential quantifier in the goal may be eliminated with c , the newly port name obtained from H . The goal is $In\ c\ b::s \wedge c = id\ a$. The left side of the disjunction holds by unfolding In which evaluates to $c = b \vee In\ c\ s$, where the latter holds from H . The right side of the disjunction holds straightforward from H . Therefore the goal holds.

\Rightarrow

Suppose $H: \exists x, In\ x\ s \wedge x = id\ a$. The objective is to prove that $derivativePortInvolved\ s\ a = [derivative(a)]$ holds. By $derivativePortInvolved$'s definition, it returns $[derivative(a)]$ if there exists a port name x where $x \in s$ and $x = id\ a$. From H , one can derive such port name by introducing the existential quantifier in H . H is $H: In\ x\ s \wedge x = id\ a$. By unfolding $derivativePortInvolved$'s definition, the goal holds straightforward from its definition. \square

Lemma 16 (Soundness of *allDerivativesFromPortsInvolved*). $\forall\ names, \forall\ ports, allDerivativesFromPortsInvolved\ names\ ports = flat_map\ (derivativePortInvolved\ names)\ ports$. The function *allDerivativesFromPortsInvolved* applies *derivativePortInvolved* with a set of name ports denoted by *names* on a set of ports denoted by *ports* by means of *flat_map*, yielding a set containing all ports $p \in ports$ filtered by *derivativePortInvolved*.

Proof. This proof holds straightforward from *allDerivativesFromPortsInvolved*'s definition. Upon unfolding, it is *flat_map (derivativePortInvolved names) ports*. \square

Lemma 17 (Soundness of *portsOutsideTransition*). $\forall\ input, \forall\ ports, portsOutsideTransition\ input\ ports = false \leftrightarrow \exists\ b, In\ b\ ports \wedge id\ input = b$. The function *portsOutsideTransition* with a port denoted by *input* and a set of port names *ports* returns *false* if there is not a port name $p \in ports$, $p = id\ input$ (*id* is the field of the record *port* that denotes a port name), otherwise returning *true*.

Proof.

\Rightarrow

Let $H: \text{portsOutsideTransition input ports} = \text{false}$. The objective is to prove that $\exists b, \text{In } b \text{ ports} \wedge \text{id input} = b$ holds. The proof proceeds by induction on the set of ports ports .

(i) Induction Basis: let ports be an empty set, denoted by $[]$. H is

$H: \text{portsOutsideTransition input } [] = \text{false}$. The goal is $\exists b, \text{In } b [] \wedge \text{id input} = b$.

From $\text{portsOutsideTransition}$'s definition, if ports is an empty set, $\text{portsOutsideTransition}$ yields true . By simplification, H is $H: \text{true} = \text{false}$ which is a contradiction. Therefore the current goal is discharged.

(ii) Induction Hypothesis: suppose $IH: \text{portsOutsideTransition input ports} = \text{false} \rightarrow \exists b, \text{In } b \text{ ports} \wedge \text{id input} = b$ holds for a nonempty arbitrary set of port names ports .

(iii) Inductive Step: the aim is to prove that the Induction Hypothesis holds for a set s with an element added to it. Let $H: \text{portsOutsideTransition input } a::\text{ports} = \text{false}$, where a is a port name. The objective is to prove $\exists b, \text{In } b a::\text{ports} \wedge \text{id input} = b$. By simplification on H , it unfolds to $(\text{if id input} \neq a \text{ then portsOutsideTransition input ports else false}) = \text{false}$. From the inequality \neq in H , two possible situations can be derived, each generating the same goal to be proved.

(a) Let $\text{id input} \neq a$. A new hypothesis $H_2: \text{id input} \neq a$ is added to the proof context. The goal is $\exists b, \text{In } b a::\text{ports} \wedge \text{id input} = b$. H is $H: \text{portsOutsideTransition input ports}$. By applying IH on H , H is $\exists b, \text{In } b \text{ ports} \wedge \text{id input} = b$. From H , a port x can be derived, by introducing the existential quantifier in H , where $H: \text{In } x \text{ ports} \wedge \text{id input} = x$. The existential quantifier in the goal over the variable b can be eliminated with x . The goal is $\text{In } x a::\text{ports} \wedge \text{id input} = x$. The left side of the conjunction in the goal holds from $H: \text{In } x a::\text{ports}$ evaluates to $x = a \vee \text{In } x \text{ ports}$, where the right side of this disjunction holds directly from H . The right side of the goal holds directly from H . The goal is proved.

(b) Let $\text{id input} = a$. The goal is $\exists b, \text{In } b a::\text{ports} \wedge \text{id input} = b$. H_2 now is $H_2: \text{id input} = a$. H is $H: \text{false} = \text{false}$. The existential quantifier in the goal over the variable b may be eliminated with a . The goal is $\text{In } a a::\text{ports} \wedge \text{id input} = a$. The left side of the conjunction in the goal holds from $H: \text{In } x$

$a::ports$ evaluates to $a = a \vee In\ a\ ports$, where the left side of this disjunction holds. The right side of the goal holds directly from H . The goal is proved.

\Leftarrow

Let $ports$ be a set of port names and $input$ a port. Suppose $H: \exists b, In\ b\ ports \wedge id\ input = b$. The objective is to prove that $portsOutsideTransition\ input\ ports = false$ holds. The proof proceeds by induction on $ports$.

- (i) Induction Basis: let $H: \exists b, In\ b\ [] \wedge id\ input = b$. The goal is $portsOutsideTransition\ input\ [] = false$. From H , the contradiction $In\ b\ []$ can be derived, which states that b is in the empty set, denoted by $[]$. The goal is discharged.
- (ii) Induction Hypothesis: suppose $IH: \exists b, In\ b\ ports \wedge id\ input = b \rightarrow portsOutsideTransition\ input\ ports = false$ holds for a nonempty arbitrary set of port names, depicted by $ports$.
- (iii) Inductive Step: the objective is to prove that $portsOutsideTransition\ input\ a::ports = false$, where a is a port name. H is $H: \exists b, In\ b\ a::ports \wedge id\ input = b$. By simplification on the goal, it unfolds to $(if\ id\ input \neq a\ then\ portsOutsideTransition\ input\ ports\ else\ false) = false$. Upon analyzing the inequality in the goal, one may obtain two possible goals, each for each derivable case of this analysis.
 - (a) let $id\ input \neq a$. the goal is $portsOutsideTransition\ input\ ports = false$. A new hypothesis $H_2: id\ input \neq a$ is added to the proof context. By applying IH to the goal, it is $\exists b, In\ b\ ports \wedge id\ input = b$. From H , by introducing the existential quantifier on the variable b , a new port x can be derived. H is $In\ x\ a::ports \wedge id\ input = x$. By simplification on H , it unfolds to $(x = a \vee In\ x\ ports) \wedge id\ input = x$. The left side of the conjunction in H lets one derive the following.
 - i. if $x = a$, $H_3: id\ input = a$ also holds. H is $x = a \wedge id\ input = x$. The goal is $\exists b, In\ b\ ports \wedge id\ input = b$. As H_3 contradicts H_2 , this goal is discharged.
 - ii. if $In\ x\ ports$ holds, $H_3: id\ input = x$ also holds. The goal is $\exists b, In\ b\ ports \wedge id\ input = b$. H is $In\ x\ ports \wedge id\ input = x$. The existential quantifier

in the goal may be eliminated with x . The goal is $\text{The goal is } \textit{In } x \textit{ ports} \wedge \textit{id input} = x$. The goal holds directly from H .

(b) let $\textit{id input} = a$. The goal is $\textit{false} = \textit{false}$ which holds.

□

Lemma 18 (Soundness of *retrievePortsOutsideTransition*). $\forall \textit{input}, \forall \textit{ports}, \textit{retrievePortsOutsideTransition input ports} \neq [] \leftrightarrow \exists a, \textit{portsOutsideTransition a ports} = \textit{true} \wedge \textit{In a input}$. The function *retrievePortsOutsideTransition* with a set of ports *input* and a set of port names *ports* returns all ports $p \in \textit{input}$ where p 's name is not in *ports* by means of *portsOutsideTransition*.

Proof.

\Rightarrow

Suppose $H: \textit{retrievePortsOutsideTransition input ports} \neq []$. The aim is to prove $\exists a, \textit{portsOutsideTransition a ports} = \textit{true} \wedge \textit{In a input}$. The proof proceeds by induction on *input*.

(i) Induction Basis: let *input* be an empty set denoted by $[]$.

H is $H: \textit{retrievePortsOutsideTransition [] ports} \neq []$. The goal is $\exists a, \textit{portsOutsideTransition a ports} = \textit{true} \wedge \textit{In a []}$. By simplification of H , it is $[] \neq []$ which is a contradiction. Therefore this goal is discharged.

(ii) Induction Hypothesis: suppose $IH: \textit{retrievePortsOutsideTransition input ports} \neq [] \rightarrow \exists a, \textit{portsOutsideTransition a ports} = \textit{true} \wedge \textit{In a input}$, for an arbitrary nonempty set of ports *input*.

(iii) Inductive Step: let $H: \textit{retrievePortsOutsideTransition a::input ports} \neq []$. The objective is to prove $\exists a_0, \textit{portsOutsideTransition a}_0 \textit{ ports} = \textit{true} \wedge \textit{In a}_0 a::\textit{input}$. By simplifying H , it is $(\textit{if equiv_dec (portsOutsideTransition a ports) true then a a::retrievePortsOutsideTransition input ports else retrievePortsOutsideTransition input ports}) \neq []$. By analyzing *equiv_dec* in H , a goal is derived for each possible return of *equiv_dec*'s destruction.

(a) let $\textit{portsOutsideTransition a ports} = \textit{true}$. A new hypothesis

$H_2: \textit{portsOutsideTransition a ports} = \textit{true}$ is added in the proof context. H is a

$:: \text{retrievePortsOutsideTransition input ports} \neq []$. The goal is $\exists a_0, \text{portsOutsideTransition } a_0 \text{ ports} = \text{true} \wedge \text{In } a_0 a::\text{input}$. The existential quantifier in the goal over a_0 can be eliminated with a . The goal is $\text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a a::\text{input}$. The left part of the conjunction in the goal holds from H_2 . The right part of the goal unfolds to $a = a \neq \text{In } a \text{ input}$ where the left side of this disjunction holds. Therefore the goal holds.

(b) let $\text{portsOutsideTransition } a \text{ ports} \neq \text{true}$. A new hypothesis

$H_2: \text{portsOutsideTransition } a \text{ ports} \neq \text{true}$ is added to the proof context. H is $H: \text{retrievePortsOutsideTransition input ports}$. The goal is $\exists a_0, \text{portsOutsideTransition } a_0 \text{ ports} = \text{true} \wedge \text{In } a_0 a::\text{input}$. By applying IH on H , H is $H: \exists a, \text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a \text{ input}$. From H , it is possible to derive a port x by introducing the existential quantifier. Therefore, $H: \text{portsOutsideTransition } x \text{ ports} = \text{true} \wedge \text{In } x \text{ input}$. The existential quantifier in the goal can be eliminated by instantiating x . The goal is $\text{portsOutsideTransition } x \text{ ports} = \text{true} \wedge \text{In } x a::\text{input}$. The left part of the goal holds directly from H . The right part of the goal evaluates to $a = x \neq \text{In } x \text{ input}$ where the right side of this disjunction holds. The goal is proved.

\Leftarrow

Suppose $H: \exists a, \text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a \text{ input}$. The objective is to prove $\text{retrievePortsOutsideTransition input ports} \neq []$. The proof proceeds by induction on the set of ports denoted by input .

- (i) Induction Basis: let ports be an empty set. H is $H: \exists a, \text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a []$. The goal is to prove $\text{retrievePortsOutsideTransition input } [] \neq []$. From H , an hypothesis $H_2: \text{In } a []$ can be derived. As H_2 is a contradiction, this goal is discharged.
- (ii) Induction Hypothesis: suppose $IH: \exists a, \text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a \text{ input} \rightarrow \text{retrievePortsOutsideTransition input ports} \neq []$ holds for a nonempty arbitrary set of port names ports .
- (iii) Inductive Step: The aim is to prove $\text{portsOutsideTransition input } (a :: \text{ports}) = \text{false}$. Let $H: \exists a_0, \text{portsOutsideTransition } a_0 \text{ ports} = \text{true} \wedge \text{In } a_0 a :: \text{input}$. By

simplification of the goal, it unfolds to $(\text{if } \text{equiv_dec } (\text{portsOutsideTransition } a \text{ ports}) \text{ true then } a :: \text{retrievePortsOutsideTransition input ports else retrievePortsOutsideTransition input ports}) \neq []$. Upon destructing equiv_dec , there are two possibilities:

- (a) let $\text{portsOutsideTransition } a \text{ ports} = \text{true}$. The goal is $a :: \text{retrievePortsOutsideTransition input ports}$. As a added to a set denoted by the return of $\text{retrievePortsOutsideTransition input ports}$ is a nonempty set (it contains at least a as an element), this goal holds.
- (b) Let $\text{portsOutsideTransition } a \text{ ports} \neq \text{true}$. Therefore a new hypothesis H_2 is added to the proof context, where $H_2: \text{portsOutsideTransition } a \text{ ports} \neq \text{true}$. The goal is $\text{retrievePortsOutsideTransition input ports}$. By applying the induction hypothesis IH on the goal, it is $\exists a, \text{portsOutsideTransition } a \text{ ports} = \text{true} \wedge \text{In } a \text{ input}$. H lets one derive a port x by eliminating the existential quantifier on a_0 . H is $H: \text{portsOutsideTransition } x \text{ ports} = \text{true} \wedge \text{In } x (a :: \text{input})$. Therefore the existential quantifier on the goal can be eliminated with x . The goal is $\text{portsOutsideTransition } x \text{ ports} = \text{true} \wedge \text{In } x \text{ input}$. The left part of the disjunction in the goal holds directly from H . The right part can be derived from H by $\text{In } x (a :: \text{input})$, which by simplification unfolds to $x = a \vee \text{In } x \text{ input}$. Let
 - i. $x = a$. The goal is $\text{In } x \text{ input}$. As from $H \text{ portsOutsideTransition } x \text{ ports} = \text{true}$ holds, if $x = a$, from H and from H_2 , both $\text{portsOutsideTransition } x \text{ ports} = \text{true}$ and $\text{portsOutsideTransition } a \text{ ports} \neq \text{true}$ holds, which is a contradiction. Therefore this goal is discharged.
 - ii. $\text{In } x \text{ input}$. The goal is $\text{In } x \text{ input}$, which holds.

□

Lemma 19 (Soundness of $\text{hasDataInThetaDelta}$). $\forall p, \forall \text{thetadelta}$,
 $\text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true} \leftrightarrow \exists a, \text{In } a \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a)) \wedge (\text{snd}(a) \neq \text{None}))$. The function $\text{hasDataInThetaDelta}$ with a port p and a set of pairs $(\text{name}, \text{optiondata})$ depicted by thetadelta where name denotes a port name and option data is a data item, $\text{hasDataInThetaDelta}$ returns true if there is not a pair

$a \in \text{thetadelta}$ where a 's name is equal to p 's name and the data item in a is different from None .

Proof.

\Rightarrow

Let $H: \text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true}$, for some p a port and thetadelta a set of pairs $(\text{name}, \text{optiondata})$. The objective is to prove $\exists a, \text{In } a \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a))) \wedge (\text{snd}(a) \neq \text{None})$. The proof proceeds by induction on the set of pairs thetadelta .

- (i) Induction Basis: let thetadelta be an empty set. H is $H: \text{hasDataInThetaDelta } p [] = \text{true}$. The goal is $\exists a, \text{In } a [] \wedge (\text{id } p = (\text{fst}(a))) \wedge (\text{snd}(a) \neq \text{None})$. As by $\text{hasDataInThetaDelta}$'s definition H is a contradiction, by simplification on H it is $H: \text{false} = \text{true}$. The goal is discharged.
- (ii) Induction Hypothesis: suppose the induction basis now holds for a nonempty set thetadelta . Therefore let IH a hypothesis where $IH: \text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true} \rightarrow \exists a, \text{In } a \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a))) \wedge (\text{snd}(a) \neq \text{None})$.
- (iii) Inductive Step: the objective is to prove that the Induction Hypothesis holds for a bigger set than the one supposed in the Induction Hypothesis. Therefore let an element a be added to thetadelta . H is $H: \text{hasDataInThetaDelta } p (a :: \text{thetadelta}) = \text{true}$. The aim is to prove that $\exists a_0, \text{In } a_0 (a :: \text{thetadelta}) \wedge (\text{id } p = (\text{fst}(a_0))) \wedge (\text{snd}(a_0) \neq \text{None})$. By analyzing H , with simplification on H , it unfolds to $H: (\text{if equiv_dec } (\text{id } p) (\text{fst } a) \text{ then if } (\text{snd } a) \neq \text{None} \text{ then true else hasDataInThetaDelta } p \text{ thetadelta} \text{ else hasDataInThetaDelta } p \text{ thetadelta}) = \text{true}$. Upon destructing equiv_dec to analyze the possible cases provided by H , one goal is generated for each possible return.
 - (a) let $(\text{id } p) = (\text{fst } a)$. H is $H: (\text{if } (\text{snd } a) \neq \text{None} \text{ then true else hasDataInThetaDelta } p \text{ thetadelta}) = \text{true}$. A new hypothesis $H_2: (\text{id } p) = (\text{fst } a)$ is added to the proof context. By destructing nequiv_dec , the non-equivalence relation denoted by \neq , two new goals, equal to the current goal are obtained, one for each possible value of nequiv_dec .
 - (1) let $(\text{snd } a) \neq \text{None}$. H is $H: \text{true} = \text{true}$. A new hypothesis H_3 is added to the proof context, where $H_3: (\text{snd } a) \neq \text{None}$. The goal is $\exists a_0, \text{In } a_0$

$(a :: \text{thetadelta}) \wedge (id\ p = (fst(a_0)) \wedge (snd(a_0) \neq \text{None}))$. The existential quantifier in the goal may be eliminated with a . The goal is $In\ a\ (a :: \text{thetadelta}) \wedge (id\ p = (fst(a)) \wedge (snd(a) \neq \text{None}))$. The goal can be split into three parts: first, $In\ a\ (a :: \text{thetadelta})$ holds because a is in $a :: \text{thetadelta}$. Therefore $In\ a\ (a :: \text{thetadelta})$ unfolds to $a = a \vee In\ a\ \text{thetadelta}$ which holds. $id\ p = (fst(a))$ holds from H_2 and $(snd(a) \neq \text{None})$ holds from H_3 . The goal therefore holds.

(2) let $(snd\ a) = \text{None}$. H is $H: \text{hasDataInThetaDelta}\ p\ \text{thetadelta} = \text{true}$. A new hypothesis H_3 is added to the proof context, where $H_3: (snd\ a) = \text{None}$. The goal is $\exists\ a_0, In\ a_0\ (a :: \text{thetadelta}) \wedge (id\ p = (fst(a_0)) \wedge (snd(a_0) \neq \text{None}))$. By applying IH to H , it is $\exists\ a, In\ a\ \text{thetadelta} \wedge (id\ p = (fst(a)) \wedge (snd(a) \neq \text{None}))$. Therefore a new port x can be introduced into the proof context from H . H is $H: In\ x\ \text{thetadelta} \wedge (id\ p = (fst(x)) \wedge (snd(x) \neq \text{None}))$. The existential quantifier in the goal can be eliminated with x . The goal is $In\ x\ (a :: \text{thetadelta}) \wedge (id\ p = (fst(x)) \wedge (snd(x) \neq \text{None}))$. The goal can be split into three parts: $In\ x\ (a :: \text{thetadelta})$ holds because it evaluates to $x = a \vee In\ x\ \text{thetadelta}$, where the latter holds from H . Both $(id\ p = (fst(x)))$ and $(snd(x) \neq \text{None})$ holds from H , proving the goal.

(b) let $(id\ p) \neq (fst\ a)$. H is $H: \text{hasDataInThetaDelta}\ p\ \text{thetadelta} = \text{true}$. A new hypothesis H_2 is added to the proof context, where $H_2: (id\ p) \neq (fst\ a)$. The goal is $\exists\ a_0, In\ a_0\ (a :: \text{thetadelta}) \wedge (id\ p = (fst(a_0)) \wedge (snd(a_0) \neq \text{None}))$. By applying IH to H , it is $\exists\ a, In\ a\ \text{thetadelta} \wedge (id\ p = (fst(a)) \wedge (snd(a) \neq \text{None}))$. Therefore a new port x can be introduced into the proof context from H . H is $H: In\ x\ \text{thetadelta} \wedge (id\ p = (fst(x)) \wedge (snd(x) \neq \text{None}))$. The existential quantifier in the goal can be eliminated with x . The goal is $In\ x\ (a :: \text{thetadelta}) \wedge (id\ p = (fst(x)) \wedge (snd(x) \neq \text{None}))$. The goal can be split into three parts: $In\ x\ (a :: \text{thetadelta})$ holds because it evaluates to $x = a \vee In\ x\ \text{thetadelta}$, where the latter holds from H . Both $(id\ p = (fst(x)))$ and $(snd(x) \neq \text{None})$ holds from H , proving the goal.

\Leftarrow

Let H be a hypothesis, $H: \exists\ a, In\ a\ \text{thetadelta} \wedge (id\ p = (fst(a)) \wedge (snd(a) \neq \text{None}))$.

The objective is to prove $\text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true}$. The proof proceeds by induction on the set of pairs thetadelta .

- (i) Induction Basis: let thetadelta be an empty set denoted by $[]$. H is $H: \exists a, \text{In } a [] \wedge (\text{id } p = (\text{fst}(a)) \wedge (\text{snd}(a) \neq \text{None}))$. The goal is $\text{hasDataInThetaDelta } p [] = \text{true}$. As the leftmost part of H , $\text{In } a []$, introduces a contradiction, this goal is discharged.
- (ii) Induction Hypothesis: suppose the Induction Hypothesis holds for a nonempty set thetadelta . This means that $IH: \exists a, \text{In } a \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a)) \wedge (\text{snd}(a) \neq \text{None})) \rightarrow \text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true}$.
- (iii) Inductive Step: The objective is to prove that the Induction Hypothesis holds for a nonempty set composed by $a :: \text{thetadelta}$, where thetadelta is a set of pairs $(\text{name}, \text{optiondata})$. Therefore, let $H: \exists a_0, \text{In } a_0 (a :: \text{thetadelta}) \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. By introducing the existential quantifier in H , one can derive a pair $(\text{name}, \text{optiondata})$ denoted by x . H is $\text{In } x (a :: \text{thetadelta}) \wedge (\text{id } p = (\text{fst}(x)) \wedge (\text{snd}(x) \neq \text{None}))$. By simplification of the goal, it unfolds to $(\text{if equiv_dec } (\text{id } p) (\text{fst } a) \text{ then if } (\text{snd } a) \neq \text{None} \text{ then true else hasDataInThetaDelta } p \text{ thetadelta} \text{ else hasDataInThetaDelta } p \text{ thetadelta}) = \text{true}$. Upon analyzing equiv_dec in the goal by destruction, two hypothesis are obtained, along with their respective goals to be proved.

- (a) let $(\text{id } p) = (\text{fst } a)$. The goal is $(\text{if } (\text{snd } a) \neq \text{None} \text{ then true else hasDataInThetaDelta } p \text{ thetadelta}) = \text{true}$. By analyzing the inequality present in the goal denoted by nequiv_dec , there are two possibilities.

- (1) let $(\text{snd } a) \neq \text{None}$. The goal is $\text{true} = \text{true}$ which holds.
- (2) let $(\text{snd } a) = \text{None}$. The goal is $\text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true}$.

A new hypothesis H_2 is then added to the proof context, where $H: (\text{snd } a) = \text{None}$. By applying the induction hypothesis IH , the goal is $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. From H , three new hypothesis can be derived: $H_3: \text{In } x (a :: \text{thetadelta})$, $H_4: (\text{id } p = (\text{fst}(x)))$ and $H_5: (\text{snd}(x) \neq \text{None})$. By simplification on H_3 , it unfolds to $H_3: x = a \vee \text{In } x \text{ thetadelta}$. By case analysis on the disjunction in H_3 , a new goal is generated:

- (A) let $H_3: x = a$. The goal is $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. By rewriting H_3 on H_2 , H_2 is $(\text{snd } x) = \text{None}$ which contradicts H_5 . The current goal is discharged.
- (B) let $H_3: \text{In } x \text{ thetadelta}$. The goal is $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. The existential quantifier in the goal over a_0 may be eliminated by instantiating x . The goal is $\text{In } x \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(x)) \wedge (\text{snd}(x) \neq \text{None}))$. From H_3 , H_4 and H_5 , the goal holds.
- (b) let $(\text{id } p) \neq (\text{fst } a)$. A new hypothesis $H_2: (\text{id } p) \neq (\text{fst } a)$ is added to the proof context. The goal is $\text{hasDataInThetaDelta } p \text{ thetadelta} = \text{true}$. By applying IH on the goal, it is $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. H is $H: \text{In } x (a :: \text{theTadelta}) \wedge (\text{id } p = (\text{fst}(x)) \wedge (\text{snd}(x) \neq \text{None}))$. From H , three new hypothesis can be derived: $H_3: \text{In } x (a :: \text{theTadelta})$, $H_4: (\text{id } p = (\text{fst}(x)))$ and $H_5: (\text{snd}(x) \neq \text{None})$. By simplification on H_3 , it unfolds to $H_3: x = a \vee \text{In } x \text{ thetadelta}$. By case analysis on the disjunction in H_3 , a new goal is generated:
- (i) let $H_3: a = x$. The goal is $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. By rewriting H_3 in H_4 , H_4 is $H_4: (\text{id } p = (\text{fst}(a)))$ which contradicts H_2 . The goal is discharged.
- (ii) let $H_3: \text{In } x \text{ thetadelta}$. $\exists a_0, \text{In } a_0 \text{ thetadelta} \wedge (\text{id } p = (\text{fst}(a_0)) \wedge (\text{snd}(a_0) \neq \text{None}))$. By H_3 , H_4 and H_5 , the goal is proved.

□

Lemma 20 (Soundness of *checkPorts*). $\forall t, \forall \text{theTadelta}, \text{checkPorts } t \text{ theTadelta} = \text{false} \leftrightarrow \exists a, \text{In } a t \wedge \text{negb } (\text{hasDataInThetaDelta } a \text{ theTadelta}) = \text{false}$. The function *checkPorts* with a set of ports t and a set of pairs $p = (\text{name}, \text{option data})$ returns *false* if there is an element $a \in \text{theTadelta}$ where *hasDataInThetaDelta* with a and *theTadelta* returns *false* and *true* otherwise. The aim of *checkPorts* is to verify if every element in a set of ports does not have data in $\theta.\delta$ by means of *hasDataInThetaDelta* with every element in the set of ports and a set of pairs denoting ports in $\theta.\delta$ and the data item each port has in $\theta.\delta$.

Proof.

\Rightarrow

Let $H: \text{checkPorts } t \text{ thetadelta} = \text{false}$. The goal is to prove $\exists a, \text{In } a \ t \wedge \text{negb} (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. The proof proceeds by induction on the set of ports depicted by t .

- (i) Induction Basis: let t be an empty set, denoted by $[]$. H is $H: \text{checkPorts } [] \text{ thetadelta} = \text{false}$. The goal is to prove $\exists a, \text{In } a \ [] \wedge \text{negb} (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. By checkPorts 's definition, H unfolds to a contradiction by simplification: H is $H: \text{true} = \text{false}$. Therefore this goal is discharged.
- (ii) Induction Hypothesis: suppose the Induction Basis holds for a nonempty set of ports t . In other words, let $IH: \text{checkPorts } t \text{ thetadelta} = \text{false} \rightarrow \exists a, \text{In } a \ t \wedge \text{negb} (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for an instance of a set t with an element added to it, namely $a :: t$. Let $H: \text{checkPorts } (a :: t) \text{ thetadelta} = \text{false}$. The goal is $\exists a_0, \text{In } a_0 \ (a :: t) \wedge \text{negb} (\text{hasDataInThetaDelta } a_0 \text{ thetadelta}) = \text{false}$. By simplification on H , it unfolds to $(\text{if equiv_dec } (\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) \text{ true then checkPorts } t \text{ thetadelta then false}) = \text{false}$. Upon analyzing equiv_dec by destruction, two hypothesis are obtained, for each the current goal may hold.
 - (a) let $(\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) = \text{true}$. A new hypothesis H_2 is added to the proof context, where $H_2: (\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) = \text{true}$. The goal is $\exists a_0, \text{In } a_0 \ (a :: t) \wedge \text{negb} (\text{hasDataInThetaDelta } a_0 \text{ thetadelta}) = \text{false}$. H is $H: \text{checkPorts } t \text{ thetadelta} = \text{false}$. By applying IH in H , it is $H: \exists a, \text{In } a \ t \wedge \text{negb} (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. From H it is possible to introduce the existential quantifier, generating x as a port in the proof context. H is $H: \text{In } x \ t \wedge \text{negb} (\text{hasDataInThetaDelta } x \text{ thetadelta}) = \text{false}$. The existential quantifier in the goal on a_0 may be eliminated by instantiating x . The goal is $\text{In } x \ (a :: t) \wedge \text{negb} (\text{hasDataInThetaDelta } x \text{ thetadelta}) = \text{false}$. The left side of the conjunction in the goal holds because by simplification, it unfolds to $x = a \vee \text{In } x \ t$, where the right side of this disjunction holds from H . The right side of the conjunction in the goal holds directly from H .

- (b) let $(\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) \neq \text{true}$. A new hypothesis H_2 is added to the proof context, where $H_2: (\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) \neq \text{true}$. H is $H: \text{false} = \text{false}$. The goal is $\exists a_0, \text{In } a_0 (a :: t) \wedge \text{negb } (\text{hasDataInThetaDelta } a_0 \text{ thetadelta}) = \text{false}$. The existential quantifier in the goal on a_0 may be eliminated with a . The goal is $\text{In } a (a :: t) \wedge \text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. From H_2 and *bool*'s definition in Coq, one can derive $(\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) = \text{false}$. The left side of the goal holds because by simplification it unfolds to $a = a \vee \text{In } a \ t$, where the left side of this disjunction holds. The right side of the disjunction on the goal holds directly from H_2 . Therefore the goal holds.

\Leftarrow

Let $H: \exists a, \text{In } a \ t \wedge \text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. The objective is to prove $\text{checkPorts } t \text{ thetadelta} = \text{false}$. The proof proceeds by induction on t .

- (i) Induction Basis: let t be an empty set denoted by $[]$. Then H is $H: \exists a, \text{In } a \ [] \wedge \text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}$. The goal is $\text{checkPorts } [] \text{ thetadelta} = \text{false}$. As from $H \exists a, \text{In } a \ []$ is a contradiction, the current goal is discharged.
- (ii) Induction Hypothesis: suppose the Induction Basis holds for a nonempty set of ports t . Therefore, let $IH: (\exists a, \text{In } a \ t \wedge \text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta}) = \text{false}) \rightarrow \text{checkPorts } t \text{ thetadelta} = \text{false}$.
- (iii) Inductive Step: the objective is to prove that the Induction Basis holds for a nonempty set t with an element a added to t . Therefore, let $H: \exists a_0, \text{In } a_0 (a :: t) \wedge \text{negb } (\text{hasDataInThetaDelta } a_0 \text{ thetadelta}) = \text{false}$. The goal is $\text{checkPorts } (a :: t) \text{ thetadelta} = \text{false}$. By simplification on the goal, it unfolds to $(\text{if equiv_dec } (\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) \text{ true then checkPorts } t \text{ thetadelta then false}) = \text{false}$. Upon analyzing *equiv_dec* in the goal, two possible goals are obtained.
 - (a) let $(\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) = \text{true}$. A new hypothesis is added to the proof context, where $H_2: (\text{negb } (\text{hasDataInThetaDelta } a \text{ thetadelta})) = \text{true}$. The goal is $\text{checkPorts } t \text{ thetadelta}$. By applying IH on the goal, it is $\exists a_0, \text{In } a_0 \ t \wedge \text{negb } (\text{hasDataInThetaDelta } a_0 \text{ thetadelta}) = \text{false}$. From H ,

a new port x is introduced in the proof context. H is $In\ x\ (a :: t) \wedge negb\ (hasDataInThetaDelta\ x\ thetadelta) = false$. The existential quantifier over a_0 in the goal can be eliminated by instantiating x . The goal is $In\ x\ t \wedge negb\ (hasDataInThetaDelta\ x\ thetadelta) = false$. Also, from H it is possible to derive $H_3: In\ x\ (a :: t)$ and $H_4: negb\ (hasDataInThetaDelta\ x\ thetadelta) = false$. By simplification on H_3 , it is $x = a \vee In\ x\ t$. Upon destructing H_3 , there are two possibilities, where for each the current goal must hold.

- (1) let $H_3: x = a$. The goal is $In\ x\ t \wedge negb\ (hasDataInThetaDelta\ x\ thetadelta) = false$. By rewriting H_3 , replacing the occurrence of a by x in H_2 , H_2 is $H_2: (negb\ (hasDataInThetaDelta\ a\ thetadelta)) = true$ which contradicts H_4 . The current goal is discharged.
- (2) let $H_3: In\ x\ t$. The goal is $In\ x\ t \wedge negb\ (hasDataInThetaDelta\ x\ thetadelta) = false$. Therefore the goal holds directly from H_3 and H_4 .

- (b) let $(negb\ (hasDataInThetaDelta\ a\ thetadelta)) \neq true$. The goal is $false = false$ which holds.

□

Lemma 21 (Soundness of *step'*). $\forall\ k, \forall\ input, \forall\ ports, \forall\ s, step'\ k\ input\ ports\ s \neq [] \rightarrow \exists\ a, In\ a\ s \wedge (set_eq\ (ports)((fst(fst(a)))) \ \&\&\ (onlyPortsInvolvedContainsData\ (fst(fst(a)))\ k\ input) \ \&\&\ (evalCompositeDc\ (input)\ (snd(fst(a)))) = true$. The function *step'* is bound to return all possible states reachable from a transition: given a natural number k which denotes the current step of the run, a set *input* which denotes the set of TDS given as parameter to the run, a set of ports *ports* which denotes the ports with data in $\theta.time(k)$, namely $\theta.N(k)$ and a set of transitions denoted by *s*, returns the reachable states with this transition, given that the ports in *s* are in $\theta.N(k)$, only these ports have data and that the data constraint is satisfied. It is important to note that here, *s* denotes the return of a transition relation as defined by CACoq, i.e., the output of *T* with a state $q \in Q$, $T\ q$. Hence *s* has type $set(set\ name \times DC \times set(state))$.

Proof. Let $H: step'\ k\ input\ ports\ s \neq []$. The objective is to prove that $\exists\ a, In\ a\ s \wedge (set_eq\ (ports)((fst(fst(a)))) \ \&\&\ (onlyPortsInvolvedContainsData\ (fst(fst(a)))\ k\ input) \ \&\&\ (evalCompositeDc\ (input)\ (snd(fst(a)))) = true$ holds. The proof proceeds by induction on the set of transitions *s*.

- (i) Induction Basis: let s be an empty set. The goal is to prove $\exists a, In\ a\ s \wedge (set_eq\ (ports)((fst(fst(a)))) \&\& (onlyPortsInvolvedContainsData\ (fst(fst(a)))\ k\ input) \&\& (evalCompositeDc\ (input)\ (snd(fst(a)))) = true$. H is $H: step'\ k\ input\ ports\ [] \neq []$. From $step'$'s definition, by simplification of H it unfolds to $[] \neq []$ which is a contradiction. Therefore the current goal is discharged.
- (ii) Induction Hypothesis: suppose the Inductive Basis holds for a nonempty arbitrary set of transitions s . In other words, let $IH: step'\ k\ input\ ports\ s \neq [] \rightarrow \exists a, In\ a\ s \wedge (set_eq\ (ports)((fst(fst(a)))) \&\& (onlyPortsInvolvedContainsData\ (fst(fst(a)))\ k\ input) \&\& (evalCompositeDc\ (input)\ (snd(fst(a)))) = true$.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for an arbitrary set s with a new element a added to it. Therefore let $H: step'\ k\ input\ ports\ (a :: s) \neq []$. The goal is $\exists a_0, In\ a_0\ (a :: s) \wedge (set_eq\ (ports)((fst(fst(a_0)))) \&\& (onlyPortsInvolvedContainsData\ (fst(fst(a_0)))\ k\ input) \&\& (evalCompositeDc\ (input)\ (snd(fst(a_0)))) = true$. By simplification on H , it is $(if\ equiv_dec\ (set_eq\ ports\ (fst\ (fst\ a)) \&\& onlyPortsInvolvedContainsData\ (fst\ (fst\ a))\ k\ input \&\& evalCompositeDc\ input\ (snd\ (fst\ a)))\ true\ then\ snd\ a\ ++\ step'\ k\ input\ ports\ s\ else\ step'\ k\ input\ ports\ s) \neq []$. By analyzing $equiv_dec$ in H , there are two possible hypothesis, where for each of them the current goal must hold.
- (a) let $(set_eq\ ports\ (fst\ (fst\ a)) \&\& onlyPortsInvolvedContainsData\ (fst\ (fst\ a))\ k\ input \&\& evalCompositeDc\ input\ (snd\ (fst\ a))) = true$. H is $H: snd\ a\ ++\ step'\ k\ input\ ports\ s \neq []$. A new hypothesis $H_2: (set_eq\ ports\ (fst\ (fst\ a)) \&\& onlyPortsInvolvedContainsData\ (fst\ (fst\ a))\ k\ input \&\& evalCompositeDc\ input\ (snd\ (fst\ a))) = true$ is added to the proof context. The goal is $\exists a_0, In\ a_0\ (a :: s) \wedge (set_eq\ (ports)((fst(fst(a_0)))) \&\& (onlyPortsInvolvedContainsData\ (fst(fst(a_0)))\ k\ input) \&\& (evalCompositeDc\ (input)\ (snd(fst(a_0)))) = true$. The existential quantifier in the goal over a_0 can be eliminated by instantiating a . The goal is $In\ a\ (a :: s) \wedge (set_eq\ (ports)((fst(fst(a_0)))) \&\& (onlyPortsInvolvedContainsData\ (fst(fst(a)))\ k\ input) \&\& (evalCompositeDc\ (input)\ (snd(fst(a)))) = true$. By simplification, the left side of the disjunction in the goal unfolds to $a = a \vee In\ a\ s$ where the left side of this disjunction holds. The right side of the goal holds from H_2 . The goal is

proved.

- (b) $(\text{set_eq } \text{ports } (\text{fst } (\text{fst } a)) \ \&\& \ \text{onlyPortsInvolvedContainsData } (\text{fst } (\text{fst } a)) \ k \ \text{input} \ \&\& \ \text{evalCompositeDc } \text{input } (\text{snd } (\text{fst } a))) \neq \text{true}$. The goal is $\exists a_0, \text{In } a_0 (a :: s) \wedge (\text{set_eq } (\text{ports})((\text{fst}(\text{fst}(a_0)))) \ \&\& \ (\text{onlyPortsInvolvedContainsData } (\text{fst}(\text{fst}(a_0))) \ k \ \text{input}) \ \&\& \ (\text{evalCompositeDc } (\text{input}) (\text{snd}(\text{fst}(a_0)))) = \text{true}$. H is $H: \text{step}' \ k \ \text{input } \text{ports } s \neq []$. By applying IH in H . H is $H: \exists a, \text{In } a \ s \wedge (\text{set_eq } (\text{ports})((\text{fst}(\text{fst}(a)))) \ \&\& \ (\text{onlyPortsInvolvedContainsData } (\text{fst}(\text{fst}(a))) \ k \ \text{input}) \ \&\& \ (\text{evalCompositeDc } (\text{input}) (\text{snd}(\text{fst}(a)))) = \text{true}$. From H , a new transition x can be obtained by introducing the existential quantifier over a . H is $H: \text{In } x \ s \wedge (\text{set_eq } (\text{ports})((\text{fst}(\text{fst}(x)))) \ \&\& \ (\text{onlyPortsInvolvedContainsData } (\text{fst}(\text{fst}(x))) \ k \ \text{input}) \ \&\& \ (\text{evalCompositeDc } (\text{input}) (\text{snd}(\text{fst}(x)))) = \text{true}$. The existential quantifier in the goal over a_0 may be eliminated by instantiating x . The goal is $\text{In } x (a :: s) \wedge (\text{set_eq } (\text{ports})((\text{fst}(\text{fst}(x)))) \ \&\& \ (\text{onlyPortsInvolvedContainsData } (\text{fst}(\text{fst}(x))) \ k \ \text{input}) \ \&\& \ (\text{evalCompositeDc } (\text{input}) (\text{snd}(\text{fst}(x)))) = \text{true}$. The left part of the conjunction in the goal holds by simplification, where it unfolds to $x = a \vee \text{In } x \ a$ where the right side of this disjunction is derivable from H . The right side of the conjunction in the goal holds from H . The goal is proved

□

Lemma 22 (Soundness of *stepa*). *This definition with a constraint automaton ca , a set of states s , a natural number k denoting the amount of steps of the current run and the limit of the streams in $\theta \in TDS^{\text{Names}}$, a set of ports input which denotes $\theta \in TDS^{\text{Names}}$ and a set of port names names which denotes the ports in $\theta.N(k)$ returns a pair $(\text{ports}, \text{states})$ where ports is ports and states are the states obtained by means of step' with k , input , names and ca 's transition relation with all states in s .*

Proof. This proof is straightforward from *stepa*'s definition. By unfolding it, it unrolls to $(\text{ports}, \text{flat_map } (\text{stepAux } ca \ k \ \text{input } \text{ports}) \ s)$, where *stepAux* only calls step' with all parameters as depicted, where the transition expected by step' is ca 's transition relation T with a state ains . Namely, *stepAux* is $\text{step}' \ k \ l \ \text{input } \text{ports} \ (T \ ca \ s)$. Therefore *stepa* returns a pair $(\text{ports}, \text{states})$ where ports is ports and states are the states obtained by means of step' with k , l , input , names and ca 's transition relation with all states $a \in s$,

proving the Lemma. \square

Lemma 23 (Soundness of *step*). *The function *step* applies *step'* to a set of states by means of *stepa*, returning a pair (*ports*, *states*) where *ports* is a set of port names that were in $\theta.N$, therefore port names on the fired transitions in the current step and *states* are all possible states reached by applying these transitions. Therefore *step* with a constraint automaton *ca*, a natural number *k* and which respectively denotes the number of steps of the current run and the upper bound of the stream, a set of states *s* in which the step will be applied and a set of ports *input* denoting $\theta \in TDS^{Names}$*

Proof. The proof is straightforward from *step*'s definition. Upon unfolding, it is *stepa ca s k input (retrievePortsFromThetaN k input)*, where the port names provided to *stepa* are the ones in $\theta.N(k)$, recovered by means of *retrievePortsFromThetaN*, a definition that only calls *thetaN*. Therefore, from Lemmas 23 and 13, the current Lemma is proved. \square

Lemma 24 (Soundness of *run'*). *The function *run'* is the core definition that enables run in constraint automata in CACoq. Its definition comprises the idea of iterating *k* times over the current automaton's configuration, storing the execution trace in a list. Therefore this function with a set of natural numbers *k*, a set of ports *input* denoting the $\theta \in TDS^{Names}$, a set of states that stands for the set of initial states of \mathcal{A} , *run'* returns a set of sets of states that denotes the execution trace of the automaton for *k* steps.*

Proof. This proof is straightforward from *run'*'s definition. By unfolding it, the definition is

```

fix rec input k acc resp :=
  match k with
  | [] => resp
  | a::t => resp ++ [snd (step ca acc a input)]
  | > rec
    (flat_map(derivativePortInvolved(fst((step ca acc a input))))
  t (snd (step ca acc a input))
end.

```

which can be read as “for every element in *k*, this definition concatenates the set of states *resp* with $[snd (step ca acc a input)]$, where the second set denotes the resulting states of the current step of the run on \mathcal{A} , recursively recalling the function defined within

run', defined by means of keyword *fix* which functions similar to *Fixpoint*, with *input* as $\theta \in TDS^{Names}$, where the TDS for each port in the transitions triggered had their derivatives calculated by means of *derivativePortInvolved*, the next step is the next natural number in *k* and the next set of states to be iterated is the result of the current step. This definition acts as intermediate for *run* which will be soon addressed, where *k* is an crescent set of natural numbers, ranging from 0 to *k*. Therefore, this Lemma is proved by Lemmas 3, 14, 15 and 23. \square

C.3 Product Automata

The following Lemmas comprise the construction of product automata, regarding how the resulting elements of the tuple that is a product automaton are built, the resulting set of states, resulting set of ports, resulting transition relation and the resulting set of initial states.

Lemma 25 (Soundness of *resultingStatesSet*). $\forall a1, \forall a2, \forall a, \forall b, \text{In } (a, b) \text{ (resultingStatesSet } a1 \ a2) \leftrightarrow \text{In } a \text{ (ConstraintAutomata.Q } a1) \wedge \text{In } b \text{ (ConstraintAutomata.Q } a2)$. The function *resultingStatesSet* with two constraint automata \mathcal{A}_1 and \mathcal{A}_2 returns the Cartesian product of \mathcal{A}_1 's states set with \mathcal{A}_2 's states set. The Lemma states that for all elements (a, b) in the resulting set produced by *resultingStatesSet*, *a* must be in \mathcal{A}_1 's states set and *b* \mathcal{A}_2 's states set.

Proof. The proof is straightforward from *resultingStatesSet*'s definition. By unfolding, it is *list_prod* (*ConstraintAutomata.Q* *a1*) (*ConstraintAutomata.Q* *a2*) where *list_prod* is a function that returns the product of two lists, defined in Coq's standard library. As sets hereby used are implemented as lists, it is possible to use *list_prod* for the necessary means, which soundness proof is also present in Coq's standard library. \square

Lemma 26 (Soundness of *resultingNameSet*). $\forall a1, \forall a2, \forall a, \text{In } a \text{ (resultingNameSet } a1 \ a2) \leftrightarrow \text{In } a \text{ (ConstraintAutomata.N } a1) \vee \text{In } a \text{ (ConstraintAutomata.N } a2)$. The function *resultingNameSetSound* with two constraint automata \mathcal{A}_1 and \mathcal{A}_2 returns the union of \mathcal{A}_1 's names set with \mathcal{A}_2 's names set. The Lemma states that any element in the resulting set must be either in \mathcal{A}_1 's names set or \mathcal{A}_2 's names set.

Proof. The proof is straightforward from *resultingNameSet*'s definition. By unfolding, it

is `set_union (ConstraintAutomata.N a1) (ConstraintAutomata.N a2)` where `set_union` is a function that returns the union of two sets of type `ListSet`, defined in Coq's standard library. The soundness proof of `set_union` can be found in Coq's standard library. \square

Lemma 27 (Soundness of `resultingInitialStatesSet`). $\forall a1, \forall a2, \forall a, \forall b, \text{In } (a,b) \text{ (resultingInitialStatesSet } a1 \ a2) \leftrightarrow \text{In } a \text{ (ConstraintAutomata.Q0 } a1) \wedge \text{In } b \text{ (ConstraintAutomata.Q0 } a2)$. The function `resultingStatesSet` with two constraint automata \mathcal{A}_1 and \mathcal{A}_2 returns the Cartesian product of \mathcal{A}_1 's initial states set with \mathcal{A}_2 's initial states set. The Lemma states that for all states (a,b) in the resulting set produced by `resultingInitialStatesSet`, q_1 must be in \mathcal{A}_1 's states set and \mathcal{A}_2 's states set.

Proof. The proof is straightforward from `resultingInitialStatesSet`'s definition. By unfolding, it is `list_prod (ConstraintAutomata.Q0 a1) (ConstraintAutomata.Q0 a2)` where `list_prod` is a function that returns the product of two lists, defined in Coq's standard library. As sets hereby used are implemented as lists, it is possible to use `list_prod` for the necessary means, which soundness proof is also present in Coq's standard library. \square

The next Lemmas comprise the soundness of the formalized definitions in order to obtain the transition relation of a product automaton by means of the rules discussed in Section 5.1.

Lemma 28 (Soundness of `evaluateConditionsFirstRule`). $\forall t1, \forall t2, \forall \text{names1}, \forall \text{names2}, \text{evaluateConditionsFirstRule } t1 \ t2 \ \text{names1} \ \text{names2} = \text{true} \leftrightarrow \text{set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1) (fst(fst(t2))))} = \text{true}$. This Lemma asserts that the function `evaluateConditionsFirstRule` returns `true` if it with $t1$ and $t2$ as two transitions of type $(\text{set } (\text{name}) \times DC \times \text{set } (\text{state}))$, names1 and names2 denoting \mathcal{A}_1 and \mathcal{A}_2 's set names respectively, `evaluateConditionsFirstRule` returns `true` only if the intersection of names1 with the set of port names of $t2$ equals the intersection of names2 with $t1$'s set of port names, in accordance with the first rule denoted in Definition 11.

Proof.

\Rightarrow

Let $H: \text{evaluateConditionsFirstRule } t1 \ t2 \ \text{names1} \ \text{names2} = \text{true}$. Upon unfolding `evaluateConditionsFirstRule` in H , H is $H: (\text{if set_eq (set_inter equiv_dec names2 (fst (fst$

$t1)))$ (*set_inter equiv_dec names1 (fst (fst t2)))* *then true else false*) = *true*. The goal is *set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1) (fst(fst(t2))))* = *true*. By case analysis on H , let

- (i) *set_eq (set_inter equiv_dec names2 (fst (fst t1))) (set_inter equiv_dec names1 (fst (fst t2)))* = *true*. H is $H: \text{true} = \text{true}$. A new hypothesis $H_2: \text{set_inter equiv_dec names2 (fst (fst t1)) (set_inter equiv_dec names1 (fst (fst t2))) = true}$ is added to the proof context. The goal is *set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1) (fst(fst(t2))))* = *true*. By rewriting H_2 , the goal is *true = true* which holds.
- (ii) *set_eq (set_inter equiv_dec names2 (fst (fst t1))) (set_inter equiv_dec names1 (fst (fst t2)))* = *false*. H is $H: \text{false} = \text{true}$. The current goal is *set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1) (fst(fst(t2))))* = *true* which is discharged because H is a contradiction.

\Leftarrow

Let *set_eq (set_inter equiv_dec (names2) (fst(fst(t1)))) (set_inter equiv_dec (names1) (fst(fst(t2))))* = *true*. The objective is to prove *evaluateConditionsFirstRule t1 t2 names1 names2* = *true*. By unfolding *evaluateConditionsFirstRule* it is (*if set_eq (set_inter equiv_dec names2 (fst (fst t1))) (set_inter equiv_dec names1 (fst (fst t2))) then true else false*) = *true*. By rewriting H in the goal, it replaces *set_eq (set_inter equiv_dec names2 (fst (fst t1))) (set_inter equiv_dec names1 (fst (fst t2)))* with *true*. The goal unfolds to *true = true* which holds. \square

Lemma 29 (Soundness of *buildResultingTransitionFromStatesRule1*). $\forall p1, \forall p2, \text{buildResultingTransitionFromStatesRule1 } p1 \ p2 \neq [] \leftrightarrow p2 \neq []$. The function *buildResultingTransitionFromStatesRule1* is an auxiliary function which will be used in order to create the resulting states of a transition in the product automaton retrieved. Therefore *buildResultingTransitionFromStatesRule1* with a state $p1$ denoting an state of \mathcal{A}_1 and a set of states $p2$ denoting the set of states returned in a transition of \mathcal{A}_2 returns a set of states (p, a) for every state $a \in p2$. This Lemma states that it returns an empty set only if the given set of states $p2$ is an empty set.

Proof. This proof is straightforward from *buildResultingTransitionFromStatesRule1*'s definition. By unfolding, it is

```

match p2 with
| [] => []
| a::t => (p1,a)::
    buildResultingTransitionFromStatesRule1 p1 t
end.

```

which suffices for the current Lemma to hold. \square

Lemma 30 (Soundness of *buildResultingTransitionFromStatesBothTransitionsRule1*). $\forall p1, \forall p2, \text{buildResultingTransitionFromStatesBothTransitionsRule1 } p1 \ p2 \neq [] \leftrightarrow \exists a, \text{In } a \ p1 \wedge \text{buildResultingTransitionFromStatesRule1 } a \ p2 \neq []$. The aim of *buildResultingTransitionFromStatesBothTransitionsRule1*'s behavior lies in the application of *buildResultingTransitionFromStatesRule1* with all states $a \in p1$ and $p2$ as a set of states, with $p1$ denoting the resulting set of states of a transition of \mathcal{A}_1 and $p2$ the resulting set of states of a transition of \mathcal{A}_2 .

Proof. This proof is straightforward from *buildResultingTransitionFromStatesBothTransitionsRule1*'s definition. By unfolding, it is

```

match p1 with
| [] => []
| a::t => buildResultingTransitionFromStatesRule1 a p2++
    buildResultingTransitionFromStatesBothTransitionsRule1 t p2
end.

```

From *buildResultingTransitionFromStatesBothTransitionsRule1*'s definition, it does not return an empty set if $p1$ is not an empty set and there is an element a in $p2$ where *buildResultingTransitionFromStatesRule1* $a \ p2$ also returns a non empty set. Therefore by Lemma 29 \square

Lemma 31 (Soundness of *buildResultingTransitionFromSingleStateRule1*). $\forall Q1, \forall Q2, \forall \text{transition1}, \forall \text{transition2}, \forall \text{names1}, \forall \text{names2}, \text{buildResultingTransitionFromSingleStateRule1 } Q1 \ Q2 \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2} \neq [] \leftrightarrow (\text{evaluateConditionsFirstRule } (\text{transition1}) \ (\text{transition2}) \ (\text{names1}) \ (\text{names2})) = \text{true}$. The function *buildResultingTransitionFromSingleStateRule1* with two states $Q1$ and $Q2$ denoting a state of \mathcal{A}_1 and a state of \mathcal{A}_2 respectively, two transitions *transitions1* and *transitions2* denoting a transition of \mathcal{A}_1 and a transition of \mathcal{A}_2 respectively and two set names *names1* and *names2* with *names1* as the

name set of \mathcal{A}_1 and *names2* as \mathcal{A}_2 's name set returns a set with a single element, namely the transition built with those elements as denoted by the first rule in Definition 11

Proof.

\Rightarrow

The proof is straightforward from *buildResultingTransitionFromSingleStateRule1*'s definition in the hypothesis H : *buildResultingTransitionFromSingleStateRule1 Q1 Q2 transition1 transition2 names1 names2* . By unfolding, it is

```

    if (evaluateConditionsFirstRule (transition1) (transition2) (names1) (names2))

    == true then

        [((Q1,Q2),(((set_union equiv_dec (fst(fst(transition1))) (fst(fst(transition2))))),
        (snd(fst(transition1)))

                                (snd(fst(transition2)))),(buildResultingTransitionFromStatesBothTr
        (snd(transition2)))]

```

else []. In order to *buildResultingTransitionFromSingleStateRule1* to return a non empty set containing a single transition built as denoted by the first rule in Definition 11, (*evaluateConditionsFirstRule (transition1) (transition2) (names1) (names2)*) = *true* must hold.

\Leftarrow Suppose H : (*evaluateConditionsFirstRule (transition1) (transition2) (names1) (names2)*) = *true*. The aim is to prove *buildResultingTransitionFromSingleStateRule1 Q1 Q2 transition1 transition2 names1 names2* $\neq []$ The proof is straightforward from *buildResultingTransitionFromSingleStateRule1*'s definition in the goal. By unfolding it, the goal is *if (evaluateConditionsFirstRule (transition1) (transition2) (names1) (names2)) == true then* [((*Q1,Q2*),(((*set_union equiv_dec (fst(fst(transition1))) (fst(fst(transition2))*))),*ConstraintAutomata.andDc (snd(fst(transition1)))*

(*snd(fst(transition2))*)),(*buildResultingTransitionFromStatesBothTransitionsRule1 (snd(transition2))*)]

else []. By rewriting H in the goal, it unfolds to [((*Q1,Q2*),(((*set_union equiv_dec (fst(fst(transition1))) (fst(fst(transition2))*))),*ConstraintAutomata.andDc (snd(fst(transition1))*

(*snd*(*fst*(*transition2*))))),(*buildResultingTransitionFromStatesBothTransitionsRule1* *Q1 Q2 transition1 transition2 names1 names2* = *buildResultingTransitionFromSingleStateRule1* *Q1 Q2 transition1 a names1 names2*). The function *buildTransitionFromMoreTransitionsRule1* with two states *Q1* and *Q2* denoting a state of \mathcal{A}_1 and a state of \mathcal{A}_2 respectively, two transitions *transitions1* and *transitions2* denoting a single transition of \mathcal{A}_1 and a set of transitions denoting the transition relation of \mathcal{A}_2 respectively and two set names *names1* and *names2* with *names1* as the name set of \mathcal{A}_1 and *names2* as \mathcal{A}_2 's names set returns the result of applying *buildResultingTransitionFromSingleStateRule1* *Q1 Q2 transition1 a names1 names2*, which is *buildResultingTransitionFromSingleStateRule1* with all required parameters, for every *a* a transition, $a \in \text{transitions2}$. The aim of *buildTransitionFromMoreTransitionsRule1* is to apply *buildResultingTransitionFromSingleStateRule1* to a fixed transition of the transition relation of \mathcal{A}_1 denoted by *transition1* with \mathcal{A}_2 's transition relation, depicted by *transitions2* in order to obtain resulting transitions by means of the first rule depicted in Definition 11.

Proof. The proof is straightforward from *buildTransitionFromMoreTransitionsRule1*'s definition. It suffices to unfold its definition to obtain

```

match transition2 with
| [] => []
| a::t => (buildResultingTransitionFromSingleStateRule1 Q1 Q2
           transition1 a names1 names2)++
           (buildTransitionFromMoreTransitionsRule1 Q1 Q2
            transition1 t names1 names2)
end.

```

which suffices to prove the current Lemma, along with Lemma 29. □

Lemma 33 (Soundness of *buildTransitionFromMoreAllTransitionsSingleState*). $\forall Q1, \forall Q2, \forall transition1, \forall transition2, \forall names1, \forall names2, transition1 = a :: t \rightarrow buildTransitionFromMoreAllTransitionsSingleState Q1 Q2 transitions1 transitions2 names1 names2 = buildTransitionFromMoreTransitionsRule1 Q1 Q2 a transition2 names1 names2$. The

aim of *buildTransitionFromMoreAllTransitionsSingleState* is to apply *buildTransitionFromMoreTransitionsRule1* to all transitions in the set of transitions *transition1* that denotes \mathcal{A}_1 's transition relation, therefore building all possible resulting transitions by means of *buildResultingTransitionFromSingleStateRule1* that implements the first rule depicted in Definition 11 for fixed states *Q1* and *Q2*, being a single state of \mathcal{A}_1 and \mathcal{A}_2 , respectively. The aim of *buildTransitionFromMoreAllTransitionsSingleState* is to obtain all possible transitions with *buildResultingTransitionFromSingleStateRule1* for all transitions from both automata with departing states *Q1* and *Q2*.

Proof. The proof is straightforward from *buildTransitionFromMoreAllTransitionsSingleState*'s definition. Upon unfolding, it is

```

match transition1 with
| [] => []
| a::t => (buildTransitionFromMoreTransitionsRule1 Q1 Q2 a
           transition2 names1 names2)++
          (buildTransitionFromMoreAllTransitionsSingleState Q1 Q2 t
           transition2 names1 names2)
end.

```

which suffices to prove that for every $a \in \text{transition1}$, *buildTransitionFromMoreAllTransitionsSingleState* applies *buildTransitionFromMoreTransitionsRule1* to a with the other parameters supplied to *buildTransitionFromMoreAllTransitionsSingleState*. \square

Lemma 34 (Soundness of *iterateOverStatesBuildingTransitionsOne*). $\forall Q1, \forall Q2, \forall \text{transition1}, \forall \text{transition2}, \forall \text{names1}, \forall \text{names2}, Q2 = a :: t \rightarrow \text{iterateOverStatesBuildingTransitionsOne } Q1 \ Q2 \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2} = (\text{buildTransitionFromMoreAllTransitionsSingleState } Q1 \ Q2 \ a \ \text{transition2} \ \text{names1} \ \text{names2}) ++ (\text{iterateOverStatesBuildingTransitionsOne } Q1 \ t \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2})$. The aim of *iterateOverStatesBuildingTransitionsOne* is to apply *buildTransitionFromMoreAllTransitionsSingleState* with a set of states *Q2* denoting \mathcal{A}_2 's set of states. Therefore *iterateOverStatesBuildingTransitionsOne* with *Q1* a state, *Q2* a set of states, *transition1* and *transition2* as \mathcal{A}_1 's and \mathcal{A}_2 as their respective transition relation of type $\text{state} \rightarrow (\text{set}(\text{set name}) \times DC \times (\text{set}(\text{state})))$ and *names1* as \mathcal{A}_1 's set of port names and *names2* \mathcal{A}_2 's set of port names returns the result of *buildTransitionFromMoreAllTransitionsSingleState* with every state $q \in Q2$.

Proof. The proof is straightforward from *iterateOverStatesBuildingTransitionsOne*'s definition. Upon unfolding, it is

```

match Q2 with
| [] => []
| a::t => (buildTransitionFromMoreAllTransitionsSingleState Q1 a
           (transition1 Q1) (transition2 a) names1 names2)++
           (iterateOverStatesBuildingTransitionsOne Q1 t transition1
            transition2 names1 names2)
end.

```

Therefore, because *iterateOverStatesBuildingTransitionsOne*'s definition states that with a nonempty set *Q2*, *iterateOverStatesBuildingTransitionsOne* returns $(\text{buildTransitionFromMoreAllTransitionsSingleState } Q1 \ a \ (\text{transition1 } Q1) \ (\text{transition2 } a) \ \text{names1} \ \text{names2})++(\text{iterateOverStatesBuildingTransitionsOne } Q1 \ t \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2})$ this Lemma holds. \square

Lemma 35 (Soundness of *buildAllTransitionsRule1*). $\forall Q1, \forall Q2, \forall \text{transition1}, \forall \text{transition2}, \forall \text{names1}, \forall \text{names2}, Q1 = a :: t \rightarrow \text{buildAllTransitionsRule1 } Q1 \ Q2 \ \text{transitions1} \ \text{transitions2} \ \text{names1} \ \text{names2} = (\text{iterateOverStatesBuildingTransitionsOne } a \ Q2 \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2}) ++ (\text{buildAllTransitionsRule1 } t \ Q2 \ \text{transition1} \ \text{transition2} \ \text{names1} \ \text{names2})$. The function *buildAllTransitionsRule1* with *Q1* as the set of states of \mathcal{A}_1 , *Q2* as the set of states of \mathcal{A}_2 , *transition1* and *transition2* as \mathcal{A}_1 and \mathcal{A}_2 's transition relation respectively and *names1* and *names2* as \mathcal{A}_1 and \mathcal{A}_2 's names set respectively returns all obtainable transitions as denoted by the first rule in Definition 11 by means of *iterateOverStatesBuildingTransitionsOne* with all states $a \in Q1$.

Proof. The proof is straightforward from *buildAllTransitionsRule1*'s definition. Upon unfolding, it is

```

match Q1 with
| [] => []
| a::t => (iterateOverStatesBuildingTransitionsOne a Q2 transition1 transition2
           names1 names2)++ (buildAllTransitionsRule1 t Q2 transition1 transition2 names1
                             names2)
end

```

Therefore, because *buildAllTransitionsRule1*'s definition states that with a nonempty

set $Q1$, $buildAllTransitionsRule1$ returns $(iterateOverStatesBuildingTransitionsOne\ a\ Q2\ (transition1\ Q1)\ (transition2\ a)\ names1\ names2)++\ (buildAllTransitionsRule1\ t\ Q2\ transition1\ transition2\ names1\ names2)$ this Lemma holds. \square

Lemma 36 (Soundness of $transitionsRule1$). $\forall\ a1, \forall\ a2, transitionsRule1\ a1\ a2 = buildAllTransitionsRule1\ (ConstraintAutomata.Q\ a1)\ (ConstraintAutomata.Q\ a2)\ (ConstraintAutomata.T\ a1)\ (ConstraintAutomata.T\ a2)\ (ConstraintAutomata.N\ a1)\ (ConstraintAutomata.N\ a2)$. The aim of this definition lies on building a set containing all transitions that can be retrieved from the product automaton construction with the first transition rule denoted in Definition 11. Therefore $transitionsRule1$ with $a1$ denoting a constraint automaton \mathcal{A}_1 and $a2$ as another constraint automaton \mathcal{A}_2 invokes $buildAllTransitionsRule1$ with, respectively, $a1$'s states set, $a2$'s states set, $a1$'s transition relation, $a2$'s transition relation, $a1$'s names set and $a2$'s names set.

Proof. The proof is straightforward from $transitionsRule1$. By unfolding $transitionsRule1$, it is $buildAllTransitionsRule1\ (ConstraintAutomata.Q\ a1)\ (ConstraintAutomata.Q\ a2)\ (ConstraintAutomata.T\ a1)\ (ConstraintAutomata.T\ a2)\ (ConstraintAutomata.N\ a1)\ (ConstraintAutomata.N\ a2)$, which holds. The soundness of this Lemma also relies on Lemmas regarding all functions implemented in order to achieve $transitionsRule1$, namely Lemmas 28,29,30,31,32, 34. \square

Lemma 37 (Soundness of $intersectionNAndNames$). $\forall\ tr, \forall\ names2, intersectionNAndNames\ tr\ names2 = true \leftrightarrow set_inter\ equiv_dec\ (fst\ (fst(tr)))\ names2 = nil$. The function $intersectionNAndNames$ implements the comparison of tr 's associated names set, where tr is a transition of \mathcal{A}_1 and $names2$ is \mathcal{A}_2 's names set as required by the second rule of the resulting transition relation depicted by Definition 11. Therefore $intersectionNAndNames$ returns $true$ if the intersection of $names2$ with tr 's associated names set is the empty set, denoted by nil .

Proof.

\Rightarrow

Suppose $H: intersectionNAndNames\ tr\ names2 = true$. The objective is to prove $set_inter\ equiv_dec\ (fst\ (fst(tr)))\ names2 = nil$. By unfolding $intersectionNAndNames$ in H , it unfolds to $H: (if\ equiv_dec\ (set_inter\ equiv_dec\ (fst\ (fst\ tr)))\ names2)\ []\ then\ true\ else\ false) = true$. Therefore by destructing $equiv_dec$ in H , two possibilities are obtained.

- (i) Suppose $H_2: (\text{set_inter fst (fst tr)}) \text{ names2} = \text{nil}$. The goal is $\text{set_inter equiv_dec (fst (fst(tr))) names2} = \text{nil}$. H is $H: \text{true} = \text{true}$. The goal holds from H_2 .
- (ii) Suppose $H_2: (\text{set_inter fst (fst tr)}) \text{ names2} = \text{nil}$. The goal is $\text{set_inter equiv_dec (fst (fst(tr))) names2} \neq \text{nil}$. H is $H: \text{false} = \text{true}$. As H is a contradiction the current goal is discharged.

\Leftarrow

Suppose $H: \text{set_inter equiv_dec (fst (fst(tr))) names2} = \text{nil}$. The objective is to prove $\text{intersectionNAndNames tr names2} = \text{true}$. By unfolding $\text{intersectionNAndNames}$ in the goal, it is $(\text{if equiv_dec (set_inter equiv_dec (fst (fst tr)) names2)} \text{ then true else false}) = \text{true}$. Upon rewriting H in the goal, it unfolds to $\text{true} = \text{true}$ which holds. \square

Lemma 38 (Soundness of $\text{iterateOverOutboundStatesRule2}$). $\forall p1, \forall q2, \text{iterateOverOutboundStatesRule2 } p1 \ q2 \neq [] \leftrightarrow p1 \neq []$. The function $\text{iterateOverOutboundStatesRule2}$ with a set of states $p1$ and a state $q2$ returns a set containing states $(a, q2)$ where $a \in p1$.

Proof.

\rightarrow

Suppose $H: \text{iterateOverOutboundStatesRule2 } p1 \ q2 \neq []$. The goal is to prove $p1 \neq []$. The proof proceeds by induction on the set of states $p1$.

- (i) Induction Basis: let $p1 \neq []$. H is $H: \text{iterateOverOutboundStatesRule2 } [] \ q2 \neq []$. The goal is $p1 \neq []$. By simplification on H it unfolds to $H: [] \neq []$. As H is a contradiction, the goal is discharged.
- (ii) Induction Hypothesis: Suppose the Induction Basis holds for an non empty arbitrary set $p1$. Therefore suppose $IH: \text{iterateOverOutboundStatesRule2 } p1 \ q2 \neq [] \rightarrow p1 \neq []$.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for a set t with a new element a added to it. Therefore let

$H: \text{iterateOverOutboundStatesRule2 } (a :: p1) \ q2 \neq []$. The goal is $p1 \neq []$. Upon unfolding $\text{iterateOverOutboundStatesRule2}$ in the goal, it is

$\text{match } p1 \text{ with}$

$| [] \Rightarrow []$

$| a::t \Rightarrow \text{set_add equiv_dec } ((a, q2))$

(*iterateOverOutboundStatesRule2* t $q2$)

end.

From *iterateOverOutboundStatesRule2*'s definition, it returns an empty set only if the set of states given as parameter $p1$ is not empty. Therefore the goal holds.

\Leftarrow

The proof is straightforward from *iterateOverOutboundStatesRule2*'s definition. Suppose $H: p1 \neq []$. By unfolding *iterateOverOutboundStatesRule2*, it is

match $p1$ with

| $[] \Rightarrow []$

| $a::t \Rightarrow \text{set_add equiv_dec } ((a, q2))$

(*iterateOverOutboundStatesRule2* t $q2$)

end.

which by simplification is $\text{set_add equiv_dec } ((a, q2))$ (*iterateOverOutboundStatesRule2* t $q2$). The function *set_add* adds the element to the set denoted by (*iterateOverOutboundStatesRule2* t $q2$) if $(a, q2)$. Therefore from H the goal holds. \square

Lemma 39 (Soundness of *createSingleTransition*). $\forall q1, \forall \text{transition}, \forall a2States, \forall a2Names, \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \neq [] \leftrightarrow \text{intersectionNAndNames } (\text{transition}) \ (a2Names) = \text{true} \wedge a2States \neq []$. The function *createSingleTransition* is a function that implements the construction of a single transition by means of the second rule introduced in the transition relation of Definition 11. Therefore *createSingleTransition* with a state of \mathcal{A}_1 $q1$, the return of a transition denoted by *transition*, a set of states $a2States$ denoting \mathcal{A}_2 's set of states and $a2Names$ as the names set of \mathcal{A}_2 .

Proof.

\Rightarrow

Suppose $H: \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \neq []$. The objective is to prove $\text{intersectionNAndNames } (\text{transition}) \ (a2Names) = \text{true} \wedge a2States \neq []$. The proof proceeds by induction on the set of states $a2States$.

- (i) Induction Basis: let $a2States = []$ (an empty set). H is $H: \text{createSingleTransition } q1 \text{ transition } [] \ a2Names \neq []$. The goal is $\text{intersectionNAndNames } (\text{transition}) \ (a2Names) = \text{true} \wedge a2States \neq []$. By simplification on H , it is $H: [] \neq []$. As H is a contradiction, the current goal is discharged.

(ii) Induction Hypothesis: suppose the Induction Basis holds for an arbitrary $a2States$. Therefore let $IH: createSingleTransition\ q1\ transition\ a2States\ a2Names \neq [] \rightarrow intersectionNAndNames\ (transition)\ (a2Names) = true \wedge a2States \neq []$.

(iii) Inductive Step: let $a2States$ now be a set with an element added to it, namely $(a :: a2States)$. H is $H: createSingleTransition\ q1\ transition\ (a :: a2States)\ a2Names \neq []$. The goal is $intersectionNAndNames\ (transition)\ (a2Names) = true \wedge (a :: a2States) \neq []$. By simplification of $intersectionNAndNames$, H unfolds to $H: (if\ equiv_dec\ (intersectionNAndNames\ transition\ a2Names)\ true\ then\ (q1,\ a,\ (fst\ transition,\ iterateOverOutboundStatesRule2\ (snd\ transition)\ a)) :: createSingleTransition\ q1\ transition\ a2States\ a2Names\ else\ createSingleTransition\ q1\ transition\ a2States\ a2Names) \neq []$. Therefore by case analysis on $equiv_dec$, two hypothesis are obtained.

(a) Let $H_2: (intersectionNAndNames\ transition\ a2Names) = true$. H is $H: (q1,\ a,\ (fst\ transition,\ iterateOverOutboundStatesRule2\ (snd\ transition)\ a)) :: createSingleTransition\ q1\ transition\ a2States\ a2Names$. The goal is $intersectionNAndNames\ (transition)\ (a2Names) = true \wedge (a :: a2States) \neq []$. The right part of the conjunction in the goal holds from H_2 and its left part holds because $(a :: a2States)$ is not an empty set. The goal is proved.

(b) $H_2: (intersectionNAndNames\ transition\ a2Names) \neq true$. H unfolds to $H: createSingleTransition\ q1\ transition\ a2States\ a2Names \neq []$. The goal is $intersectionNAndNames\ (transition)\ (a2Names) = true \wedge (a :: a2States) \neq []$. By applying IH on H , it is $H: intersectionNAndNames\ (transition)\ (a2Names) = true \wedge a2States \neq []$. From H , two hypothesis can be derived, namely $H_3: intersectionNAndNames\ (transition)\ (a2Names) = true$ and $H_4: a2States \neq []$. As H_3 contradicts H_2 , the current goal is discharged.

\Leftarrow

Let $H: intersectionNAndNames\ (transition)\ (a2Names) = true \wedge a2States \neq []$. The objective is to prove $createSingleTransition\ q1\ transition\ a2States\ a2Names \neq []$. The proof proceeds by induction on the set of states $a2States$.

(i) Induction Basis: let $a2States$ an empty set. H is $H: intersectionNAndNames\ (transition)\ (a2Names) = true \wedge [] \neq []$. The goal is $createSingleTransition\ q1\ transition$

\square $a2Names \neq []$. From H two hypothesis can be introduced in the proof context, namely $H_2: \text{intersectionNAndNames } (transition) (a2Names) = true$ and $H_3: [] \neq []$. As H_3 is a contradiction, the current goal is discharged.

(ii) Induction Hypothesis: suppose the Induction Basis holds for a nonempty set $a2States$. Therefore let $IH: \text{intersectionNAndNames } (transition) (a2Names) = true \wedge a2States \neq [] \rightarrow \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \neq []$.

(iii) Inductive Step: let $a2States$ now be a set with an element added to it, namely $(a :: a2States)$. The objective is to prove that the Induction Basis holds for a set $(a :: a2States)$. Therefore H is $H: \text{intersectionNAndNames } (transition) (a2Names) = true \wedge (a :: a2States) \neq [] \wedge (a :: a2States) \neq []$. The goal is $\text{createSingleTransition } q1 \text{ transition } (a :: a2States) \ a2Names \neq []$. By simplification of the goal, it unfolds to $(\text{if equiv_dec } (\text{intersectionNAndNames } transition \ a2Names) \ true \ \text{then } (q1, a, (\text{fst } transition, \text{iterateOverOutboundStatesRule2 } (\text{snd } transition) \ a)) :: \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \ \text{else } \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names) \neq []$. By case analysis of equiv_dec in the goal, two goals are obtained.

(a) Let $H_2: (\text{intersectionNAndNames } transition \ a2Names) = true$. The goal is $(q1, a, (\text{fst } transition, \text{iterateOverOutboundStatesRule2 } (\text{snd } transition) \ a)) :: \text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \neq []$ which holds.

(b) Let $H_2: (\text{intersectionNAndNames } transition \ a2Names) \neq true$. The goal is $\text{createSingleTransition } q1 \text{ transition } a2States \ a2Names \neq []$. By applying IH in the goal, it unfolds to $\text{intersectionNAndNames } (transition) (a2Names) = true \wedge a2States \neq []$. By H , two hypothesis can be obtained, namely $H_3: \text{intersectionNAndNames } (transition) (a2Names) = true$ and $H_4: a2States \neq []$. As H_3 contradicts H_2 , The current goal is discharged.

□

Lemma 40 (Soundness of $\text{createTransitionRule2}$). $\forall q1, \forall transitions, \forall q2, \forall names2, transitions = a :: t \rightarrow \text{createTransitionRule2 } q1 \text{ transitions } q2 \text{ names2} = (\text{createSingleTransition } q1 \ a \ q2 \ names2) ++ (\text{rec } t \ q2 \ names2)$. The function $\text{createTransitionRule2}$ is defined by means of an anonymous recursive function defined by means of fix . Therefore

this Lemma states that *createTransitionRule2* with a state from \mathcal{A}_1 *q1*, a set of the return of a transition relation with a state, namely a set *transitions* of type *set (set (name) × DC × (set(state)))*, a set *q2* denoting the set of states Q of \mathcal{A}_2 and a set of states *names2* which denotes \mathcal{A}_2 's set of names returns the resulting transitions built by means of *createSingleTransition*. In short, *createTransitionRule2* applies the second rule from Definition 11 in a set of states *q2*.

Proof. The proof is straightforward from *createTransitionRule2*'s definition. Upon unfolding, it is

```
fix rec transitions q2 names2 :=
  match transitions with
  | [] ⇒ []
  | a::t ⇒ (createSingleTransition q1 a q2 names2)++(rec t q2 names2)
```

end. From *createTransitionRule2*'s definition, if *transitions* is not an empty set then *createTransitionRule2* applies *createSingleTransition* to every element $a \in \text{transitions}$, proving the current Lemma. \square

Lemma 41 (Soundness of *createTransitionRule2AllStates*). $\forall Q1, \forall \text{transitions}, \forall \text{names2}, \forall a2States, Q1 = a :: t \rightarrow (\text{createTransitionRule2 } a (\text{transitions}(a)) a2States \text{names2})++(\text{createTransitionRule2 } t \text{transitions names2 } a2States)$. The function *createTransitionRule2AllStates* applies the construction of the transition relation by means of the second rule provided by Definition 11. Therefore *createTransitionRule2AllStates* with a set of states *Q1* denoting the set of states of \mathcal{A}_1 , The transition relation of \mathcal{A}_1 denoted by *transitions*, \mathcal{A}_2 's names set as *names2* and \mathcal{A}_2 's states set as *a2States*

Proof. The proof is straightforward from *createTransitionRule2AllStates*'s definition. Upon unfolding, it is

```
match Q1 with
| [] ⇒ []
| a::t ⇒ (createTransitionRule2 a (transitions(a)) a2States names2)++(createTransitionRule2
t transitions names2 a2States)

end.
```

From *createTransitionRule2AllStates*'s definition, if *Q1* is not an empty set then *createTransitionRule2AllStates* applies *createTransitionRule2* to every state $a \in Q1$, where

the provided transition relation's return is $\text{transitions}(a)$, proving the current Lemma. \square

Lemma 42 (Soundness of transitionsRule2). $\forall a1, \forall a2, \text{transitionsRule2 } a1 \ a2 = (\text{createTransitionRule2AllStates } (\text{ConstraintAutomata.Q } a1) (\text{ConstraintAutomata.T } a1) (\text{ConstraintAutomata.N } a2) (\text{ConstraintAutomata.Q } a2))$. The aim of this definition lies on building a set containing all transitions that can be retrieved from the product automaton construction with the second transition rule denoted in Definition 11. Therefore transitionsRule2 with $a1$ denoting a constraint automaton \mathcal{A}_1 and $a2$ as another constraint automaton \mathcal{A}_2 applies $\text{createTransitionRule2AllStates}$ with, respectively, $a1$'s states set, $a1$'s transition relation, $a2$'s names set and $a2$'s names set.

Proof. The proof is straightforward from transitionsRule2 . By unfolding transitionsRule2 , it is $(\text{createTransitionRule2AllStates } (\text{ConstraintAutomata.Q } a1) (\text{ConstraintAutomata.T } a1) (\text{ConstraintAutomata.N } a2) (\text{ConstraintAutomata.Q } a2))$ which holds. The soundness of this Lemma also relies on Lemmas regarding all functions implemented in order to achieve transitionsRule2 , namely Lemmas 37,38,39,40 and 41, \square

Lemma 43 (Soundness of $\text{iterateOverOutboundStatesRule3}$). $\forall q2, \forall p1, \text{iterateOverOutboundStatesRule3 } q2 \ p1 \neq [] \leftrightarrow p1 \neq []$. The function $\text{iterateOverOutboundStatesRule3}$ with a state $q2$ and a set $p1$ of states returns a set containing states $(q2, a)$, for all $a \in p1$.

Proof.

\rightarrow

Suppose $H: \text{iterateOverOutboundStatesRule3 } q2 \ p1 \neq []$. The goal is to prove $p1 \neq []$. The proof proceeds by induction on the set of states $p1$.

- (i) Induction Basis: let $p1 \neq []$. H is $H: \text{iterateOverOutboundStatesRule3 } q2 \ [] \neq []$. The goal is $p1 \neq []$. By simplification on H it unfolds to $H: [] \neq []$. As H is a contradiction, the goal is discharged.
- (ii) Induction Hypothesis: Suppose the Induction Basis holds for an non empty arbitrary set $p1$. Therefore suppose $IH: \text{iterateOverOutboundStatesRule3 } q2 \ p1 \neq [] \rightarrow p1 \neq []$.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for a set t with a new element a added to it. Therefore let

H : *iterateOverOutboundStatesRule3* $q2$ ($a :: p1$) $\neq []$. The goal is $p1 \neq []$. Upon unfolding *iterateOverOutboundStatesRule3* in the goal, it is

```
match p1 with
| [] => []
| a::t => set_add equiv_dec ((q2,a))(iterateOverOutboundStatesRule3 q2 t)
end.
```

From *iterateOverOutboundStatesRule3*'s definition, it returns an empty set only if the set of states given as parameter $p1$ is not empty. Therefore the goal holds.

\Leftarrow

The proof is straightforward from *iterateOverOutboundStatesRule3*'s definition. Suppose H : $p1 \neq []$. By unfolding *iterateOverOutboundStatesRule3*, it is

```
match p1 with
| [] => []
| a::t =>
    set_add equiv_dec ((q2,a))(iterateOverOutboundStatesRule3 q2 t)
end.
```

which by simplification is *set_add* *equiv_dec* (($a, q2$)) (*iterateOverOutboundStatesRule3* t $q2$). The function *set_add* adds the element to the set denoted by (*iterateOverOutboundStatesRule3* $q2$ t) if ($a, q2$). Therefore from H the goal holds. \square

Lemma 44 (Soundness of *intersectionNAndNames2*). $\forall tr, \forall names1, intersectionNAndNames2\ tr\ names1 = true \leftrightarrow set_inter\ equiv_dec\ (fst(fst(tr)))\ names1 = nil$. The function *intersectionNAndNames2* returns *true* if the intersection of the names set associated with *transition* with the names set depicted by $names1$ is the empty set, where *set_inter* is the set intersection operation provided by Coq's standard library *ListSet*.

Proof.

\Rightarrow

Let H : *intersectionNAndNames2* tr $names1 = true$. The objective is to prove *set_inter* *equiv_dec* ($fst(fst(tr))$) $names1 = nil$. By unfolding *intersectionNAndNames2*'s definition in H , H now is H : (*if* *equiv_dec* (*set_inter* *equiv_dec* ($fst(fst\ tr)$) $names1$) $[]$ *then true else false*) = *true*. By case analysis on *equiv_dec* in H , let

(i) H_2 : *set_inter* *equiv_dec* ($fst(fst\ tr)$) ($names1$) = *nil*. H unfolds to H : *true* = *true*.

The goal being *set_inter equiv_dec (fst(fst(tr))) names1 = nil* holds from H .

- (ii) H_2 : *set_inter equiv_dec (fst (fst tr)) (names1) ≠ nil*. H is H : *false = true*. The goal being *set_inter equiv_dec (fst(fst(tr))) names1 = nil* is discharged because H is a contradiction.

⇒

Let H : *set_inter equiv_dec (fst(fst(tr))) names1 = nil*. The objective is to prove *intersectionNAndNames2 tr names1 = true*. By unfolding *intersectionNAndNames2* in the goal, it unfolds to (*if equiv_dec (set_inter equiv_dec (fst (fst tr)) names1) [] then true else false*) = *true*. By rewriting H in the goal, it evaluates to (*if equiv_dec [] [] then true else false*) which simplifies to *true = true*, which holds. \square

Lemma 45 (Soundness of *createSingleTransitionRule3*). $\forall q2, \forall \text{transition}, \forall a1States, \forall a1Names, \text{createSingleTransitionRule3 } q2 \text{ transition } a1States \ a1Names \neq [] \leftrightarrow \text{intersectionNAndNames2 (transition) (a1Names)} = \text{true} \wedge a1States \neq []$. The function *createSingleTransitionRule3* creates transitions with a single state of \mathcal{A}_2 as specified by the third rule in Definition 11.

Proof.

⇒

Let H : *createSingleTransitionRule3 q2 transition a1States a1Names ≠ []*. The objective is to prove *intersectionNAndNames2 (transition) (a1Names) = true ∧ a1States ≠ []*. The proof proceeds by induction on the set of states *a1States*.

- (i) Induction Basis: let *a1States* be the empty set. H is H : *createSingleTransitionRule3 q2 transition [] a1Names ≠ []*. The goal is *intersectionNAndNames2 (transition) (a1Names) = true ∧ [] ≠ []*. By simplification of H , it unfolds to $\wedge [] \neq []$. As H is a contradiction, the goal is discharged.
- (ii) Induction Hypothesis: let the Induction Basis holds for an arbitrary set *a2States*. Therefore let IH : *createSingleTransitionRule3 q2 transition a1States a1Names ≠ []* → *intersectionNAndNames2 (transition) (a1Names) = true ∧ a1States ≠ []*.
- (iii) Inductive Step: the objective is to prove that the Induction Basis holds for a set $(a :: a2States)$. Therefore let H : *createSingleTransitionRule3 q2 transition (a :: a1States) a1Names ≠ []*. The goal is *intersectionNAndNames2 (transition)*

$(a1Names) = true \wedge (a :: a1States) \neq []$. By simplification of H , it unfolds to $(if\ equiv_dec\ (intersectionNAndNames2\ transition\ a1Names)\ true\ then\ (a,\ q2,(fst\ transition,\ iterateOverOutboundStatesRule3\ a\ (snd\ transition))) :: createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names\ else\ createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names) \neq []$. By case analysis of $equiv_dec$ in H , let

- (i) $H_2: (intersectionNAndNames2\ transition\ a1Names) = true$. H unfolds to $H: (a,\ q2,\ (fst\ transition,\ iterateOverOutboundStatesRule3\ a\ (snd\ transition))) :: createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names \neq []$.

The goal is $intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge (a :: a1States) \neq []$. The left part of the disjunction in the goal holds from H_2 . The right part holds by definition. Therefore the goal holds.

- (ii) $H_2: (intersectionNAndNames2\ transition\ a1Names) \neq true$. H is $H: createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names \neq []$. The goal is $intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge (a :: a1States) \neq []$. By applying IH on H , H now is $H: intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge a1States \neq []$. From H two hypothesis can be derived, namely $H_3: intersectionNAndNames2\ (transition)\ (a1Names) = true$ and $H_4: (a :: a1States) \neq []$. The left part of the goal holds from H_3 . The right part holds by definition. Therefore the goal holds.

\Leftarrow

Suppose $H: intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge a1States \neq []$. The aim is to prove $createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names \neq []$. The proof proceeds by induction on $a1States$.

- (i) Induction Basis: let $a1States$ be an empty set $[]$. H is $H: intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge [] \neq []$. The goal is $createSingleTransitionRule3\ q2\ transition\ []\ a1Names \neq []$. From H , a hypothesis $H_2: [] \neq []$ can be derived. As H_2 is a contradiction, the current goal is discharged.
- (ii) Induction Hypothesis: suppose the Induction Basis holds for a non empty set $a1States$. In other words, let $IH: intersectionNAndNames2\ (transition)\ (a1Names) = true \wedge a1States \neq [] \rightarrow createSingleTransitionRule3\ q2\ transition\ a1States\ a1Names \neq []$.

(iii) Inductive Step: the objective is to prove that the Induction Basis holds for a set composed by *a1States* with an state *a*, namely (*a :: a1States*). Therefore let $H: \text{intersectionNAndNames2 } (\text{transition}) \text{ } (a1Names) = \text{true} \wedge (a :: a1States) \neq []$. The goal is $\text{createSingleTransitionRule3 } q2 \text{ transition } (a :: a1States) \text{ } a1Names \neq []$. By simplification on the goal, it is $(\text{if equiv_dec } (\text{intersectionNAndNames2 } \text{transition } a1Names) \text{ true then } (a, q2, (\text{fst transition}, \text{iterateOverOutboundStatesRule3 } a \text{ } (\text{snd transition}))) :: \text{createSingleTransitionRule3 } q2 \text{ transition } a1States \text{ } a1Names \text{ else } \text{createSingleTransitionRule3 } q2 \text{ transition } a1States \text{ } a1Names) \neq []$. By case analysis on *equiv_dec*, let

- (a) $H_2: (\text{intersectionNAndNames2 } \text{transition } a1Names) = \text{true}$. The goal is $(a, q2, (\text{fst transition}, \text{iterateOverOutboundStatesRule3 } a \text{ } (\text{snd transition}))) :: \text{createSingleTransitionRule3 } q2 \text{ transition } a1States \text{ } a1Names$. As $(a, q2, (\text{fst transition}, \text{iterateOverOutboundStatesRule3 } a \text{ } (\text{snd transition}))) :: \text{createSingleTransitionRule3 } q2 \text{ transition}$ is a set that contains at least one known element (namely $(a, q2, (\text{fst transition}, \text{iterateOverOutboundStatesRule3 } a \text{ } (\text{snd transition}))))$, the goal is proved.
- (b) $H_2: (\text{intersectionNAndNames2 } \text{transition } a1Names) \neq \text{true}$. The goal is $\text{createSingleTransitionRule3 } q2 \text{ transition } a1States \text{ } a1Names \neq []$. From *H*, two new hypothesis may be introduced in the proof context, namely $H_3: \text{intersectionNAndNames2 } (\text{transition}) \text{ } (a1Names) = \text{true}$ and $H_4: (a :: a1States) \neq []$. As H_4 contradicts H_2 , the current goal is discharged.

□

Lemma 46 (Soundness of *createTransitionRule3*). $\forall q2, \forall \text{transitions}, \forall q1, \forall \text{names1}, \text{transitions} = a :: t \rightarrow \text{createTransitionRule3 } q2 \text{ transitions } q1 \text{ names1} = (\text{createSingleTransitionRule3 } q2 \text{ } a \text{ } q1 \text{ names1}) ++ (\text{rec } t \text{ } q1 \text{ names1})$. The function *createTransitionRule3* is defined by means of an anonymous recursive function defined by means of *fix*. Therefore this Lemma states that *createTransitionRule3* with a state from \mathcal{A}_2 *q2*, a set of the return of a transition relation with a state, namely a set *transitions* of type *set* (*set* (*name*) \times *DC* \times (*set*(*state*))), a set *q1* denoting the set of states *Q* of \mathcal{A}_1 and a set of states *names1* which denotes \mathcal{A}_1 's set of names returns the resulting transitions built by means of *createSingleTransition*. In short, *createTransitionRule3* applies the second rule

from Definition 11 in a set of states $q2$.

Proof. The proof is straightforward from *createTransitionRule3*'s definition. Upon unfolding, it is

```
fix rec transitions q1 names1 :=
  match transitions with
  | [] => []
  | a::t => (createSingleTransitionRule3 q2 a q1 names1)++(rec t q1 names1)
end.
```

From *createTransitionRule3*'s definition, if *transitions* is not an empty set then *createTransitionRule3* applies *createSingleTransitionRule3* to every element $a \in \text{transitions}$, proving the current Lemma. \square

Lemma 47 (Soundness of *createTransitionRule3AllStates*). $\forall Q2, \forall \text{transitions}, \forall \text{names1}, \forall a1States, Q2 = a :: t \rightarrow (\text{createTransitionRule3 } a (\text{transitions}(a)) a1States \text{ names1})++(\text{createTransitionRule3AllStates } t \text{ transitions names1 } a1States)$. The function *createTransitionRule3AllStates* applies the construction of the transition relation by means of the third rule provided by Definition 11. Therefore *createTransitionRule3AllStates* with a set of states $Q2$ denoting the set of states of \mathcal{A}_2 , The transition relation of \mathcal{A}_2 denoted by *transitions*, \mathcal{A}_1 's names set as *names1* and \mathcal{A}_1 's states set as *a1States*

Proof. The proof is straightforward from *createTransitionRule3AllStates*'s definition. Upon unfolding, it is

```
match Q2 with
| [] => []
| a::t => (createTransitionRule3 a (transitions(a)) a1States names1)++(createTransitionRule3
t transitions names1 a1States)
end.
```

From *createTransitionRule3AllStates*'s definition, if $Q2$ is not an empty set then *createTransitionRule3AllStates* applies *createTransitionRule2* to every state $a \in Q2$, where the provided transition relation's return is *transitions*(a), proving the current Lemma. \square

Lemma 48 (Soundness of *transitionsRule3*). $\forall a1, \forall a2, \text{transitionsRule3 } a1 a2 = (\text{createTransitionRule3AllStates } (\text{ConstraintAutomata.Q } a2) (\text{ConstraintAutomata.T } a2) (\text{ConstraintAutomata.N } a1) (\text{ConstraintAutomata.Q } a1))$. The aim of this definition lies

on building a set containing all transitions that can be retrieved from the product automaton construction with the second transition rule denoted in Definition 11. Therefore *transitionsRule3* with *a1* denoting a constraint automaton \mathcal{A}_1 and *a2* as another constraint automaton \mathcal{A}_2 applies *createTransitionRule3AllStates* with, respectively, *a2*'s states set, *a2*'s transition relation, *a1*'s names set and *a1*'s states set.

Proof. The proof is straightforward from *transitionsRule3*. By unfolding *transitionsRule3*, it is $(\text{createTransitionRule3AllStates } (\text{ConstraintAutomata.Q } a2) (\text{ConstraintAutomata.T } a2) (\text{ConstraintAutomata.N } a1) (\text{ConstraintAutomata.Q } a1))$ which holds. The soundness of this Lemma also relies on Lemmas regarding all functions implemented in order to achieve *transitionsRule3*, namely Lemmas 43,45,46 \square

Lemma 49 (Soundness of *buildTransitionRuleProductAutomaton*). $\forall a1, \forall a2$, *buildTransitionRuleProductAutomaton a1 a2* = $(\text{transitionsRule1 } a1 \ a2) ++ (\text{transitionsRule2 } a1 \ a2) ++ (\text{transitionsRule3 } a1 \ a2)$. The function *buildTransitionRuleProductAutomaton* with two automata *a1* and *a2* returns the set of transitions produced by means of *transitionsRule1*, *transitionsRule2* and *transitionsRule3*.

Proof. The proof is straightforward from *buildTransitionRuleProductAutomaton*'s definition. It suffices to unfold it in order to *buildTransitionRuleProductAutomaton* be $(\text{transitionsRule1 } a1 \ a2) ++ (\text{transitionsRule2 } a1 \ a2) ++ (\text{transitionsRule3 } a1 \ a2)$. Therefore by Lemmas 36,42,48, the current Lemma is proved. \square

Lemma 50 (Soundness of *recoverResultingStatesPA*). $\forall st, \forall t$, *recoverResultingStatesPA st t* $\neq [] \leftrightarrow \exists a, \text{In } a \ t \wedge st = fst(a)$. The function *recoverResultingStatesPA* is defined as a function that retrieves the return of a transition relation as depicted by Definition 12. Given a state (a,b) and a transition relation *t*, *recoverResultingStatesPA* returns all transitions with its departing state as (a,b) , where *t* is a set of possible transitions, being a set of type $set \ (state \times state2 \times (set \ name \times DC \times set \times (state \times state2)))$ with *state* is the type of states of \mathcal{A}_1 and *states2* the type of states of \mathcal{A}_2 .

Proof.

\Rightarrow

Let $H: \text{recoverResultingStatesPA } st \ t \neq []$. The objective is to prove $\exists a, \text{In } a \ t \wedge st = fst(a)$. The proof proceeds by induction on the transitions set *t*.

- (i) Induction Basis: Let t be an empty set. H is $H: \text{recoverResultingStatesPA } st \ [] \neq []$ while the goal is $\exists a, \text{In } a \ [] \wedge st = fst(a)$. By simplification of H it unfolds to $[] \neq []$. As H is a contradiction, this goal is discharged.
- (ii) Induction Hypothesis: Suppose the Induction Basis holds for an arbitrary set t . Therefore let $IH: \text{recoverResultingStatesPA } st \ t \neq [] \rightarrow \exists a, \text{In } a \ t \wedge st = fst(a)$.
- (iii) Inductive Step: The objective is to prove that the Induction Basis holds for a set composed by t with an element a added to it. Therefore let $H: \text{recoverResultingStatesPA } st \ (a :: t) \neq []$. The goal is to prove $\exists a_0, \text{In } a_0 \ (a :: t) \wedge st = fst(a_0)$. By simplification on H , it unfolds to $(\text{if equiv_dec } st \ (fst \ a) \ \text{then } snd \ a :: \text{recoverResultingStatesPA } st \ t \ \text{else } \text{recoverResultingStatesPA } st \ t) \neq []$. Therefore by case analysis on equiv_dec in H , let
- (a) $H_2: st = (fst \ a)$. H unfolds to $H: snd \ a :: \text{recoverResultingStatesPA } st \ t$. The goal is $\exists a_0, \text{In } a_0 \ (a :: t) \wedge st = fst(a_0)$. The existential quantifier over a_0 may be eliminated by instantiating a as a_0 . The goal is $\text{In } a \ (a :: t) \wedge st = fst(a)$. The left side of the goal holds by simplification, which unfolds to $a = a \vee \text{In } a \ t$. The right side of the goal holds from H_2 .
- (b) $H_2: st \neq (fst \ a)$. H is $H: \text{recoverResultingStatesPA } st \ t \neq []$. The goal is $\exists a_0, \text{In } a_0 \ (a :: t) \wedge st = fst(a_0)$. By applying IH on H , it is $\exists a, \text{In } a \ t \wedge st = fst(a)$. From H , a new transition of the same type of the elements in t can be obtained by introducing the existential quantifier in H . Let x be this element. H is $\text{In } x \ t \wedge st = fst(x)$. The existential quantifier in the goal may be eliminated by instantiating x as a_0 . The goal unfolds to $\text{In } a_0 \ (a :: t) \wedge st = fst(a_0)$. From H , two hypothesis can be derived by eliminating the conjunction in this hypothesis. Let $H_3: \text{In } x \ t$ and $H_4: st = fst(a)$. The left part of the goal holds by simplification, which unfolds to $x = t \vee \text{In } x \ t$ where the right side of this disjunction holds from H_3 . The right side of the conjunction in the goal holds from H_4 .

\Rightarrow

Let $H: \exists a, \text{In } a \ t \wedge st = fst(a)$. The objective is to prove $\text{recoverResultingStatesPA } st \ t \neq []$. The proof proceeds by induction on the set t .

- (i) Induction= Basis: let t be an empty set. Therefore H unfolds to $H: \exists a, \text{In } a [] \wedge st = fst(a)$. From H , an element x of type of elements of t may be introduced in the proof context. The goal is $recoverResultingStatesPA\ st [] \neq []$. As the left side of the conjunction in H is a contradiction, the current goal is discharged.
- (ii) Induction Hypothesis : suppose the Induction Basis holds for an arbitrary set t . Therefore let $IH: \exists a, \text{In } a\ t \wedge st = fst(a) \rightarrow recoverResultingStatesPA\ st\ t \neq []$.
- (iii) Inductive Step: the objective is to prove that the Induction Basis holds for a set composed by t with an element a added to it. Therefore let $H: \exists a_0, \text{In } a_0\ (a :: t) \wedge st = fst(a_0)$. The goal is $recoverResultingStatesPA\ st\ (a :: t) \neq []$. By simplification, the goal is $(\text{if } equiv_dec\ st\ (fst\ a)\ \text{then } snd\ a :: recoverResultingStatesPA\ st\ t\ \text{else } recoverResultingStatesPA\ st\ t) \neq []$. By destructing $equiv_dec$ in the goal, let
 - (a) $H_2: st = (fst\ a)$. The goal is $snd\ a :: recoverResultingStatesPA\ st\ t$ which holds.
 - (b) $H_2: st \neq (fst\ a)$. The goal is $recoverResultingStatesPA\ st\ t) \neq []$. By applying IH on the goal, it is $\exists a_0, \text{In } a_0\ t \wedge st = fst(a_0)$. From H , a new element x may be added to the proof context by introducing the existential quantifier in H . H is $H: \text{In } x\ (a :: t) \wedge st = fst(x)$. By eliminating the disjunction in H , two hypothesis can be derived, namely $H_3: \text{In } x\ (x :: t)$ and $H_4: st = fst(x)$. By simplification on H_3 , it is $a = x \vee \text{In } x\ t$. By eliminating the disjunction in H_3 , let
 - (1) $H_5: a = x$. By rewriting H_5 in H_4 , H_4 unfolds to $st = fst(x)$. The goal is $\exists a, \text{In } a\ t \wedge st = fst(a)$. As H_4 contradicts H_2 , the current goal is discharged.
 - (2) $H_5: \text{In } x\ t$. The goal is $\exists a_0, \text{In } a_0\ t \wedge st = fst(a_0)$. The existential quantifier over a_0 in the goal may be eliminated by instantiating x . The goal is $\text{In } x\ t \wedge st = fst(x)$ which holds from H_4 and H_5 .

□

Lemma 51 (Soundness of *transitionPA*). $\forall s, \forall a1, \forall a2\ transitionPA\ s = recoverResultingStatesPA\ s\ (buildTransitionRuleProductAutomaton\ a1\ a2)$. The function *transitionPA* is the definition of the resulting transition relation for product automata. The idea is

to model a transition relation as defined by Definition 12 for product automata. This Lemma states that the transition relation is built upon the set of transitions as returned by *buildTransitionRuleProductAutomaton* by means of *recoverResultingStatesPA* in order to retrieve the result of the transition with *s*, where *a1* and *a2* are two constraint automaton formalized in Coq.

Proof. The proof is straightforward from *transitionPA*'s definition. It suffices to unfold its definition in order to obtain *recoverResultingStatesPA s (buildTransitionRuleProductAutomaton a1 a2)*. \square

Lemma 52 (Soundness of *buildPA*). $\forall a1, \forall a2, \text{buildPA } a1 \ a2 = (\text{resultingStatesSet } a1 \ a2) (\text{resultingNameSet } a1 \ a2) (\text{transitionPA}) (\text{resultingInitialStatesSet } a1 \ a2)$. The function *buildPA* returns the product automaton composed from *a1* and *a2* as two constraint automata formalized in Coq.

Proof. The proof is straightforward from *buildPA*'s definition. By unfolding, it is $(\text{resultingStatesSet } a1 \ a2) (\text{resultingNameSet } a1 \ a2) (\text{transitionPA}) (\text{resultingInitialStatesSet } a1 \ a2)$. Therefore by Lemmas 25, 26,27 and 51 the current Lemma is proved. \square

Bibliography

- [1] J. C. Knight, “Safety critical systems: challenges and directions,” in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 547–550.
- [2] S. Gerhart, D. Craigen, and T. Ralston, “Case study: Paris metro signaling system,” *IEEE Software*, vol. 11, no. 1, pp. 32–28, 1994.
- [3] J. Peleska, “Formal methods for test automation-hard real-time testing of controllers for the airbus aircraft family,” *IDPT’02*, vol. 1, 2002.
- [4] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, “Model checking flight control systems: The airbus experience,” in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 18–27.
- [5] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *IEEE computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [6] N. Leveson *et al.*, “Medical devices: The therac-25,” *Appendix of: Safeware: System Safety and Computers*, 1995.
- [7] R. Milner, “Some directions in concurrency theory.” in *FGCS*, vol. 88, 1988, pp. 163–164.
- [8] J. S. Ostro, “Formal methods for the specification and design of real-time safety critical systems,” *Journal of Systems and Software*, vol. 18, no. 1, pp. 33–60, 1992.
- [9] E. A. Lee, “Cyber physical systems: Design challenges,” in *Object Oriented Real-Time Distributed Computing*, 2008.
- [10] E. Grilo and B. Lopes, “Formalization and certification of software for smart cities,” in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2018, pp. 662–669.

- [11] D. W. Loveland, *Automated Theorem Proving: a logical basis*. Elsevier, 2014.
- [12] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner, *The COQ Proof Assistant: User's Guide: Version 5.6*. INRIA, 1992.
- [13] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.
- [14] N. de Souza Silva, “Verificação formal de sistemas embarcados em carro elétrico,” Master’s thesis, Universidade Federal de Goiás, 2015.
- [15] X. Leroy, “Formal certification of a compiler back-end or: programming a compiler with a proof assistant,” in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 42–54.
- [16] F. Arbab, “Reo: a channel-based coordination model for component composition,” *Mathematical Structures in Computer Science*, vol. 14, no. 3, p. 329–366, 2004.
- [17] ———, “Coordination for component composition,” *Electronic Notes in Theoretical Computer Science*, vol. 160, pp. 15 – 40, 2006, proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).
- [18] N. Kokash and F. Arbab, *Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 21–41.
- [19] Y. Li and M. Sun, “Modeling and verification of component connectors in coq,” *Science of Computer Programming*, vol. 113, pp. 285–301, 2015.
- [20] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 357–368.
- [21] A. V. Aho and J. D. Ullman, *Foundations of computer science*. Computer Science Press, Inc., 1992.
- [22] D. Hitchcock, “The peculiarities of stoic propositional logic,” *Mistakes of reason: Essays in honour of John Woods*, pp. 224–242, 2005.

- [23] J. Marenbon, *Medieval Philosophy: an historical and philosophical introduction*. Routledge, 2006.
- [24] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Bibliopolis Naples, 1984, vol. 9.
- [25] T. Coquand, “Une théorie des constructions,” Ph.D. dissertation, Paris 7, 1985.
- [26] F. Pfenning and C. Paulin-Mohring, “Inductively defined types in the calculus of constructions,” in *International Conference on Mathematical Foundations of Programming Semantics*. Springer, 1989, pp. 209–228.
- [27] P. Letouzey, “A new extraction for coq,” in *International Workshop on Types for Proofs and Programs*. Springer, 2002, pp. 200–219.
- [28] H. Geuvers, “Proof assistants: History, ideas and future,” *Sadhana*, vol. 34, no. 1, pp. 3–25, 2009.
- [29] T. Coe, T. Mathisen, C. Moler, and V. Pratt, “Computational aspects of the pentium affair,” *IEEE Computational Science and Engineering*, vol. 2, no. 1, pp. 18–30, 1995.
- [30] C. Hoare, “An overview of some formal methods for program design,” *Computer*, no. 9, pp. 85–91, 1987.
- [31] J. C. Knight, C. L. DeJong, M. S. Gobble, and L. G. Nakano, “Why are formal methods not used more widely?” in *Fourth NASA formal methods workshop*. Citeseer, 1997.
- [32] G. Bella, L. C. Paulson, and F. Massacci, “The verification of an industrial payment protocol: The set purchase phase,” in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 12–20.
- [33] M. Fitting, *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.
- [34] T. Coquand and C. Paulin, “Inductively defined types,” in *COLOG-88*. Springer, 1990, pp. 50–66.
- [35] C. Paulin-Mohring, “Introduction to the calculus of inductive constructions,” 2015.

- [36] D. Delahaye, “A tactic language for the system coq,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2000, pp. 85–95.
- [37] S. Owre, J. M. Rushby, and N. Shankar, “Pvs: A prototype verification system,” in *International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.
- [38] K. Slind and M. Norrish, “A brief overview of hol4,” in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 28–32.
- [39] S. Owre, J. Rushby, N. Shankar *et al.*, “Pvs specification and verification system,” *URL: pvs. csl. sri. com*, 2001.
- [40] S. E. of Philosophy, “Church’s Type Theory,” <https://plato.stanford.edu/entries/type-theory-church/>, 2018, [Online; accessed 08/19/2018].
- [41] G. Gonthier, “The four colour theorem: Engineering of a formal proof,” in *Computer mathematics*. Springer, 2008, pp. 333–333.
- [42] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.
- [43] T. Coquand, “An analysis of girard’s paradox,” Ph.D. dissertation, INRIA, 1986.
- [44] Y. Bertot, “A short presentation of coq,” in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 12–16.
- [45] Y. Bertot and V. Komendantsky, “Fixed point semantics and partial recursion in coq,” in *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*. ACM, 2008, pp. 89–96.
- [46] W. A. Howard, “The formulae-as-types notion of construction,” *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.
- [47] M. P. Papazoglou, “Service-oriented computing: Concepts, characteristics and directions,” in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE, 2003, pp. 3–12.

- [48] C. Atkinson and T. Kuhne, “Model-driven development: a metamodeling foundation,” *IEEE software*, vol. 20, no. 5, pp. 36–41, 2003.
- [49] F. Arbab, “The iwim model for coordination of concurrent activities,” in *International Conference on Coordination Languages and Models*. Springer, 1996, pp. 34–56.
- [50] S.-S. T. Jongmans and F. Arbab, “Overview of thirty semantic formalisms for reo.” *Scientific Annals of Computer Science*, vol. 22, no. 1, 2012.
- [51] S. Navidpour and M. Izadi, “Linear temporal logic of constraint automata,” in *Advances in Computer Science and Engineering*. Springer, 2008, pp. 972–975.
- [52] F. Arbab and F. Mavaddat, “Coordination through channel composition,” in *International Conference on Coordination Languages and Models*. Springer, 2002, pp. 22–39.
- [53] F. Arbab and J. J. Rutten, “A coinductive calculus of component connectors,” in *International Workshop on Algebraic Development Techniques*. Springer, 2002, pp. 34–55.