Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

Recife

2020

Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

A B.Sc. Dissertation presented to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Universidade Federal de Pernambuco

Centro de Informática

Bachelor in Computer Engineering

Supervisor: Gustavo Henrique Porto de Carvalho

Recife

2020

Acknowledgements

Abstract

Theories of concurrency such as Communicating Sequential Processes (CSP) allow system specifications to be expressed clearly and analyzed with precision. However, the state explosion problem, common to model checkers in general, is a real constraint when attempting to verify system properties for large systems. An alternative is to ensure these properties via proof development. This work will provide an approach on how we can develop a theory of CSP in the Coq proof assistant, and evaluate how this theory compares to other theorem prover-based frameworks for the process algebra CSP. We will implement an infrastructure for declaring syntactically and semantically correct CSP specifications in Coq, along with native support for process representation through Labelled Transition Systems (LTSs), in addition to traces refinement analysis.

Keywords: Process algebra. LTS. Traces refinement. CSP. Proof assistant. Coq. QuickChick.

Resumo

Teorias de concorrência tais como *Communicating Sequential Processes* (CSP) permitem que especificações de sistemas sejam descritas com clareza e analisadas com precisão. No entanto, o problema da explosão de estados, comum aos verificadores de modelo em geral, é uma limitação real na tentativa de verificar propriedades de um sistema complexo. Uma alternativa é garantir essas propriedades através do desenvolvimento de provas. Este trabalho fornecerá uma abordagem sobre como se pode desenvolver uma teoria de CSP no assistente de provas Coq, além de compará-la com outros frameworks baseados em provadores de teoremas para a álgebra de processos CSP. Portanto, será implementada uma infraestrutura para declarar especificações sintatica e semanticamente corretas de CSP em Coq, juntamente com um suporte nativo para a representação de processos por meio de Sistemas de Transições Rotuladas (LTSs), além de análise de refinamento no modelo de *traces*.

Palavras-chave: Álgebra de processos. LTS. Refinamento no modelo de *traces*. CSP. Assistente de provas. Coq. QuickChick.

List of Figures

List of Tables

List of abbreviations and acronyms

CSP	Communicating Sequential Processes
FDR	Failures-Divergence Refinement
LTS	Labelled Transition System
SOS	Structured Operational Semantics
UTP	Unified Theories of Programming

Contents

1 Introduction

Concurrency is an attribute of any system that allows multiple components to perform operations at the same time. The understanding of this property is essential in modern programming because major areas, such as distributed and real-time systems, rely on this concept to work properly. As a result, the variety of applications enabled by the concurrency feature is broad: aircraft and industrial control systems, routing algorithms, peer-to-peer networks, client-server applications and parallel computation, to name a few.

Since concurrent systems may have parts that execute in parallel, the combination of ways in which these parts can interact raises the complexity in designing such systems. Phenomena like deadlock, livelock, nondeterminism and race condition can emerge from these interactions, so these issues must be addressed in order to avoid undesired behavior. Typically, testing cannot provide enough evidence to guarantee properties such as deadlock freedom, divergence freedom and determinism for a given system.

That being said, CSP (a theory for Communicating Sequential Processes) introduces a convenient notation that allows systems to be described in a clear and accurate way. More than that, it has an underlying theory that enables designs to be analysed and proven correct with respect to desired properties. The FDR (Failures-Divergence Refinement) tool is a model checker for CSP responsible for making this process algebra a practical tool for specification, analysis and verification of systems. System analysis is achieved by allowing the user to make assertions about processes and then exploring every possible behavior, if necessary, to check the truthfulness of the assertions made.

Although it is undeniable that FDR is a useful tool in the analysis of systems described in CSP, it has a limitation common to standard model checkers in general: the state explosion problem. An alternative way for deciding whether a system meets its specification is by proof development. Examples of this different approach are CSP-Prover and Isabelle/UTP, both frameworks based on the theorem prover Isabelle. Nevertheless, to the best of our knowledge, there is not a theory for CSP in the Coq proof assistant yet. Considering that, the main research question of this work is the following: how could we develop a theory of CSP in Coq, exploiting the main advantages of this proof assistant?

1.1 Objectives

The main objective (MO) of this work is to define in Coq a theory for concurrent systems, based on a limited scope of the process algebra CSP. This objective is unfolded into the following specific objectives (SO):

- SO1: study CSP and frameworks based on this process algebra.
- SO2: define a syntax for CSP in Coq, based on a restricted version of the CSP_M language (machine readable language for CSP).
- SO3: provide support for the LTS-based (Labelled Transition System) representation, considering the Structured Operational Semantics (SOS) of CSP.
- SO4: make use of the QuickChick tool to search for counterexamples of the traces refinement relation.

1.2 An overview of CSP_{Coq}

Consider the following CSP process adapted from Schneider (1999, p. 32, example 2.3). This process represents a cloakroom attendant that might help a costumer off or on with his coat, storing an retrieving coats as appropriate:

$$channel\ coat\_on, coat\_off, store, retrieve, request\_coat, eat$$

$$\begin{aligned} SYSTEM = {} & coat\_off -> store -> request\_coat -> retrieve -> coat\_on -> SKIP \\ & [|\ \{coat\_off, request\_coat, coat\_on\}\ |] \\ & coat\_off -> eat -> request\_coat -> coat\_on -> SKIP \end{aligned}$$

We can declare such system in CSP_{Coq} by defining a specification, which consists in lists of channels and processes. This specification must also abide by a set of contextual rules that will be discussed further in this work.

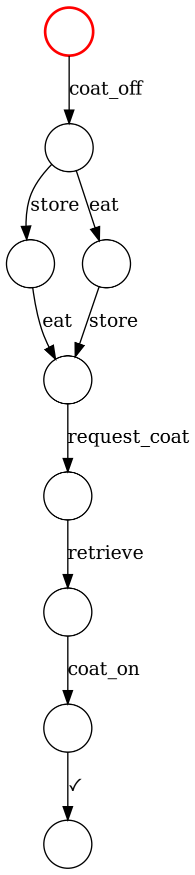
```
Definition example : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
    [ Channel {{"coat_off", "coat_on", "request_coat", "retrieve", "store", "eat"}} ]
    [ "SYSTEM" ::=
      "coat_off" --> "store" --> "request_coat" --> "retrieve" --> "coat_on" --> SKIP
      [| {{"coat_off", "request_coat", "coat_on"}} |]
      "coat_off" --> "eat" --> "request_coat" --> "coat_on" --> SKIP ]
  ).
Defined.
```

Furthermore, we can execute the following command to compute the process LTS and output the graph in dot language:

```
Compute generate_dot (compute_ltsR example "SYSTEM" 100).
```

The Figure 1 is a visual representation of the graph outputted by this command. The image is generated using GraphViz software.

Figure 1 – The process LTS graph.



1.3 Main contributions

The main contributions of this work are the following:

- Abstract and concrete syntax for a subset of CSP operators.
- Context rules for CSP specifications.
- Operational semantics via SOS approach.
- Inductive and functional definitions of a labelled transition system.
- Proof of correctness for the functional definition of the LTS.
- Inductive and functional definitions of traces.

- Proof of correctness for the functional definition of traces.
- Tactic macro that automates trace relation proofs.
- Formal definition of the traces refinement.
- Refinement verification using QuickChick.

1.4 Document structure

Apart from this introductory chapter, in which we discuss about the motivation behind this work and its main objective, and also take a quick look at an example that illustrates what can be done using the framework developed, this monograph contains three more chapters. The content of these chapters are detailed bellow:

Chapter 2 Discusses fundamental concepts such as CSP theory, SOS approach, trace refinement and LTS representation. Moreover, this chapter introduces the Coq proof assistant and its functional language Gallina, along with an introduction to proof development (tactics) and the Ltac language inside this tool, which gives support for developing tactic macros.

Chapter 3 Provides an in-depth look at the implementation of CSP$_{Coq}$, including its abstract and concrete syntax, and language semantics. Furthermore, the LTS process representation support, using the GraphViz software, is also detailed in this chapter.

Chapter 4 Concludes this monograph by presenting a comparison between the infrastructure described in this work and other interactive theorem provers based on CSP. It also addresses possible topics for future work.

2 Background

Before jumping into the specifics of the implementation of CSP$_{Coq}$, we need to understand some elements of the CSP language itself, such as the concrete syntax and the semantics defined in both denotational and operational models (section 2.1). Beyond that, it is also important to provide an overview of what an interactive theorem prover is: the Coq proof assistant fundamentals such as tactics, as well as the embedded Ltac language (section 2.2). We must also address the QuickChick property-based testing tool (section 2.3), which is the Coq implementation of QuickCheck (CLAESSEN; HUGHES, 2000). This chapter gives an introduction to each one of these concepts.

2.1 Communicating sequential processes

In 1978, Tony Hoare's *Communicating Sequential Processes* (HOARE, 1978) described a theory to help us understand concurrent systems, parallel programming and multiprocessing. More than that, it has introduced a way to decide whether a program meets its specification. This theory quickly evolved into what is known today as the CSP programming language. This language belongs to a class of notations known as process algebras, where concepts of communication and interaction are presented in an algebraic style.

Since the main goal of CSP is to provide a theory-driven framework for designing systems of interacting components an reasoning about them, we must introduce the concept of a component, or as we will be referencing it from now on, a *process*. Processes are self-contained entities that once combined they can describe a system, which is yet another larger process that may itself be combined as well with other processes.

The way a process communicates with the environment is through its *interface*. The interface of a process is the set of all the events that the process has the potential to engage in. At last, an *event* represents the atomic part of the communication itself. It is the piece of information the processes rely on to interact with one another. A process can either participate actively or passively in a communication, depending on whether it performed or suffered the action. Events may be external, meaning they appear in the process interface; indicate termination, represented by the event $\checkmark$; or be internal, and therefore unknown for the environment, denoted by the event τ.

The most basic process one can define is $STOP$. Essentially, this process never interacts with the environment and its only purpose is to declare the end of an execution. In other words, it illustrates a deadlock: a state in which the process can not engage in any event or make any progress whatsoever. It could be used to describe a computer that failed booting because one of its components is damaged, or a camera that can no longer take pictures due to storage space shortage.

Another basic process is $SKIP$. It indicates that the process has reached a successful termination state, which also means that it has finished executing. We can use $SKIP$ to illustrate an athlete that has crossed the finish line, or a build for a project that has passed.

Provided these two trivial processes, $STOP$ and $SKIP$, and the knowledge of what a process interface is, we can apply a handful of CSP operators to define more descriptive processes. For example, let a be an event in the process P interface. One can write the new process P as $a \rightarrow STOP$, meaning that this process behaves as $STOP$ after performing a. This operator is known as the *event prefix*, and it is pronounced as "then".

The choice between processes can be constructed in two different ways in CSP: externally and internally. An *external choice* between two processes implies the ability to perform any event that either process can engage in. Therefore, the environment has control over the outcome of such decision. On the other hand, if the process itself is the only responsible for deciding which event from its interface will be communicated, thus which process it will resolve to, then we call it an *internal choice*. Note that this operator is essentially a source of non-deterministic behavior.

To illustrate the difference between these choice operators, consider the following scenario: a cafeteria may operate by either letting the costumers choose between ice cream and cake for desert, or by making this choice itself (e.g. employees decide), having the clients no take on what deserts they will get. In the first specification, the choice is external to the business and it might be described as $ice\_cream \rightarrow SKIP \,\Box\, cake \rightarrow SKIP$, whereas it is internal in the latter, thus $ice\_cream \rightarrow SKIP \sqcap cake \rightarrow SKIP$ would capture such business rule.

CSP introduces two approaches for describing a parallel execution between processes: the *alphabetized parallel* and the *generalized parallel*. Let A be the interface of process P, and B the interface of process Q. An alphabetized parallel combination of these processes is described as $P \; {}_A\|_B \; Q$. Events in the intersection of A and B must be simultaneously engaged in by the processes P and Q. In other words, an event that appears in both process interfaces can only be communicated if the two processes are ready to perform this event. Any other event that does not match this criteria can be engaged in by its corresponding process independently.

The semantics are similar for the generalized version of the parallel operator. The only change being its constructor, that takes the synchronization alphabet alone as the interface argument the processes must agree upon. Let C be the intersection of previously defined interfaces A and B. The generalized parallel between process P and Q is written as $P \underset{C}{\|} Q$.

Both versions of the parallel operator may be used to describe a marathon where every participant is a process that runs in parallel with each other. They must all start the race at the same time, but they are not expected to cross the finish line all together. We can use the alphabetized parallel to specify the combination between two participants as $\mathit{RUNNER1}\ {}_{\{start,finish1\}}\|_{\{start,finish2\}}\ \mathit{RUNNER2}$, or use the generalized version of the operator instead: $\mathit{RUNNER1} \underset{\{start\}}{\|} \mathit{RUNNER2}$.

Another CSP operator that provides a concurrent execution of processes is the *interleaving* operator. Different from the parallel operators, the interleaving represents a combination of processes that do not require any synchronization at all. The processes applied to this operation execute totally independent of each other. This might be the case of two vending machines at a supermarket. They operate completely separate from each other, receiving payments, processing changes, and releasing snacks. In other words, there is no dependency regarding the communication of events between the vending machines. That being said consider the process $\mathit{VENDING\_MACHINE}$ as $\mathit{pay} \rightarrow \mathit{select\_snack} \rightarrow \mathit{return\_change} \rightarrow \mathit{release\_snack}$. Then, the process that specifies both machines operating together is described as $\mathit{VENDING\_MACHINE} \mathbin{|||} \mathit{VENDING\_MACHINE}$.

The last two operators we will be discussing are the *sequential composition* and *event hiding*. Before we continue, the reader must be aware that there are others CSP operators for combining processes apart from the ones presented in this chapter, but they will not be supported by the framework implemented in this project.

Sometimes it is necessary to pass the control over execution from one process to another, and for that we use sequential composition. It means that the first process has reached a successful termination state and now the system is ready to behave as the second process in the composition. For instance, parents can choose to let their children play only after completing their homework. That being the case, the process CHILD could be modeled as $\mathit{HOMEWORK}; \mathit{FUN}$, where

$$\mathit{HOMEWORK} = \mathit{choose\_subject} \rightarrow \mathit{study} \rightarrow \mathit{answer\_exercises} \rightarrow \mathit{SKIP}$$
$$\mathit{FUN} = \mathit{build\_lego} \rightarrow \mathit{watch\_cartoons} \rightarrow \mathit{play\_videogame} \rightarrow \mathit{SKIP}$$

In this example, the process FUN can only be executed after the process $\mathit{HOMEWORK}$ has successfully terminated.

Last but not least, we have the event hiding operator. A system designer may choose to hide events from a process interface to prevent them from being recognized by other processes. That way, the environment can not distinguish this particular event, thus no process can engage in it. Event hiding proves to be useful when processes placed in parallel should not be allowed to synchronize on certain events. Consider, for example, that a school teacher is communicating each student individually his or her test grade. It has to be done in such way that no student gets to know other test grades besides his or her own. The process *TEACHER* may be modeled as $show\_grade \rightarrow discuss\_questions \rightarrow SKIP$, so a teacher concerned with the students privacy can be described as $TEACHER \setminus \{show\_grade\}$.

2.1.1 Structured operational semantics

There are three major complementary approaches for describing and reasoning about the semantics of CSP programs. These are through *algebraic*, *denotational* (also called *behavioral*), and *operational semantics*. We will be focusing in the last one, which tries to understand all the actions and decisions that process implementations can make as they proceed.

The operational semantics for CSP language describes how a valid program is interpreted as sequences of computational steps. By evaluating the initial events of a process and finding out how it will behave immediately after performing them, this approach enables us to explore the state space of any process. All we need to do is repeat this step until we have covered the transition system picture of the process we are interested in.

It is traditional to present operational semantics as a logical inference system: Plotkin's SOS, or *Structured Operational Semantics* style. A process has a given action if, and only if, that is deducible from the rules given.

We start by analyzing the process *STOP*. Since it is unable to engage in any event whatsoever, there are no inference rules for it. Then, we move forward to the next primitive process: *SKIP*. While *STOP* has no actions of itself, *SKIP* is able to perform a single event, which is the termination event ✓. The lack of antecedents in the following rule means it is always the case that *SKIP* may perform ✓ and behave as *STOP*.

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

The event prefix operation also spares the antecedents in its inference rule, so the conclusion is immediately deduced: if the process is initially able to perform a, then after performing a it behaves like P.

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

The transition rules for external choice reflect the fact that the first external event resolves the choice in favor of the process performing the event. In addition, as we can see in the first two rules, the choice is not resolved on the occurrence of internal events.

Control over resolution of the choice is external because the events of both choices are initially available.

$$\frac{P \xrightarrow{\tau} P'}{P \mathbin{\Box} Q \xrightarrow{\tau} P' \mathbin{\Box} Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \mathbin{\Box} Q \xrightarrow{\tau} P \mathbin{\Box} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \mathbin{\Box} Q \xrightarrow{a} P'} \quad (a \neq \tau)$$

$$\frac{Q \xrightarrow{a} Q'}{P \mathbin{\Box} Q \xrightarrow{a} Q'} \quad (a \neq \tau)$$

The internal choice is an operation that guarantees the process to behave as either of its components on any execution. This state change happens “silently”, thus this transition is followed by the communication of internal event τ, as we can see in the inference rules for this operation.

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

We can separate the rules for the alphabetized parallel into two categories: one that describes the independent execution of each process, and other defining the synchronized step performed at once by the components. The first two inference rules capture the ability of both sides performing events that are not in the common interface, thus executing them independently. The third rule dictates the joint step, where both processes are able to perform the event, so they communicate it at the same time.

$$\frac{P \xrightarrow{\mu} P'}{P \ {}_A\|_B \ Q \xrightarrow{\mu} P' \ {}_A\|_B \ Q} \quad (\mu \in (A \cup \{ \tau \} \setminus B))$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \ {}_A\|_B \ Q \xrightarrow{\mu} P \ {}_A\|_B \ Q'} \quad (\mu \in (B \cup \{ \tau \} \setminus A))$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \ {}_A\|_B \ Q \xrightarrow{a} P' \ {}_A\|_B \ Q'} \quad (a \in A^{\checkmark} \cap B^{\checkmark})$$

The transition rules for the generalized parallel are very similar to the ones for the previous operation. The main difference lies in the side condition, since this version of parallelism is only interested in the interface alphabet. The same rule categories for the alphabetized parallel apply to this operation.

$$\frac{P \xrightarrow{\mu} P'}{P \underset{A}{\|} Q \xrightarrow{\mu} P' \underset{A}{\|} Q} \quad (\mu \notin A^{\checkmark})$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \underset{A}{\|} Q \xrightarrow{\mu} P \underset{A}{\|} Q'} \quad (\mu \notin A^{\checkmark})$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \underset{A}{\|} Q \xrightarrow{a} P' \underset{A}{\|} Q'} \quad (a \in A^{\checkmark})$$

The interleave operation describes a parallel execution between processes that do not synchronize in any event except termination $\checkmark$. In other words, this operation is a particular case of the generalized parallelism, where the interface alphabet is empty, thus the event $\checkmark$ being the only event that can be performed simultaneously by the components.

$$\frac{P \xrightarrow{\mu} P'}{P \mathbin{|||} Q \xrightarrow{\mu} P' \mathbin{|||} Q} \quad (\mu \neq \checkmark)$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \mathbin{|||} Q \xrightarrow{\mu} P \mathbin{|||} Q'} \quad (\mu \neq \checkmark)$$

$$\frac{P \xrightarrow{\checkmark} P' \qquad Q \xrightarrow{\checkmark} Q'}{P \mathbin{|||} Q \xrightarrow{\checkmark} P' \mathbin{|||} Q'}$$

As we already know, the hiding operator removes all events in a given alphabet from the process interface, preventing other processes to engage in them. The process to which the event hiding is applied can then behave just like it would without the operator, except the events in the given alphabet are made internal and then renamed to τ. Such behavior is capture by the inference rules:

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \quad (a \in A)$$

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \quad (\mu \notin A)$$

The last operational rules we need to discuss are for the sequential composition operator. Initially, this combination behaves as the process to the left of the operator until it terminates. Then, the execution control is granted to the other process in the composition. The control handover is represented by the communication of the internal event τ, as we can see in the second rule:

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q} \quad (a \neq \checkmark)$$

$$\frac{P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q}$$

2.1.2 Traces refinement

A pretty reasonable way for gathering information from a process interacting with the environment is by keeping track of the events this process engages in. This sequence of communication between process and environment, presented in a chronological order, is what we call a *trace*. Traces can either be finite or infinite, and it depends on the observation span and the nature of the process itself.

Because this record is easily observed by the environment and it represents a single interaction, it is often used to build models of CSP processes. As a matter of fact, there is one named after it: the *traces* model, represented by the symbol $\mathcal{T}$. It defines the meaning of a process expression as the set of sequences of events (traces) that the process can be observed to perform. This model is one of the three major denotational models of CSP, the other ones being the *stable failures* $\mathcal{F}$ and the *failures-divergences* model $\mathcal{N}$.

The notion of refinement is a particularly useful concept for specifying the correctness of a CSP process. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions as another, then we may replace a worse component by a better one without degrading the properties of the system.

Definition. (Traces Refinement) Let P and Q be two CSP processes, and *traces* be a function that yields the set of all possible traces of a given CSP process, we say that Q trace-refines P if, and only if, every trace of Q is also a trace of P:

$$P \sqsubseteq_{\mathcal{T}} Q \iff \mathit{traces}(Q) \subseteq \mathit{traces}(P)$$

If we consider P to be a specification which determines possible safe states of a system, then we can think of $P \sqsubseteq_{\mathcal{T}} Q$ as saying that Q is a safe implementation: no wrong events will be allowed.

2.1.3 Machine-readable version of CSP

In the beginning, CSP was typically used as a blackboard language. In other words, it was conceived to describe communicating and interacting processes for a human audience. Theories such as CSP have a higher chance of acceptance among the industry and academy (e.g. for teaching purposes) when they have tool support available. For that reason, the need of a notation that could actually be used with tools emerged.

The machine-readable CSP, usually denoted as CSP$_M$, not only provides a notation for tools such as FDR model-checker to be build upon but also extends the existing theory by using a functional programming language to describe and manipulate things like events and process parameters.

The Table 1 shows for every CSP process constructor discussed in section 2.1 the corresponding ASCII representation according to CSP$_M$ language.

Table 1 – The ASCII representation of CSP.

Constructor	Syntax	ASCII form
Stop	$STOP$	STOP
Skip	$SKIP$	SKIP
Event prefix	$e \to P$	e -> P
External choice	$P \Box Q$	P [] Q
Internal choice	$P \sqcap Q$	P |~| Q
Alphabetized parallel	$P \; {}_A\|\|_B \; Q$	P [A || B] Q
Generalized parallel	$P \underset{A}{\|\|} Q$	P [| A |] Q
Interleave	$P \; \|\|\| \; Q$	P ||| Q
Sequential composition	$P \; ; Q$	P ; Q
Event hiding	$P \setminus A$	P \ A

2.2 The Coq proof assistant

A proof assistant is a software for helping construct proofs of logical propositions. Essentially, it is a hybrid tool that automates the more routine aspects of building proofs while relying on human intervention for more complex steps. There is a variety of proof assistants including Isabelle, Agda, ATS, Idris and Coq, among others. This work is based around the Coq proof assistant.

Coq can be viewed as a combination of a functional programming language plus a set of tools for stating and proving logical assertions. Moreover, the Coq environment provides high-level facilities for proof development, including a large library of common definitions and lemmas, powerful tactics for constructing complex proofs semi-automatically, and a special-purpose programming language for defining new proof-automation tactics for specific situations.

Coq's native functional programming language is called *Gallina*. Before we discuss about the proof development aspect of this interactive theorem prover, we need to introduce the most essential elements we may find in a Gallina program. Consider the following definition of natural numbers in Coq:

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

This declaration tells Coq that we are defining a *type*. The capital-letter O constructor represents zero. When the S constructor is applied to the representation of the natural number n, the result is the representation of $n+1$, where S stands for "successor". An `Inductive` definition carves out a subset of the whole space of constructor expressions and gives it a name, in this case, *nat*.

Having defined *nat*, we can write functions that operate on natural numbers, such as the predecessor function:

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

Note that we do not need recursion to define the predecessor function, but simple pattern matching is not enough for more interesting computations involving natural numbers. For example, to check that a number n is even, we may need to recursively check whether $n-2$ is even. In order to do that, we use the keyword `Fixpoint` instead of `Definition`:

```
Fixpoint evenb (n:nat) : bool :=
  match n with
    | O ⇒ true
    | S O ⇒ false
    | S (S n') ⇒ evenb n'
  end.
```

Yet another way for defining evenness is through *inductive declaration*. Consider the following two rules: *the number* 0 *is even*, and *if* n *is even, then* S $(S\ n)$ *is even*. Lets call the first rule *ev_0* and then the second *ev_SS*. Using *ev* for the name of evenness property, we can write the following inference rules:

$$\frac{}{ev\ 0} \quad (ev\_0)$$

$$\frac{ev\ n}{ev\ (S\ (S\ n))} \quad (ev\_SS)$$

Now, we can translate these rules into a formal Coq definition. Each constructor in this definition corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

This definition is different from previous use of `Inductive`. We are defining a function from *nat* to *Prop*, in other words, a property of numbers. The type of each constructor must be specified explicitly (after a colon), and each constructor's type must have the form *ev n* for some natural number n.

2.2.1 Building proofs

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods. Proof development in Coq is done through a language of tactics that allows a user-guided proof process.

Recall the functional definition of evenness we introduced in the previous section, *evenb*. Suppose we want to prove that consecutive numbers have opposite parity. In other words, if $S\ n$ is even, then n is not, and if $S\ n$ is not even, then n is. One way to assert this statement is through the following proposition: $\forall\ (n : nat),\ evenb\ (S\ n) = negb\ (evenb\ n)$.

Eventually, during a proof development, one may find useful to make assertions about smaller intermediary steps of a theorem proof. This can be done either inside the main proof tree or in a completely separate one. The "divide and conquer" approach can help decreasing the number of steps in a proof and even reduce its overall complexity. In this example, we will first introduce a lemma to prove the involutive property of the negation function *negb*. This can be achieved in Coq with following commands:

```
Lemma negb_involutive : ∀ (b : bool),
  negb (negb b) = b.
Proof.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
```

Qed.

See the proof for Lemma *negb_involutive* in table 2

Table 2 – Proof of Lemma negb_involutive

Next step in Coq	Proof situation
$Proof.$	$forall\ b : bool, negb\ (negb\ b) = b$
$destruct\ b.$	$negb\ (negb\ true) = true$
	$negb\ (negb\ false) = false$
$-_{1/2}$	$negb\ (negb\ true) = true$
$simpl.$	$true = true$
$reflexivity.$	$-_{1/2}$ completed
$-_{2/2}$	$negb\ (negb\ false) = false$
$simpl.$	$false = false$
$reflexivity.$	$-_{2/2}$ completed, proof completed by Qed

End of proof of Lemma negb_involutive

The proof editing mode in Coq is entered whenever asserting a statement. Keywords such as `Lemma`, `Theorem`, and `Example` do so by allowing us to give the statement a name and the proposition we want to prove. Additionally, the commands `Proof` and `Qed` delimit, respectively, the beginning and the end of the sequence of tactic commands.

The keywords `destruct`, `simpl`, and `reflexivity` are examples of tactics. A tactic is a command that is used to guide the process of checking some claim we are making. The tactic `destruct` generates two sub-goals, one for each boolean value, which we must prove separately in order to prove the main goal. This strategy is also known as proof by case analysis.

The tactic `simpl` is often used in situations where we want to evaluate a compound expression, eventually reducing it to a simplified, easier-to-understand term. It facilitates our decisions in a proof development by resolving all the computations that can be done in a given state of the goal. Additionally, the tactic `reflexivity` finishes a proof by showing that both sides of an equation contain identical values.

Once we proved this lemma, it is now available to be used inside other proofs such as the one of the theorem we stated in the beginning of this section:

```
Theorem evenb_S : ∀ n : nat,
  evenb (S n) = negb (evenb n).
Proof.
  intros. induction n.
  - simpl. reflexivity.
  - simpl. simpl in IHn. rewrite IHn.
    rewrite negb_involutive. reflexivity.
Qed.
```

See the proof for Theorem *evenb_S* in table 3

Table 3 – Proof of Theorem evenb_S

Next step in Coq	Proof situation
Proof.	——— $forall\ n : nat,\ evenb\ (S\ n) = negb\ (evenb\ n)$
intros.	$n : nat$ ——— $evenb\ (S\ n) = negb\ (evenb\ n)$
induction n.	——— $evenb\ 1 = negb\ (evenb\ 0)$
	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ ——— $evenb\ (S\ (S\ n)) = negb\ (evenb\ (S\ n))$
$-_{1/2}$	——— $evenb\ 1 = negb\ (evenb\ 0)$
simpl.	——— $false = false$
reflexivity.	$-_{1/2}$ completed

Continuing proof of Theorem evenb_S on the next page

Table 3 – Proof of Theorem evenb_S continued

Next step in Coq	Proof situation
$-_{2/2}$	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ ——— $evenb\ (S\ (S\ n)) = negb\ (evenb\ (S\ n))$
$simpl.$	$n : nat$ $IHn : evenb\ (S\ n) = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ match\ n\ with\quad \mid 0 => false$ $\mid S\ n' => evenb\ n' \quad end$
$simpl\ in\ IHn.$	$n : nat$ $IHn : match\ n\ with\quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ match\ n\ with\quad \mid 0 => false$ $\mid S\ n' => evenb\ n' \quad end$
$rewrite\ IHn.$	$n : nat$ $IHn : match\ n\ with\quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = negb\ (negb\ (evenb\ n))$
$rewrite$ $negb\_involutive.$	$n : nat$ $IHn : match\ n\ with\quad \mid 0 => false \quad \mid S\ n' => evenb\ n'$ $end = negb\ (evenb\ n)$ ——— $evenb\ n = evenb\ n$
$reflexivity.$	$-_{2/2}$ completed, proof completed by Qed

End of proof of Theorem evenb_S

Note that we have added the quantifier $\forall\ n{:}nat$, so that our theorem talks about all natural numbers n. The tactic `intros` is responsible for moving the quantifier into the context of current assumptions.

This example demonstrates a proof by induction over natural numbers that is made possible in Coq by the `induction` tactic. Following this principle, to show that a proposition holds for all natural numbers n we must prove: the base case ($n = 0$), and then the induction step, which is, for any number n', if the proposition holds for n', then so it does for $S\ n'$.

The tactic `rewrite` tells Coq to perform a replacement in the goal, whether it is based on an assumption (hypothesis) from the proof context or a completely separate proof such as the Lemma *negb_involutive*.

Consider another theorem on natural numbers: for all n, if n is even, then the predecessor of the predecessor of n is also even. This theorem can be proved in Coq using the following commands:

```
Theorem ev_minus2 : ∀ n:nat,
  ev n → ev (pred (pred n)).
Proof.
  intros.
  destruct H.
  - simpl. apply ev_0.
  - simpl. apply H.
Qed.
```

See the proof for Theorem *ev_minus2* in table 4

Table 4 – Proof of Theorem ev_minus2

Next step in Coq	Proof situation
Proof.	——————————— $forall\ n : nat,\ ev\ n \ -> ev\ (pred\ (pred\ n))$
intros.	$n : nat$ $H : ev\ n$ ——————————— $ev\ (pred\ (pred\ n))$
destruct H.	——————————— $ev\ (pred\ (pred\ 0))$
	$n : nat$ $H : ev\ n$ ——————————— $ev\ (pred\ (pred\ (S\ (S\ n))))$
$-_{1/2}$	——————————— $ev\ (pred\ (pred\ 0))$

Continuing proof of Theorem ev_minus2 on the next page

Table 4 – Proof of Theorem ev_minus2 continued

Next step in Coq	Proof situation
simpl.	———————————— $ev\ 0$
apply ev_0.	$-_{1/2}$ completed
$-_{2/2}$	$n : nat$ $H : ev\ n$ ———————————— $ev\ (pred\ (pred\ (S\ (S\ n))))$
simpl.	$n : nat$ $H : ev\ n$ ———————————— $ev\ n$
apply H.	$-_{2/2}$ completed, proof completed by Qed

End of proof of Theorem ev_minus2

This time around, the tactic `intros` moves not only the quantifier into the context, but also the *hypothesis* that consists of the antecedent of the implication (*modus ponens*).

As we discussed before, the tactic `destruct` introduces proof by case analysis. In this example, it is responsible for generating, from the hypothesis, two sub-goals based on the inductive definition of evenness we have provided: one where $n = 0$ and the other where $n = S\ (S\ n)$. Once again, we must prove them separately so Coq accepts the theorem.

We then proceed to use the tactic `apply`, passing the term we find useful for proving each sub-goal as argument. In the first branch, where our goal is to prove $ev\ 0$, we apply the first rule of our inductive definition, *ev_0*, which concludes this sub-proof. In the second branch, we have $ev\ n$ as goal, which is already an assumption of ours (introduced to the proof context by the command `intros`). This also concludes the second sub-proof, thus proving the entire statement (theorem *ev_minus2*).

There are many other tactics and variations of them that can be used when proving a proposition. Apart from the ones we have already discussed in this subsection, other commonly used tactic commands are `unfold`, `inversion`, and `contradiction`. Further explanation on theses tactics will be provided as needed throughout the chapter 3.

2.2.2 The tactics language

Ltac is the tactic language for Coq. It provides the user with a high-level "toolbox" for tactic creation, allowing one to build complex tactics by combining existing ones with constructs such as conditionals, looping, backtracking, and error catching.

Imagine we want to prove that the number 4 does not appear in the list of consecutive natural numbers ranging from 0 to 3. By using the keyword `Example`, we can assert this statement in Coq and develop our proof:

```
Example elem_not_in_list : ¬ (In 4 [0 ; 1 ; 2 ; 3]).
Proof.
  unfold not. simpl. intros.
  destruct H.
  - inversion H.
  - destruct H.
    × inversion H.
    × destruct H.
      + inversion H.
      + destruct H.
        { inversion H. }
        { contradiction. }
Qed.
```

The first tactic unfolds the definition of $\neg$ (not) in the goal, replacing our initial statement by $In\ 4\ [0\ ;\ 1\ ;\ 2\ ;\ 3] \rightarrow False$. The second tactic reduces the new goal by computing the function *In*, which leaves us with the disjunctions $0 = 4 \vee 1 = 4 \vee 2 = 4 \vee 3 = 4 \vee False$ in the antecedent of the implication. Then, the tactic `intros` moves this antecedent to the proof context, introducing a new hypothesis H and leaving the literal *False* as goal.

From this point on, the proof develops a pattern: we perform a destruction followed by an inversion of the hypothesis until we can end the proof by contradiction. The recurrence of the tactic `destruct` lets us focus in one equality from the disjunction at a time: first the hypothesis becomes $0 = 4$, then $1 = 4$ and so on. The tactic `inversion` finishes each sub proof created by the previous command by deriving all the necessary conditions that should hold for the assumption to be proved. In this case, since none of these equalities is true, there is no condition that satisfies the proposition, thus proving our goal.

Since this pattern in the sequence of tactics is now exposed, we can define a tactic macro for proving propositions of the format "not in" using the keyword `Ltac`:

```
Ltac solve_not_in := unfold not;
```

```
  let H := fresh "H" in (
    intros H; repeat (contradiction + (destruct H; [> inversion H | ]))
  ).
Example elem_not_in_list' : ¬ (In 4 [0 ; 1 ; 2 ; 3]).
Proof. solve_not_in. Qed.
```

The new tactic *solve_not_in* works by first unfolding the function *not*, therefore deriving an implication, then moving the premise to the context of assumption, and finally repeating the following steps until the proof is finished: try to finish the proof by searching for a *contradiction* in the assumptions (such as a false hypothesis), if it fails, *destruct* the hypothesis and apply *inversion* to it in order to prove the first sub-goal yielded by the previous tactic.

2.3 QuickChick

QuickChick is a set of tools and techniques for combining randomized property-based testing with formal specification and proof in the Coq ecosystem. It is the equivalent of Haskell's QuickCheck for Coq proof assistant.

There are four basic elements in property-based random testing: an *executable property* such as for deciding whether a number is even, *generators* for random inputs to the property, *printers* for converting data structures like numbers to strings when reporting counterexamples, and *shrinkers*, which are used to search for minimal counterexamples when errors occur.

Consider the following example extracted from Lampropoulos e Pierce (2020). The function *remove* takes a natural number x and a list of natural numbers l and removes x from the list.

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | h::t ⇒ if h =? x then t else h :: remove x t
  end.
```

We can write assertions that represent our expectations regarding this function. One possible specification for *remove* might be this property:

```
Conjecture removeP : ∀ x l, ¬ (In x (remove x l)).
```

The keyword `Conjecture` treats our property *removeP* as an axiom. This proposition claims that x never occurs in the result of *remove* x l for any x and l. Such statement turns out to be false, as we would discover if we were to try to prove it. A different — perhaps much more efficient — way to discover the discrepancy between the definition and specification is to test it:

```
QuickChick removeP.
```

The *QuickChick* command takes an “executable” property and attempts to falsify it by running it on many randomly generated inputs, resulting in output like this:

```
0
[0, 0]
Failed! After 17 tests and 12 shrinks
```

This means that, if we run *remove* with x being 0 and l being the two-element list containing two zeros, then the property *removeP* fails.

With this example in hand, we can see that the *then* branch of remove fails to make a recursive call, which means that only one occurrence of x will be removed from the list. The last line of the output records that it took 17 tests to identify some fault-inducing input and 12 “shrinks” to reduce it to a minimal counterexample.

3 A theory for CSP in Coq

The chapter 2 provided an overview of the essential concepts for understanding the implementation of the CSP$_{Coq}$ language, setting the scene for an in-depth explanation of the language developed in this work. That being said, chapter 3 will explain how each of these concepts combined together made it possible to develop a theory of communicating sequential processes in Coq proof assistant.

The section 3.1 discusses the implementation of the language's abstract and concrete syntax, whereas section 3.2 explains how the SOS style of defining language semantics translates into Coq as an inductive declaration. Furthermore, section 3.3 provides details on both functional and inductive definitions of LTS, along with further explanation on the GraphViz software integration. Finally, section 3.4 guides the reader through the definition of traces refinement as an executable property and also presents randomized testing based on this property using QuickChick plugin.

3.1 Syntax

The CSP$_{Coq}$ language provides support to all process constructors and operations discussed in section 2.1. Those include the processes *STOP* and *SKIP*, and the operations event prefix, external choice, internal choice, alphabetized parallel, generalized parallel, interleave, sequential composition, and event hiding. This section introduces the reader to the implementation of both the abstract and concrete syntax of the CSP$_{Coq}$ language in Coq.

3.1.1 Abstract syntax

In order to define all these CSP operations in Coq, we declared the following inductive types: *event*, *event_tau_tick*, *channel*, *alphabet*, *proc_body*, and *proc_def*. The type *event* represents all external events. As we said before, external events are all events that are neither internal (τ) nor indicate termination ($\checkmark$). On the other hand, the type *event_tau_tick* provides not only a constructor for the external events, but also for the especial events τ and $\checkmark$.

For describing a set of external events, we have the *channel* and *alphabet* types. They are syntactically equivalent, the difference between these two constructors being purely semantic: the *channel* type is used to account for all external events that may be communicated in a CSP$_{Coq}$ specification, while the constructor provided by the *alphabet* type is applied, for example, to enumerate all external events in a process interface in an alphabetized parallel combination.

The *proc_body* and *proc_def* types describe, respectively, the constructors for the operations we mentioned earlier in the beginning of this subsection, and the process attribution statement. The constructors available for the *proc_body* type allow us to combine processes in order to create more complex ones, while the *proc_def* type provides a constructor that makes it possible to identify a process by a string, and that is, to give it a name.

The last element from the CSP syntax that we described is the *specification* type. A CSP specification can be perceived as a file containing multiple channels of events and process declarations. Ultimately, it is a context that holds information such as all the events that can be performed, and all process and corresponding definitions that compose a system. The way we introduce this concept in CSP$_{Coq}$ is via a record type. Records are constructions that allow the definition of a set of attributes and propositions that must be satisfied in order to successfully define this structure.

To illustrate the abstract syntax defined via inductive types in Coq, consider the CSP$_M$ processes

```
PRINTER := accept -> print -> STOP
PARKING_PERMIT_MCH := TICKET [ {cash, ticket} || {cash, change} ] CHANGE
```

and their representations in CSP$_{Coq}$ language

```
Proc “PRINTER” (ProcPrefix (Event “accept”) (ProcPrefix (Event “print”) STOP))

Proc “PARKING_PERMIT_MCH” (
  ProcAlphaParallel (ProcRef “TICKET”) (ProcRef “CHANGE”)
  (Alphabet (set_add event_dec “cash” (set_add event_dec “ticket” (empty_set event))))
  (Alphabet (set_add event_dec “cash” (set_add event_dec “change” (empty_set event))))
)
```

As one might notice from the examples above, the abstract syntax – though it dictates how well-formed expressions are constructed – is not a pleasant way of writing statements or at least reading them. For that matter, we need a more convenient notation. One that resembles the CSP$_M$ operators and, therefore, facilitates both implementation and understanding of CSP$_{Coq}$ processes and specifications.

3.1.2 Concrete syntax

In order to define a more appropriate notation for the CSP$_{Coq}$ language, the proof assistant's `Notation` command was used. This command allows the declaration of a new symbolic notation for an existing definition. The examples bellow demonstrate how this command is used to assign symbols (operators) to previously defined constructors.

```
Notation “a --> P” := (ProcPrefix a P) (at level 80, right associativity).
Notation “P [] Q” := (ProcExtChoice P Q) (at level 90, left associativity).
Notation “P [[ A \\ B ]] Q” := (ProcAlphaParallel P Q (Alphabet A) (Alphabet B)) (at
level 90, no associativity).
```

Along with the assignment of a notation symbol, we can specify its *precedence level* and its *associativity*. The precedence level helps Coq parse compound expressions, whereas the associativity setting helps to disambiguate expressions containing multiple occurrences of the same symbol. Coq uses precedence levels from 0 to 100, and left, right, or no associativity.

In the command lines above, the prefix operation has the higher precedence among all three operators and associates to the right. On the other hand, the external choice has a left associativity while the alphabetized parallel operator does not associate at all, meaning that parentheses are necessary to create a compound expression with multiple parallel operations.

Then, we can use these symbols to rewrite the process examples from the previous subsection in a much more friendly and recognizable way:

```
“PRINTER” ::= “accept” --> “print” --> STOP
“MACHINE” ::= ProcRef “TICKET” [[ {{“cash”, “ticket”}} \\ {{“cash”, “change”}} ]]
ProcRef “CHANGE”
```

The Table 5 displays a comparison between the CSP$_M$ operators we discussed and the CSP$_{Coq}$ language concrete syntax, achieved via Coq's `Notation` command:

3.2 Structured operational semantics

Now that we have a good understanding of not only how the abstract syntax but also a convenient notation of the CSP$_{Coq}$ language were implemented in the proof assistant, it is time for us to discuss language semantics in Coq. All the declarations presented in this section are based on the inference rules discussed in subsection 2.1.1. More specifically, we will address how those SOS rules from the previous chapter were ported into Coq's environment.

Table 5 – The CSP$_{Coq}$ concrete syntax.

Constructor	CSP$_M$	CSP$_{Coq}$
Stop	STOP	STOP
Skip	SKIP	SKIP
Event prefix	e -> P	e --> P
External choice	P [] Q	P [] Q
Internal choice	P |~| Q	P |~| Q
Alphabetized parallel	P [A || B] Q	P [[A \\ B]] Q
Generalized parallel	P [| A |] Q	P [| A |] Q
Interleave	P ||| Q	P ||| Q
Sequential composition	P ; Q	P ;; Q
Event hiding	P \ A	P \ A
Process definition	P := Q	P ::= Q
Process name	P	ProcRef "P"

Recall the inductive definition of the evenness property exemplified in section 2.2. In that example, it was possible to rewrite the inference rules for such property in terms of an inductive declaration in Coq, where each rule was translated into a propositional statement, more specifically, a logical implication. We will use the same approach to define the semantic rules of the CSP$_{Coq}$ language.

Initially, a notation was defined to represent the SOS relation, in order to increase the readability of the inductive declaration. Thus, this new notation could be used in the constructors of the relational definition. The following Coq command line, similarly to the ones in the previous section, creates the infix notation "S # P // a ==> Q", which can be pronounced "in the specification *S*, the process *P*, after communicating *a*, behaves like the process *Q*":

```
Reserved Notation "S '#' P '//' a '==>' Q" (at level 150, left associativity).
```

To exemplify the usage of this notation inside the inductive definition, lets revisit the inference rule for the prefix operator:

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

This structure can be translated into Coq as a single constructor in the SOS relation:

```
| prefix_rule (S : specification) (P : proc_body) (a : event) :
```

```
S # (a --> P) // Event a ==> P
```

As far as the notation goes, this constructor can be interpreted as follows: given a specification S, a process P, and an event a, in the context of S, the process a --> P, after communicating event a, behaves as the process P. In other words, it is always true that, in the prefix operation $a \rightarrow P$, after a is performed, resolves to P.

Consider the following examples of constructors from the SOS relation. They refer to the external choice and alphabetized parallelism operations respectively. Note that each one of these operations need more than one inference rule in order to fully describe their semantics, as explained in subsection 2.1.1. However, the definitions we are about to discuss only consider two of these rules: one that evaluates the external choice to the left-hand side operand, and the joint step rule of the alphabetized parallelism operation, that represents the synchronization event between the processes.

```
| ext_choice_left_rule (S : specification) (P Q : proc_body) :
  ∀ (P' : proc_body) (a : event_tau_tick),
      ¬ eq a Tau →
      (S # P // a ==> P') →
      (S # P [] Q // a ==> P')
| alpha_parall_joint_rule (S : specification) (P Q : proc_body) (A B : set event) :
  ∀ (P' Q' : proc_body) (a : event),
      set_In a (set_inter event_dec A B) →
      (S # P // Event a ==> P') →
      (S # Q // Event a ==> Q') →
      S # P [[ A \\ B ]] Q // Event a ==> P' [[ A \\ B ]] Q'
```

The *ext_choice_left_rule* constructor encodes the inference rule that solves the external choice operation for the left operand. As we saw earlier, this behavior is described in the SOS by the rule

$$\frac{P \xrightarrow{a} P'}{P \mathrel{\Box} Q \xrightarrow{a} P'} \quad (a \neq \tau)$$

which translates into the logical proposition

```
¬ eq a Tau → (S # P // a ==> P') → (S # P [] Q // a ==> P')
```

In this statement, we can see that the first term corresponds to the side condition of the inference rule, which guarantees that the event in question is not the internal event τ. The second term is the main premise of the rule, ensuring that it is possible for the event a to evolve the process P into P' in the specification S. Together, the side condition and hypothesis establish the necessary conditions to resolve the external choice operation to the left hand operand.

Similarly, the inference rule that defines the synchronous communication of an event by two processes combined by the alphabetized parallelism operation, is described in Coq by the constructor *alpha_parall_joint_rule*. This rule, as we can see from its sequent notation below, has a side condition which guarantees that the event in question belongs to the intersection of the processes interfaces. Furthermore, it has two premises, which ensure this event is able to evolve both operands of the combination, that is, the processes to the left and to the right of the parallelism operator can communicate the event.

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \;{}_A\|_B\; Q \xrightarrow{a} P' \;{}_A\|_B\; Q'} \quad (a \in A^{\checkmark} \cap B^{\checkmark})$$

The side condition and the premises are rewritten in the inductive definition as antecedents of a logical implication:

```
set_In a (set_inter event_dec A B) →
(S # P // Event a ==> P') →
(S # Q // Event a ==> Q') →
S # P [[ A \\ B ]] Q // Event a ==> P' [[ A \\ B ]] Q'
```

Note that this constructor does not consider the ✓ event as a possible synchronization event between the processes, even though it is included in the inference rule (side condition) we saw above. The reason for this is that we defined another constructor, *alpha_parall_tick_joint_rule*, dedicated exclusively to this joint step allowed by the termination event.

The last example of constructor we want to highlight from the inductive definition of the SOS is the *process reference* operation. This constructor defines the rules for unfolding a process definition inside the body of another process:

```
| reference_rule (S : specification) (P : proc_body) (name : string) :
  ∀ (Q : proc_body),
      eq P (ProcRef name) →
      eq (get_proc_body S name) (Some Q) →
      S # P // Tau ==> Q
```

As we can see, there are two premises for successfully unfolding a definition. First, the process that needs unfolding must be of the form *ProcRef name*, where *ProcRef* is a constructor of the type *proc_body*, and *name* is a string that identifies the process in a specification. Second, the process referenced by *ProcRef name* must be defined in the specification. To that end, the function *get_proc_body* searches for this definition inside the specification, while the equality ensures this search results in "*Some Q*", where Q is a process body. It is important to emphasize that such definition differs from the one implemented by the FDR tool. This approach will allow an ill-formed recursion such as

$P := P$ to be declared in CSP$_{Coq}$, introducing a τ action to represent the effort of unfolding the definition and, therefore, increasing the process LTS size.

3.3 Labelled transition systems

Now that we can define syntactically and semantically correct CSP processes in Coq through the definitions we declared so far, we will introduce yet another way of implementing processes, based on the idea of labelled transition systems. This section will cover how the LTS concept was embedded in the proof assistant via functional and inductive definitions, and how CSP$_{Coq}$ provides support to the GraphViz software, which enables a graph visualization of CSP processes.

A labelled transition system consists of a non-empty set of states S, with a designated initial state P_0, a set of labels L, and a ternary relation $P \xrightarrow{x} Q$ meaning that the state P can perform an action labelled x and move to state Q. In Coq, we defined a LTS in two different and complementary ways: a computable function and a relation.

To illustrate these definitions, consider the following CSP$_{Coq}$ specification:

```
Definition SPEC : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
    [ Channel {{"a", "b", "c"}} ]
    [ "P" ::= ("a" --> "b" --> STOP) [] ("c" --> STOP) ]
  ).
Defined.
```

Initially, the process "P" can perform either "a" or "c". If "a" gets communicated, the external choice resolves to the process "b" --> STOP, which can then perform "b" and finish the execution. On the other hand, if the event "c" gets communicated instead, then the combination resolves to the process STOP and also terminates. Therefore, we can list three transitions for the process "P": from state ("a" --> "b" --> STOP) [] ("c" --> STOP) to state "b" --> STOP with event "a", from state ("a" --> "b" --> STOP) [] ("c" --> STOP) to state STOP with event "c", and from state "b" --> STOP to state STOP with event "b". To better describe these transitions, we will be using the 3-tuple notation (P, a, Q), where P is the source state, a is the action (event), and Q is the target state.

We can apply the same intuition from this example in the implementation of a functional definition of LTS in Coq: starting with the initial state, compute all the immediate transitions available from this state, meaning all 3-tuples where the target states can be reached in one step from S_0. Mark the initial state as "visited" and any other state discovered in the previous step as “not visited”. Then, repeat this step for each unvisited state until there are no states to be visited anymore, keeping track of the states you have already visited and the ones that haven’t been visited yet.

Let *compute_ltsR* be the function yielded by this algorithm, the following Coq command outputs the set of transitions from the LTS of the process “P”:

```
Compute compute_ltsR SPEC “P” 1000.

= Some
      [(“a” --> “b” --> STOP [] “c” --> STOP, Event “a”, “b” --> STOP);
      (“a” --> “b” --> STOP [] “c” --> STOP, Event “c”, STOP);
      (“b” --> STOP, Event “b”, STOP)]
: option (set transition)
```

Note that we added an integer as the third parameter of the function *compute_ltsR*. Since every recursive definition (`Fixpoint`) in Coq must have an argument that explicitly decreases in each recursive call to ensure termination, the third parameter fulfills this premise. Therefore, the number of recursive calls is limited in order to prevent endless computations: this function only returns a set of transitions when it’s able to visit all the states of the LTS before reaching the limit.

Once we have our result, we may still want to check whether this set of transitions is indeed the LTS of the process in question. One way to prove this is by demonstrating that for every process P and Q, and event a, P behaves like Q after communicating a if, and only if, the 3-tuple (P, a, Q) belongs to the set of transitions. In other words, if T is the set of transitions in the LTS representation of the process P, then every possible communication of P – and its components (inner processes) – is present in T, and every transition in T is a valid communication of process P (or the sub-processes that make up P). With that in mind, an inductive definition might be useful to prove this kind of assertion.

```
Inductive ltsR' :
  specification →
  set transition →
  set proc_body →
  set proc_body →
  Prop :=
  | lts_empty_rule (S : specification) (visited : set proc_body) :
      ltsR' S nil nil visited
  | lts_inductive_rule
        (S : specification)
```

```
    (T : set transition)
    (P : proc_body)
    (tl visited : set proc_body) :
  let T' := transitions_from P T in
  let T'' := set_diff transition_eq_dec T T' in
  let visited' := set_add proc_body_eq_dec P visited in
  let to_visit := set_diff proc_body_eq_dec
                    (set_union proc_body_eq_dec tl (target_proc_bodies T'))
                    visited' in
  (∀ (a : event_tau_tick) (P' : proc_body),
    (S # P // a ==> P') ↔ In (P,a,P') T') →
  ltsR' S T'' to_visit visited' →
  ltsR' S T (P :: tl) visited.
Definition ltsR (S : specification) (name : string) (T : set transition) : Prop :=
  match get_proc_body S name with
  | Some body ⇒ NoDup T ∧ ltsR' S T [body] nil
  | None ⇒ False
  end.
```

The constructor *lts_empty_rule* implies that the LTS relation holds when there are neither states to be visited nor transitions to validate anymore. On the other hand, the inductive step is described by the constructor *lts_inductive_rule*, which proposes that T represents the set of transitions of the LTS of process P if the following two premises are satisfied:

- Let T' be the subset of T containing all transitions from state P, $P \xrightarrow{a} Q \iff (P, a, Q) \in T'$;
- Let T'' be the set of all remaining transitions and S the set of states yet to be visited (union of the set of new states that can be reached from P and other unvisited states), the LTS relation must hold also for the sets T'' and S.

3.3.1 GraphViz integration

GraphViz is an open source graph visualization software that takes descriptions of graphs in a simple text language – in our case, the DOT language – and make diagrams in useful formats, such as images, SVG, and PDF. Moreover, this tool allows the customization of these diagrams, such as options for colors, fonts, hyperlinks, and custom node shapes. Providing support for this software enables CSP$_{Coq}$ programmers to generate a graph from the description of a LTS with one line of command. That way, the behavior of a process may be inspected and analyzed more easily, since a visual representation will be available.

In order to build the description of a graph in DOT language, it is necessary to define a mapping of the constructors of *event_tau_tick* and *proc_body* types to the *string* type. That way, it will be possible to convert each tuple from the LTS into a statement that corresponds to a transition of a graph in DOT syntax. One way to achieve this in Coq is through the `Coercion` command, which assigns a conversion function that maps an expression of one type into another.

Consider the following coercion from type *event_tau_tick* to type *string*:

```
Definition event_tau_tick_to_str (e : event_tau_tick) : string :=
  match e with
  | Tau ⇒ "τ"
  | Tick ⇒ "✓"
  | Event a ⇒ a
  end.
Coercion event_tau_tick_to_str : event_tau_tick >-> string.
```

As we can see from the code snippet above, initially, we defined a function that takes an element of type *event_tau_tick* and returns a string representation of that element. Then, a coercion between the two types is established: from this point on, Coq will convert an element of type *event_tau_tick* into a string whenever it is necessary.

Once the coercions are configured, we can define a function to create the description of a graph in the DOT language from the set of tuples that make up the LTS. The intuition behind this function is as follows: the first tuple to be processed has its starting state identified as the graph's initial node (bold and red border), and each tuple – the first included – is rewritten as a DOT syntax transition statement. In other words, the tuple (P, e, Q) generates the string "<P> -> <Q> [label = <e>];", as shown in the definitions below:

```
Fixpoint generate_dot' (lts : set transition) : string :=
  match lts with
  | nil ⇒ ""
  | (P, e, Q) :: tl ⇒ " <" ++ P ++ "> -> <" ++ Q ++ ">" ++ " [label=<" ++ e ++
">];" ++ (generate_dot' tl)
  end.
Definition style_initial_state (P : proc_body) : string := "<" ++ P ++ "> [style=bold,
color=red];".
Definition generate_dot (lts : option (set transition)) : string :=
  match lts with
  | Some ((P, e, Q) :: tl) ⇒
    "digraph LTS { " ++ (style_initial_state P) ++ (generate_dot' ((P, e, Q) :: tl)) ++ "
}"
  | _ ⇒ ""
  end.
```

To demonstrate what these definitions can achieve, recall the process MACHINE from the section 3.1.2. Here is a complete definition of that process in CSP$_{Coq}$:

```
Definition PARKING_PERMIT_MCH : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
    [ Channel {{"cash", "ticket", "change"}} ]
    [ "TICKET" ::= "cash" --> "ticket" --> ProcRef "TICKET"
    ; "CHANGE" ::= "cash" --> "change" --> ProcRef "CHANGE"
    ; "MACHINE" ::= ProcRef "TICKET" [[ {{"cash", "ticket"}} \\ {{"cash", "change"}}
]] ProcRef "CHANGE" ]
  ).
Defined.
```

To generate the description of the graph in DOT syntax for the process MACHINE, simply run this command in Coq:

```
Compute generate_dot (compute_ltsR PARKING_PERMIT_MCH "MACHINE" 1000).
```

We can then save the string output to a file named LTS.gv and run the following command in a terminal with the GraphViz software installed:

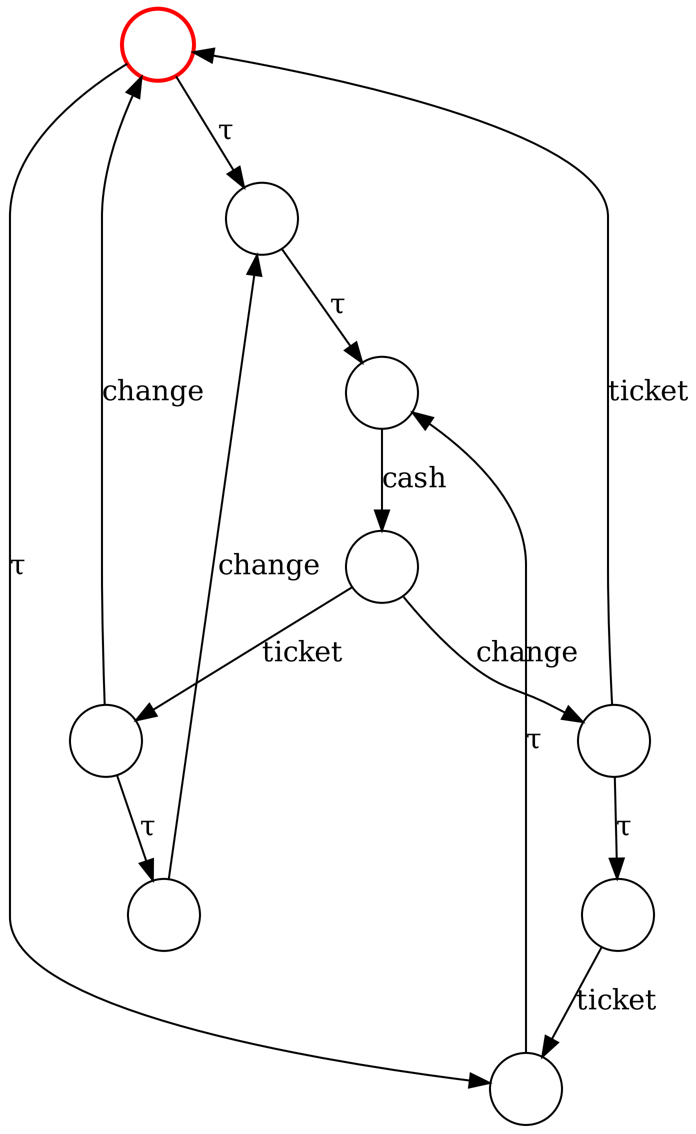
```
$ dot -Nlabel="" -Nshape=circle -Tjpeg LTS.gv -o LTS.jpeg
```

The Figure 2 shows the expected output: an image in JPEG format containing a graph – with nodes shaped as circles and no labels – that represents the process LTS:

3.4 Traces refinement

As we saw earlier in subsection 2.1.2, a simple but effective way to analyze the behavior of a process is through the sequence of externally visible events such process is capable of performing over time. This communication history is called *trace*. Using the concepts we saw previously, we may perceive trace as a list of actions that takes from one state to another in a LTS, or even the sequence of external events that one process can be communicate according to the operational semantics.

Figure 2 – Example LTS: The process MACHINE.



3.4.1 Traces-related concepts

Like the other types of CSP language we have discussed so far in this work, the trace also has its counterpart in CSP$_{Coq}$. It is defined in terms of a list of events that can be observed by the environment, that is, neither τ nor ✓ can be present in that list. This restriction is guaranteed in Coq by choosing the inductive type *event* in the trace definition, instead of the type *event_tau_tick*, which also includes internal and termination events. Below, we can check the definition in the proof assistant:

```
Definition trace := list event.
```

After defining this type, we can then declare trace as a relationship between a process P and a list of events L, so that it is possible to make logical propositions that involve whether or not L is a trace of P. Initially, we can describe four inference rules that define this relation:

- the empty list is a trace of any process;

$$\frac{}{\textit{trace}\ P\ \textit{nil}}$$

- a non-empty list L is a trace of a process P if the first element of L can be performed by P and the remaining events of this list is a trace of the process resulting from this communication;

$$\frac{P \xrightarrow{h} Q \qquad \textit{trace}\ Q\ \textit{tl}}{\textit{trace}\ P\ (h :: \textit{tl})} \quad (h \notin \{\checkmark,\ \tau\})$$

- a list L is a trace of a process P if ✓ can be performed by P and L is a trace of the process resulting from this communication;

$$\frac{P \xrightarrow{\checkmark} Q \qquad \textit{trace}\ Q\ L}{\textit{trace}\ P\ L}$$

- similarly, a list L is a trace of a process P if τ can be performed by P and L is a trace of the process resulting from this communication.

$$\frac{P \xrightarrow{\tau} Q \qquad \textit{trace}\ Q\ L}{\textit{trace}\ P\ L}$$

To illustrate this concept, consider once again our process MACHINE, whose LTS is represented by the graph in Figure 2. One can easily verify, using that image, that the sequence of events <cash, ticket, change> is a trace of that process. First, the process communicates the internal event τ twice, in order to unfold both sides of the parallel combination. Then, it proceeds to communicate the event cash, as the joint step of the process MACHINE components (processes TICKET and CHANGE). Finally, the left-hand side of the process compound may communicate first, performing the event ticket; followed by event change, communicated by the right-hand side of the operator. Remember that a trace consists of externally visible events only, therefore the first two communicated events, which both happen to be τ, do not appear in the trace.

Now, our knowledge of inductive declaration in Coq will be useful once again to embed these rules in the tool – each one of these can be translated into a constructor from the *traceR'* definition:

```
Inductive traceR' : specification → proc_body → trace → Prop :=
  | empty_trace_rule (S : specification) (P : proc_body) :
    traceR' S P nil
  | event_trace_rule (S : specification) (P P' : proc_body) (h : event) (tl : trace) :
    (S # P // Event h ==> P') →
    traceR' S P' tl →
    traceR' S P (h::tl)
  | tick_trace_rule (S : specification) (P P' : proc_body) (t : trace) :
    (S # P // Tick ==> P') →
    traceR' S P' t →
    traceR' S P t
  | tau_trace_rule (S : specification) (P P' : proc_body) (t : trace) :
    (S # P // Tau ==> P') →
    traceR' S P' t →
    traceR' S P t.
Definition traceR (S : specification) (proc_name : string) (t : trace) :=
  match (get_proc_body S proc_name) with
  | Some body ⇒ traceR' S body t
  | None ⇒ False
  end.
```

Note that the inference rules described above, in the order they were presented, correspond, respectively, to the constructors *empty_trace_rule, event_trace_rule*, *tick_trace_rule*, and *tau_trace_rule* of the *traceR'* inductive definition.

This definition of trace as a relation, as well as the definition of operational semantics in the SOS style, allows us not only to make assertions like *traceR S "P" L*, given a specification S, a process P and a list of events L, but also prove them in Coq by applying the constructors from the inductive declarations available.

It turns out that, not only these proofs, that verify the trace relationship between a process and a list of events, grow in proportion to the size of the list you want to check, they also become quite repetitive. Fortunately, this repetition allows us to define an algorithm to automatically prove assertions of this kind, and thus create a tactic macro for that purpose.

This algorithm can be defined as follows: if the goal is to prove the trace relation for an empty list, apply the rule *empty_trace_rule* and fishing the proof, otherwise, try to make progress by applying each of the other three constructors of the *traceR'* definition, making a recursive call afterwards. For goals which are operations described by the CSP$_{Coq}$ semantics, try to make progress using the constructors applicable for that operation based on pattern matching, making once again a recursive call at the end of this step.

The following definition uses the Ltac language to describe this algorithm as a tactic macro that automatically proves statements of the kind *traceR S "P" L*:

```
Ltac solve_trace' :=
  multimatch goal with
  | ⊢ traceR' _ _ nil ⇒ apply empty_trace_rule
  | ⊢ traceR' _ _ _ ⇒
    (eapply tau_trace_rule
    + eapply tick_trace_rule
    + eapply event_trace_rule); solve_trace'
  | ⊢ _ # _ -> _ // _ ==> _ ⇒ apply prefix_rule
  | ⊢ _ # ProcRef _ // _ ==> _ ⇒ eapply reference_rule; solve_trace'
  | ⊢ _ # _ [] _ // _ ==> _ ⇒
    (eapply ext_choice_tau_left_rule
    + eapply ext_choice_tau_right_rule
    + eapply ext_choice_left_rule
    + eapply ext_choice_right_rule); solve_trace'
  | ⊢ _ # _ |~| _ // _ ==> _ ⇒
    (eapply int_choice_left_rule
    + eapply int_choice_right_rule); solve_trace'
  | ⊢ _ # _ [[ _ \\ _ ]] _ // _ ==> _ ⇒
    (eapply alpha_parall_tau_indep_left_rule
    + eapply alpha_parall_tau_indep_right_rule
    + eapply alpha_parall_tick_joint_rule
    + eapply alpha_parall_joint_rule
    + eapply alpha_parall_indep_left_rule
    + eapply alpha_parall_indep_right_rule); solve_trace'
  | ⊢ _ # _ [| _ |] _ // _ ==> _ ⇒
    (eapply gener_parall_tau_indep_left_rule
    + eapply gener_parall_tau_indep_right_rule
    + eapply gener_parall_tick_joint_rule
    + eapply gener_parall_joint_rule
    + eapply gener_parall_indep_left_rule
    + eapply gener_parall_indep_right_rule); solve_trace'
  | ⊢ _ # _ ||| _ // _ ==> _ ⇒
    (eapply interleave_tick_rule
    + eapply interleave_left_rule
    + eapply interleave_right_rule); solve_trace'
  | ⊢ _ # _ ;; _ // _ ==> _ ⇒
    (eapply seq_comp_tick_rule
    + eapply seq_comp_rule); solve_trace'
  | ⊢ _ # _ \ _ // _ ==> _ ⇒
    (eapply hiding_tau_tick_rule
    + eapply hiding_not_hidden_rule
    + eapply hiding_rule); solve_trace'
  | ⊢ _ # SKIP // Tick ==> _ ⇒ apply success_termination_rule
  | ⊢ _ ≠ _ ⇒ unfold not; let H := fresh "H" in (intros H; inversion H)
  | ⊢ _ = _ ⇒ reflexivity
  | ⊢ set_In _ _ ⇒ simpl; solve_trace'
  | ⊢ ¬ set_In _ _ ⇒ solve_not_in
  | ⊢ _ ∨ _ ⇒ (left + right); solve_trace'
  end.
Ltac solve_trace := unfold traceR; simpl; solve_trace'.
```

The following proof revisits the trace example we discussed earlier in this section. Using our tactic macro *solve_trace*, we can automatically prove that the list of events <cash, ticket, change> is indeed a trace of the process MACHINE (Figure 2):

```
Example MACHINE_TRACE :
  traceR PARKING_PERMIT_MCH "MACHINE" ["cash" ; "ticket" ; "change"].
Proof. solve_trace. Qed.
```

Provided the definitions we presented so far, we can finally formulate in Coq the concept of refinement according to the traces model, along with an appropriate notation for this relationship:

```
Definition trace_refinement (S : specification) (Spec Imp : string) : Prop :=
  ∀ (t : trace), traceR S Imp t → traceR S Spec t.
Notation "S '#' P '[T=' Q" := (trace_refinement S P Q) (at level 150, left associativity).
```

As we learned in subsection 2.1.2 of the previous chapter, this definition ensures that an implementation Q refines a specification P, according to the traces model, if, and only if, every trace of Q is also a trace of P.

3.4.2 QuickChick integration

In the context of this work, the QuickChick library was used to random test the refinement property according to the traces model. That being said, our goal is to obtain a simple and automated way to test this property that relates two processes, Imp and Spec, eventually finding examples that prove the opposite. In other words, this checker guarantees that, if there is a failure during the tests, it is proven by counterexample that the refinement relation between the two processes inputted does not hold.

In order to create the checker, first we need to define a generator of random inputs, which, in our case, consists of traces of a given process. Thus, we define a random generator that takes a “specification” process and an arbitrary integer to limit the number of events in the trace. This generator returns a trace of the given process, which size is limited by the third parameter:

```
Fixpoint gen_valid_trace'
  (S : specification)
  (P : proc_body)
  (size : nat)
  : G (option semantics_trace.trace) :=
  match size with
  | O ⇒ ret nil
  | S size' ⇒
    freq_ (ret nil) [
      (1, ret nil) ;
      (size,
        bind (gen_valid_trans S P) (
          fun t ⇒ (
            match t with
            | nil ⇒ ret nil
            | (Event e, Q) :: _ ⇒
              bind (gen_valid_trace' S Q size') (
                fun ts ⇒ ret (e :: ts)
              )
            | (_, Q) :: _ ⇒
              bind (gen_valid_trace' S Q size') (
                fun ts ⇒ ret ts
              )
            end
          )
        )
      )
    ]
  end.

Definition gen_valid_trace
  (S : specification)
  (proc_id : string)
  (size : nat)
  : G (option semantics_trace.trace) :=
  match get_proc_body S proc_id with
  | None ⇒ ret None
  | Some P ⇒ gen_valid_trace' S P size
  end.
```

This generator operates as follows: while the third parameter is greater than zero, decide – based on probability – whether the generation should stop; in case it should continue, generate a transition from the current state of the process, decreasing the argument that limits the trace size; call the generator recursively, passing the state reached by the transition generated in the previous step; finally, return the concatenation of the event from this transition with the result of the recursive call. The function *freq_* is responsible for making the probabilistic choice. In our case, the option to end the generation before reaching the limit has $((1/(size+1))*100)\%$ chance to happen, while continuing with the generation has the probability of $((size/(size+1))*100)\%$.

The command that samples traces according to the generator we defined, as well as the output of this sampling process are exemplified bellow:

```
Sample (gen_valid_trace PARKING_PERMIT_MCH "MACHINE" 10).
```

Output:

```
[Some []; Some ["cash"; "change"; "ticket"; "cash"; "change"];
Some ["cash"]; Some ["cash"; "ticket"; "change"; "cash"]; Some [];
Some ["cash"; "change"; "ticket"; "cash"]; Some ["cash"; "change"; "ticket"]; ...]
```

This definition, however, depends on another generator of random input: *gen_valid_trans*. This generator receives a specification and a process body, and returns a list containing exactly one valid random transition from the given state, where the transition is represented by the pair (action, target_state):

```
Definition gen_valid_trans
  (S : specification)
  (P : proc_body)
  : G (option (list (event_tau_tick × proc_body))) :=
  match get_transitions S P with
  | None ⇒ ret None
  | Some nil ⇒ ret nil
  | Some (t :: ts) ⇒ bind (elems_ t (t :: ts)) (fun a ⇒ ret (Some [a]))
  end.
```

The following lines in Coq exemplify the usage of this function that generates valid transitions:

```
Sample (gen_valid_trans PARKING_PERMIT_MCH
  (ProcRef “TICKET” [[ {{“cash”, “ticket”}} \\ {{“cash”, “change”}} ]] ProcRef “CHANGE”)).
```

Output:
[Some [(τ,TICKET [ticket, cash || change, cash] cash → change → CHANGE)];
Some [(τ,cash → ticket → TICKET [ticket, cash || change, cash] CHANGE)]; . . .]

Note that, since this generator also computes transitions in which τ and ✓ may appear as actions, it is necessary to hide them, so that the generated trace does not consider such non-external events. The generator *gen_valid_trace* does so by concatenating only external events to the list that will be returned at the end of the computation.

Once the random trace generator is defined, we need an executable property to evaluate these inputs when applied to another process. In other words, we need a computable function to check whether the trace generated from a process *Imp* is also a trace of a process *Spec*. The idea behind this function is to try to make progress with the given process, one step at a time, consuming the events of the trace in the order in which they appear, while trying to guess non-deterministically when it is necessary to insert a τ or ✓ for the sequence of events to be accepted by the process:

```
Fixpoint check_trace'
  (S : specification)
  (P : proc_body)
  (event_list : trace)
  (fuel : nat) : option bool :=
  match fuel, event_list with
  | _, nil ⇒ Some true
  | O, _ ⇒ None
  | S fuel', e :: es ⇒
    match get_transitions S P with
    | None ⇒ None
    | Some t ⇒
      let available_moves := t in
      let valid_moves := filter (
        fun t ⇒ (is_equal (fst t) (Event e))
          || (is_equal (fst t) Tau)
          || (is_equal (fst t) Tick)
      ) available_moves in
      match valid_moves with
      | nil ⇒ Some false
      | _ ⇒
        let result := map (fun t ⇒
          if is_equal (fst t) (Event e)
          then check_trace' S (snd t) es fuel'
          else check_trace' S (snd t) (e :: es) fuel'
        ) valid_moves in
        if existsb (fun o ⇒
          match o with
          | Some true ⇒ true
          | _ ⇒ false
          end) result
        then Some true
        else if forallb (fun o ⇒
          match o with
          | Some false ⇒ true
          | _ ⇒ false
          end) result
        then Some false
        else None
      end
    end
  end.

Definition check_trace
  (S : specification)
  (proc_id : string)
  (event_list : trace)
  (fuel : nat) : option bool :=
  match fuel, get_proc_body S proc_id with
  | O, _ | _, None ⇒ None
  | S fuel', Some P ⇒ check_trace' S P event_list fuel'
  end.

Definition traceP
  (S : specification)
  (proc_id : string)
  (fuel : nat)
  (t : option semantics_trace.trace) : bool :=
  match t with
  | None ⇒ false
  | Some t' ⇒
    match check_trace S proc_id t' fuel with
    | None ⇒ false
    | Some b ⇒ b
    end
  end.
```

The function *check_trace'* initially computes all possible immediate transitions from one state. Then, it filters, from these transitions, those whose action corresponds either to the current element in the list of events, or the internal event, or the termination event. Then, we try to make progress with each of these valid events, performing a recursive call with each target state obtained from the transitions in the previous step, removing the corresponding event from the list when applicable – that is, except τ and ✓. If any of the recursive call empties the list of events, then this list is indeed a trace of the process. On the other hand, if none of the recursive calls is able to make progress, then we can assume that this list is not a trace of the process.

Provided these definitions, we can then declare the refinement property checker. This checker tests whether every randomly generated trace of an implementation process is also a trace of a specification process:

```
Definition trace_refinement_checker
  (S : specification)
  (Imp Spec : string)
  (trace_max_size : nat)
  (fuel : nat) : Checker :=
    forAll (gen_valid_trace S Imp trace_max_size) (traceP S Spec fuel).
```

In order to see the checker *trace_refinement_checker* in action, consider the following CSP$_{Coq}$ specification:

```
Definition EXAMPLE : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
      [ Channel {{"a", "b", "c"}} ]
      [ "P" ::= "a" --> "b" --> ProcRef "P" ;
        "Q" ::= ("a" --> "b" --> ProcRef "Q") [] ("c" --> STOP) ]
  ).
Defined.
```

Now, we may want to check whether "Q" trace-refines "P", that is, if every trace of "Q" is also a trace of "P". For that, we invoke our checker with the command *QuickChick*, passing 5 as the maximum trace size and the arbitrary integer 1000 as the fuel argument:

```
QuickChick (trace_refinement_checker EXAMPLE "Q" "P" 5 1000).
```

```
Some ["c"]
*** Failed after 3 tests and 0 shrinks. (0 discards)
```

As we are told by the message above, QuickChick performed 3 tests before running into a trace that invalidates the refinement relation, proving that "Q" does not trace-refine "P". We can verify this with the provided counterexample – ["c"] is a trace of process "Q", but it is not a trace of "P". On the other hand, notice that the process "Q" is actually refined by "P" (i.e. "P" trace-refines "Q"). We will not be able to prove that statement with our checker though. Instead, it may only lead us to believe that the relation holds, since QuickChick is not able to provide us with any counterexample after executing 10000 tests.

```
QuickChick (trace_refinement_checker EXAMPLE "P" "Q" 5 1000).
```

```
+++ Passed 10000 tests (0 discards)
```

Therefore, since this method is, ultimately, a testing solution, we are only interested in counterexamples, which actually prove that the refinement relation does not hold between two processes. Passing tests must be considered nothing but a mere indication at best, thus they do not count as proofs for the assertions made. In other words, this refinement-checking approach is sound (if the checked property does not hold, we do not have a trace refinement), but it is not complete (if the checked property holds, we can not guarantee whether it is a trace refinement).

4 Conclusions

In this work, we embedded a subset of the CSP language in the Coq proof assistant, giving rise to the language entitled CSP$_{Coq}$. The abstract syntax was described through inductive types, while the concrete syntax relies on the concept of notations. In addition, the inductive declaration that defines operational semantics in the SOS style was also presented. The concept of LTS was represented both in an inductive and functional approach, supporting a third-party tool that allows a custom graphic visualization of this structure.

Finally, the notion of the trace of a process was declared, along with a tactic macro that automates the proof of this relation. These accomplishments led to the definition of the refinement relation according to the traces model, in addition to the implementation of two generators and one checker for this property, in order to test it using a property-based random testing plugin.

4.1 Related work

Several studies have shown how to embed the theories of many CSP models in theorem proving tools such as Isabelle (PAULSON, 1994), and then prove both the laws of CSP in general and other coherency properties of the definitions of CSP operators over the models. In particular, we want to discuss two implementations that share the same motivation of this work: the CSP-Prover (ISOBE; ROGGENBACH, 2005) and Isabelle/UTP verification toolbox (FOSTER; ZEYDA; WOODCOCK, 2015).

CSP-Prover is an interactive theorem prover dedicated to refinement proofs within CSP based on Isabelle. It focuses on the stable failures model $\mathcal{F}$ as the underlying denotational semantics of CSP, including the CSP traces model $\mathcal{T}$ as a by-product. Consequently, CSP-Prover contains the definitions of CSP syntax and semantics, and semi-automatic proof tactics for verification of refinement relation.

Isabelle/UTP is an implementation of Hoare and He's *Unifying Theories of Programming* framework based in Isabelle/HOL. UTP is a framework for construction of denotational semantic meta-models for a variety of programming languages based on an alphabetized relational calculus. This implementation can be used to formalize semantic building blocks for programming language paradigms, prove algebraic laws of programming, and then use these laws to construct program verification tools.

The Table 6 provides a comparison between the framework we developed, the CSP-Prover, and the Isabelle/UTP, highlighting the main features shared by these implementations:

Table 6 – A feature comparison between CSP$_{Coq}$ and related work.

Feature	CSP$_{Coq}$	CSP-Prover	Isabelle/UTP
CSP Dialect	Partial	Complete	Complete
Operational semantics	Structured	Not Available	Relational
Denotational semantics (model)	Traces	Stable failures/Traces	Not Available
LTS representation	Available	Not available	Not Available
Trace relation tactics	Available	Not Available	Not Available
Refinement relation tactics	Not Available	Available	Available
Random testing (refinement)	Available	Not Available	Not Available

From the table above, we are able to conclude that the main advantages of CSP$_{Coq}$ over others embedding of CSP theory in theorem proving tools mentioned here is its emphasis on the traces model as well as the LTS representation support, enabling the user to check a list of events for being a trace of a process, solving the trace relation proof automatically, and generating a graph of the LTS of a process.

4.2 Future work

The topics listed below describe relevant activities that extend the work we developed:

- Extend the language CSP$_{Coq}$ to include the remaining CSP operators. The theory of CSP, as well as the ASCII version of the language, contemplates more advanced concepts such as parameterized processes and operations like event renaming, interruption, exception, among others. It is necessary to include these concepts in order to leverage the framework developed.
- Prove that for every process P, there is a list of transitions L such that if *ltsR S P L*, then there is natural n such that *compute_ltsR S P n = Some L*. This proof guarantees the completeness of the definition *compute_ltsR*, and that is, every set of transitions that characterize the LTS of a process is computable from this definition.
- Define the tactic macro that automatically proves if the *ltsR* relation holds for a given process. The proof for the ltsR relation can be long, and they usually follow a well-defined pattern. Creating a tactic macro would simplify tasks like checking if a set of transitions is indeed the LTS of a process.

- Formalize the concept of a extended trace, say *extended_traceR*, including the events τ and $\checkmark$; relate *traceR* and *extended_traceR*; create tactic macro that automates the proofs for the new *extended_traceR* relation. A trace, by definition, does not include the events τ and $\checkmark$, which may sometimes simplify the actual behavior of a process. Therefore, an extended definition of trace would expose these suppressed communications.

- Implement verification of forbidden use of recursion in the context of hiding and parallelism operations. These operations, when appearing in a process recursion, introduce the issue of accumulated operators. For instance, the process definition $P = (a \rightarrow P) \setminus \{a\}$, although syntactically correct, generates a new state in the LTS of the process P every time P is unfolded in the body, due to the addition of one more hiding operation to the process body: $(a \rightarrow P) \setminus \{a\} \xrightarrow{a} ((a \rightarrow P) \setminus \{a\}) \setminus \{a\} \xrightarrow{a} (((a \rightarrow P) \setminus \{a\}) \setminus \{a\}) \setminus \{a\} \xrightarrow{a} \ldots$

- Define traces refinement in terms of a bi-simulation. The notion of strong bi-simulation is, in order to be equivalent, two processes must have the same set of events available immediately, with these events leading to processes that are themselves equivalent. Since it is virtually impossible to implement a function that enumerates all traces of a process – so that we could use it to make assertions about a refinement statement – this equivalence over LTS would be the proper way to achieve this goal.

- Compress the generated LTS, removing intermediate τ's. The process unwind and sequential composition operations introduce the communication of the internal event τ as a way to take into account the "effort" of unfolding a process body inside another. Omitting these communications may, not only reduce the LTS size, but also make it more simple to spot equivalent process by looking at their LTS's.

- Prove that for all transition lists $L1$ and $L2$, if *ltsR S P L1* and *ltsR S P L2*, then $L1$ is a permutation of $L2$. Since our definition of LTS is based on a list of transitions instead of a set, this proof would guarantee that there aren't two lists of transitions with different elements that consists of the LTS of the same process. In other words, this would prove the uniqueness of the transition set of a LTS.

Bibliography

CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2000. (ICFP '00), p. 268–279. ISBN 1581132026. Disponível em: <https://doi.org/10.1145/351240.351266>. Citado na página 13.

FOSTER, S.; ZEYDA, F.; WOODCOCK, J. Isabelle/utp: A mechanised theory engineering framework. In: NAUMANN, D. (Ed.). *Unifying Theories of Programming*. Cham: Springer International Publishing, 2015. p. 21–41. ISBN 978-3-319-14806-9. Citado na página 53.

HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, ACM New York, NY, USA, v. 21, n. 8, p. 666–677, 1978. Citado na página 13.

ISOBE, Y.; ROGGENBACH, M. A generic theorem prover of csp refinement. In: HALBWACHS, N.; ZUCK, L. D. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 108–123. ISBN 978-3-540-31980-1. Citado na página 53.

LAMPROPOULOS, L.; PIERCE, B. C. *QuickChick: Property-Based Testing in Coq*. [S.l.]: Electronic textbook, 2020. v. 4. (Software Foundations, v. 4). Version 1.1, <http://softwarefoundations.cis.upenn.edu>. Citado na página 29.

PAULSON, L. C. *Isabelle: A generic theorem prover*. [S.l.]: Springer Science & Business Media, 1994. v. 828. Citado na página 53.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733. Citado na página 10.