Carlos Alberto da Silva Carvalho de Freitas

A Theory for Communicating Sequential Processes in Coq

Recife

2020

Carlos Alberto da Silva Carvalho de Freitas

A Theory for 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

This work would not have been possible without the teaching and guidance of my supervisor, Professor Gustavo Carvalho, to whom I thank especially for understanding the adverse conditions in which this work has been developed; a pandemic without precedence in the contemporary history. I also extend my thanks to the other professors at the Centro de Informática of Universidade Federal de Pernambuco, who value quality education and encourage students to seek knowledge on a daily basis.

I cannot fully express my gratitude to my parents, Píndaro and Rozilda. All I am and everything I have achieved to this day I owe them. The humble yet challenging wish of my late father, when I was only a few years old, for me to grow into a good person and make contributions to the society, has always resonated deeply with me. My mother's relentless encouragement of my education and the pride she bears in her voice when talking about me, inspires me everyday to be the best version of myself.

Finally, I want to give special thanks to the people with whom I shared closely my emotions during the entire graduation course – my classmates and my girlfriend, Nínive. The friendship and companionship of these people not only renewed my strength day after day, but made this journey enjoyable in a way that I could never have foreseen. Knowing that I could count on them has made a huge difference, and it was yet another good reason to get up every morning.

Abstract

Theories of concurrency such as Communicating Sequential Processes (CSP) allow system specifications to be expressed clearly and analysed with precision. CSP specifications are typically analysed with the aid of FDR, a refinement checker (model checker) for CSP. 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 check these properties via proof development. This work provides an initial formalisation of CSP in the Coq proof assistant, and evaluates how this theory compares to other theorem prover-based frameworks for the process algebra CSP. We develop 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 with the support of the QuickChick tool.

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

Resumo

Teorias de concorrência, tais como *Communicating Sequential Processes* (CSP), permitem que especificações de sistemas sejam descritas com clareza e analisadas com precisão. Especificações em CSP são normalmente analisadas em FDR, um verificador de refinamentos (verificador de modelos) para CSP. No entanto, o problema da explosão de estados, comum a verificadores de modelo em geral, é uma limitação real na tentativa de verificar propriedades de um sistema complexo. Uma alternativa é avaliar essas propriedades através do desenvolvimento de provas. Este trabalho fornece uma primeira formalização 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, criou-se 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* a partir do uso da ferramenta QuickChick.

Palavras-chave: CSP. Coq. Álgebra de processos. Assistentes de provas. Refinamento no modelo de *traces*. 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 behaviour. 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 concurrent 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 refinement (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 that is 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 (ISOBE; ROGGENBACH, 2005) and Isabelle/UTP (FOSTER; ZEYDA; WOODCOCK, 2015), 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 (BERTOT; CASTRAN, 2010). Considering that, the main research question of this work is the following: how could we develop a theory for 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 (PARASKEVOPOULOU et al., 2015) tool to search for counterexamples of the traces refinement relation.

1.2 An overview of CSP_{Coq}

To illustrate CSP_{Coq}, our dialect of CSP in Coq, consider the following CSP process, which has been 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 and retrieving coats as appropriate.

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

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

The system comprises six possible events (*coat_on*, *coat_off*, among others), and its behaviour is described by the parallel synchronisation of the attendant's and the costumer's behaviours, requiring that they need to synchronise on the execution of the following events: *coat_off*, *request_coat*, and *coat_on*. Regarding the other events, the attendant and the costumer are free to perform them as they wish.

We can declare such a system in CSP_{Coq} by defining a specification, which consists of 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.
```

As one can see, the main syntax of CSP_{Coq} is close to that of CSP_{M}. It is also important to emphasise that the tactic *solve_spec_ctx_rules* proves the aforementioned contextual rules with no user intervention (automatically). Furthermore, we can execute the following command to compute the process LTS and output the corresponding graph in the dot language:

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

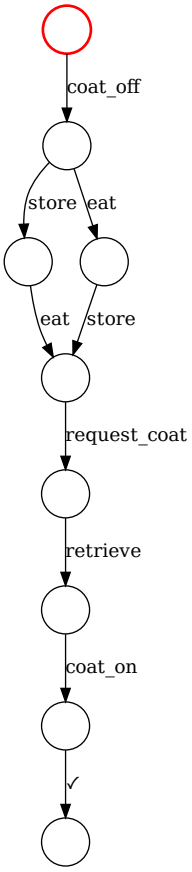
Figure 1 is a visual representation of the graph outputted by this command. The image is generated using the GraphViz software. The red circle denotes the starting node, and the edges are labelled by the performed events.

1.3 Main contributions

The main contribution of this work is an initial formalisation of CSP in Coq (CSP_{Coq}), which takes into account the following aspects:

- Abstract and concrete syntax for a subset of CSP operators.
- Contextual rules for CSP specifications.
- Proof automation for checking contextual rules.
- Operational semantics via the SOS approach.
- Inductive and functional definitions of labelled transition systems.
- Inductive and functional definitions of traces.
- Proof automation for checking whether a list of events is a valid trace.
- Formal definition of traces refinement.
- Traces refinement verification using QuickChick.

Figure 1 – The LTS for the cloakroom example



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 takes 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 below:

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

Chapter 3 Provides an in-depth look at the implementation of CSP_{Coq}, including its abstract and concrete syntax, in addition to the language semantics. Furthermore,

the support for visualising processes as LTSs, 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 related solutions based on other interactive theorem provers. It also addresses possible topics for future work.

2 Background

Before jumping into the details of the implementation of CSP_{Coq}, we first need to understand some aspects of the CSP language itself, such as its syntax and semantics (Section 2.1). Beyond that, it is also important to provide an overview of the Coq proof assistant, covering concepts such as tactics, as well as the embedded Ltac language (Section 2.2). In this chapter, we also present the QuickChick property-based testing tool (Section 3.4.2), which is the Coq implementation of QuickCheck (CLAESSEN; HUGHES, 2000).

2.1 Communicating sequential processes

In 1978, Tony Hoare proposed a theory to help us understand concurrent systems, parallel programming and multiprocessing: *Communicating Sequential Processes* (HOARE, 1978). More than that, it introduces a way to decide whether a concurrent program meets its specification. This theory quickly evolved into what is known today as the CSP specification 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 and 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, 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. An *event* represents the atomic part of the communication. 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 consequences of an action. Events may be external, meaning they appear in the process interface; may indicate termination, represented by the event $\checkmark$; or may be internal, and therefore unknown to the environment, denoted by the event τ.

One of the most basic process one can define is *STOP*. Essentially, this process never interacts with the environment and its only purpose is to denote the end of an execution without success, that is, in the sense of a failure or 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. We can use *SKIP* to illustrate an athlete that has crossed the finish line, or a build for a project that has passed. But differently to *STOP*, *SKIP* allows a continuation of behavior when followed by a sequential composition operator, explained later on in this section.

Provided these two basic processes, *STOP* and *SKIP*, and the knowledge of what the 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 define 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 first engage in. Therefore, the environment has control over the outcome of such a 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 behaviour.

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 (i.e. employees decide), having the clients no control over what deserts they will get. In the first situation, 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 a business rule.

CSP introduces two approaches for describing a parallel execution between processes: the *alphabetised parallelism* and the *generalised parallelism*. Let A be the interface of process P, and B the interface of process Q, an alphabetised 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 at the same moment. Any other event that does not match this criteria can be engaged in by its corresponding process independently.

The semantics is similar for the generalised version of the parallel operator. The only change is its constructor, which takes the synchronisation alphabet alone as the interface argument the processes must agree upon. Let C be the intersection of the previously

defined interfaces A and B, the generalised parallelism between processes P and Q is written as $P \underset{C}{\parallel} Q$.

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 alphabetised parallel operator to specify the combination between two participants as $RUNNER1 \ {}_{\{start,finish1\}}\|_{\{start,finish2\}} \ RUNNER2$, or use the generalised version of the operator instead: $RUNNER1 \underset{\{start\}}{\parallel} RUNNER2$.

Another CSP operator that models concurrent execution of processes is the *interleaving* one. Different from the parallel operators, interleaving represents a combination of processes that do not require any synchronisation 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 *VENDING_MACHINE* as $pay \rightarrow select\_snack \rightarrow return\_change \rightarrow release\_snack \rightarrow SKIP$. Then, the process that specifies both machines operating together can be described as $VENDING\_MACHINE \ ||| \ VENDING\_MACHINE$.

The last two operators we will discuss are *sequential composition* and *event hiding*, in addition to *process referencing*. 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 are not yet 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 $HOMEWORK ; FUN$, where

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

In this example, the process FUN can only be executed after the process *HOMEWORK* has successfully terminated.

We also have the event hiding operator. A system designer may choose to hide events from a process interface to prevent them from being recognised 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

synchronise 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 a way that no student gets to know other test grades besides his or her own. The process *TEACHER* may be modelled as $show\_grade \rightarrow discuss\_questions \rightarrow SKIP$, such that a teacher concerned with the students privacy can be described as $TEACHER \setminus \{show\_grade\}$.

Finally, as illustrated before, we can define new processes by referencing to previously defined ones. For instance, $SYSTEM = HOMEWORK; FUN$ defines the process *SYSTEM* whose behaviour is characterised by the sequential composition of the referenced processes *HOMEWORK* and *FUN*. It is assumed that these two last processes have been defined elsewhere.

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 *behavioural*), and *operational semantics*. Here, we focus on the last one, which is used to construct the LTS of a given process, since providing support for this in Coq is among the objectives of this work.

The operational semantics for the CSP language describes how a valid process 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 to repeat this step until we have covered the whole transition system of the process we are interested in.

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

We start by considering 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 has no 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.

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

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

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

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

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

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

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 group the rules for alphabetised parallelism into two categories: one describing the independent execution of each process, and another defining the synchronised 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, when 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 generalised parallelism are very similar to the ones for the previous operator. 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 alphabetised parallelism 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 synchronise in any event except termination ✓. In other words, this operation is a particular case of the generalised parallelism, where the interface alphabet is empty, thus the event ✓ being the only event that can be performed simultaneously by the components.

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

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

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

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 that the events in the given alphabet are made internal and then renamed to τ. Such a behaviour is captured by the following 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 following rules are for the sequential composition operator. Initially, this composition behaves as the process to the left of the operator until it successfully 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.

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

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

The last operational rule we need to discuss relates to process referencing. Assuming that N refers to the behaviour characterised by P, if a process behaves as N, after performing τ it behaves exactly as P.

$$\frac{}{N \xrightarrow{\tau} P} \quad (N = P)$$

The operational rules presented in this section have been formalised in our Coq characterisation of CSP, as we discuss later.

2.1.2 Traces refinement

A 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 communications between a process and the 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. In this work, we restrict ourselves to finite traces. Nevertheless, the behaviour of a process can be characterised by an infinite set of (finite) traces.

Because this sequence of communications is easily observed by the environment, it is often used to build models of CSP processes. As a matter of fact, there is one sequence 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* $\mathcal{N}$ models.

The notion of refinement is particularly useful for specifying the correctness of a CSP process. If we can establish a relation between components of a system that 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:

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

If we consider P to be a specification that 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 of P.

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 industry and academia (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 version of CSP, usually denoted as CSP_M, not only provides a notation for tools such as the FDR model-checker to be built but also extends the existing theory by using a functional programming language to describe and manipulate concepts like rich data types. Table 1 shows for every CSP process constructor discussed in Section 2.1 the corresponding ASCII representation according to the CSP_M language.

Table 1 – The ASCII representation of CSP

Constructor	Syntax	ASCII form
Stop	$STOP$	STOP
Skip	$SKIP$	SKIP
Event prefix	$e \rightarrow P$	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	$P \underset{A}{\parallel} Q$	P [| A |] Q
Interleave	$P \ ||| \ Q$	P ||| Q
Sequential composition	$P \ ; Q$	P ; Q
Event hiding	$P \setminus A$	P \ A

As one can see, in general, the syntax of the machine-readable version of CSP is quite close to the syntax of the original CSP language.

2.2 The Coq proof assistant

A proof assistant (an interactive theorem prover) is a software for helping to 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. When less human intervention is required, these tools are typically referred as automatic theorem provers. There is a variety of (interactive) theorem provers including Isabelle (NIPKOW; WENZEL; PAULSON, 2002), Agda (NORELL, 2007), Idris (BRADY, 2011), PVS (OWRE et al., 1992) and Coq (BERTOT; CASTRAN, 2010), among others. This work is based on 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 O constructor represents zero. When the S constructor is applied to the representation of a natural number n, the result is the representation of $S\ n$ (or simply $n + 1$), 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, whose body is defined using pattern matching.

```
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 *inductively defined propositions*. Consider the following two rules: *the number* 0 *is even*, and *if* n *is even, then* $S\ (S\ n)$ *is even*. Let us call the first rule ev_0 and then the second ev_SS. Using ev for the name of the evenness property, we can write the following inference rules.

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

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

We can formally define these rules in Coq as follows. 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 the previous use of `Inductive`. Here, 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 in this case each constructor's type must have the form $ev\ n$ for some natural number n.

Yet another style can be used to formalise the notion of evenness. The following definition (*even*) states that a natural number n is even if exists a natural number n' such that $n = 2 \times n'$. It is said that *even* is a *propositional function*, since it is a function that builds a logical proposition (`Prop`) from the given arguments.

```
Definition even (n : nat) : Prop :=
  ∃ (n' : nat), n = 2 × n'.
```

Therefore, in Coq, we can formalise concepts in functional (*evenb*), inductive (*ev*) and logical (*even*) terms. Each style has its pros and cons. Generally speaking, functional definitions are computable and promotes automatic simplification during proofs. Differently, inductive and logical definitions promote the application of rewriting rules during proof development.

2.2.1 Building proofs

As a proof development system, Coq provides interactive proof methods, decision and semi-decision procedures, 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: $\forall\ (n : nat),\ evenb\ (S\ n) = negb\ (evenb\ n)$.

During proof development, one may find useful to make assertions about smaller intermediary steps of a theorem. 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 boolean 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.
```

Table 2 shows how each tactic changes the proof goal until proving the original claim. 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. In the context of this proof, 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 command − is then used to focus on the next generated sub-goal.

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.

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

Once we proved this lemma, it is now available to be used inside other proofs such as the theorem we have stated before. In what follows, we show its proof script.

```
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 step-by-step proof for Theorem *evenb_S* in Table 3. The quantifier $\forall$ *n*:*nat* indicates that we want to prove this claim considering all natural numbers n. The tactic `intros` performs universal instantiation, rewriting the claim considering an arbitrary natural number n, which is declared in the proof context (all definitions above the proof horizontal bar).

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, which can be based on an assumption (hypothesis) from the proof context or on a completely separate proof such as the Lemma *negb_involutive*.

Table 3 – Proof of theorem evenb_S

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

To illustrate other tactics, 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 as follows. Note that here we are referring to the inductive definition of evenness (*ev*).

```
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. In this proof, besides performing universal instantiation, the tactic `intros` moves the implication antecedent to the proof context as a hypothesis. One needs to prove the implication consequent assuming that its antecedent is true; otherwise, the claim would be trivially true since, from contradiction, anything follows (the *Ex falso quodlibet* principle).

As we have discussed before, the tactic `destruct` performs 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 another where $n = S\ (S\ n)$. Once again, we must prove them separately so Coq accepts the theorem.

Table 4 – Proof of Theorem ev_minus2

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

Continuing proof of Theorem ev_minus2 on the next page

Table 4 – Proof of Theorem ev_minus2 continued

Next step in Coq	Proof situation
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

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, since it provides evidence that *ev* 0 is true. In the second branch, after simplification, we have *ev n* as goal, which is already an assumption of ours (introduced to the proof context by the command `intros`). In this situation, the tactic `apply` also concludes the second sub-proof, thus proving the entire statement (theorem *ev_minus2*).

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

2.2.2 The tactics language

Ltac is the tactics language for Coq. It provides the user with a high-level "toolbox" for tactics 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. See the proof script in what follows.

```
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 ¬ (not) in the goal, replacing our initial statement by $In\ 4\ [0\ ;\ 1\ ;\ 2\ ;\ 3] \rightarrow False$. In Coq, the logical negation of P is defined as follows: $P \rightarrow False$. Therefore, if P is true, the obtained proposition is false; if P is false, the obtained proposition is true, as expected.

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 the proof goal.

From this point on, the proof has the following 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. The tactic `contradiction` finishes the proof by searching for a *contradiction* in the proof context. We also note the use of −, ×, + and {} to focus on nested sub-goals.

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 | ]))
  ).
```

The new tactic *solve_not_in* works by first unfolding the function *not* (the propositional function that represents the logical not in Coq), therefore deriving an implication. Then it moves the implication antecedent to the proof context. Finally, it repeats 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. With this new user-defined tactic, the previous example can now be provided as follows.

```
Example elem_not_in_list' : ¬ (In 4 [0 ; 1 ; 2 ; 3]).
Proof. solve_not_in. Qed.
```

As one can see, with no further user intervention besides invoking the appropriate tactic. Furthermore, this proof script would work for lists of arbitrary size. Therefore, we say that *solve_not_in* describes a decision procedure for proving that some element is not a member of a given list.

2.3 QuickChick

QuickChick is a set of tools and techniques for combining randomised property-based testing with formal specifications and proofs in the Coq ecosystem. It is the Coq equivalent of Haskell's QuickCheck tool.

There are four basic elements in property-based random testing: an *executable property*, *generators* of 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 (the first occurence of) 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* could be the following 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 using QuickChick.

QuickChick removeP.

The *QuickChick* command takes an executable property and attempts to falsify it by running it on many randomly generated inputs, resulting in an output as follows.

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 the first occurrence of x will be removed from the list. The last line of the output records that 17 tests were necessary to identify some fault-inducing input and 12 shrinks to reduce it to a minimal counterexample.

3 A theory for CSP in Coq

Chapter 2 provided an overview of the essential concepts for understanding the implementation of CSP_{Coq}, setting the scene for an in-depth explanation of the language developed in this work. That being said, Chapter 3 explains how we developed a theory for communicating sequential processes in the Coq proof assistant.

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 inductively defined propositions. Afterwards, Section 3.3 provides details on both functional and inductive definitions of LTSs, along with further explanation on the GraphViz software integration. Finally, Section 3.4 explains how we define traces refinement as an executable property and also presents randomised testing based on this property using the QuickChick tool.

Our Coq characterisation of CSP is structured into the following files. We have a total of 675 LOC (lines of code) of formal definitions, and 94 LOC concerning the definition of special-purpose tactics.

- **syntax.v**: the abstract and concrete syntax of CSP_{Coq}.
- **semantics_sos.v**: the SOS-style semantics of CSP_{Coq}.
- **lts.v** characterisations of LTSs and integration with GraphViz.
- **semantics_trace.v**: characterisations of traces and integration with QuickChick.

We also have 745 LOC of examples illustrating the formalised concepts. All code is available at our GitHub repository: <https://github.com/casc2/tg-formal-methods>.

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*, the operations event prefix, external choice, internal choice, alphabetised parallelism, generalised parallelism, interleaving, sequential composition, and event hiding, in addition to process referencing. This section introduces the reader to the implementation of both abstract and concrete syntax of the CSP_{Coq}.

3.1.1 Abstract syntax

In order to define 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 have said before, external events are all events that are neither internal (τ) nor indicate termination ($\checkmark$). The type *event_tau_tick* provides not only a constructor for the external events, but also for the especial events τ and $\checkmark$.

```
Definition event := string.
Inductive event_tau_tick :=
  | Event (e : event)
  | Tau
  | Tick.
```

For describing a set of external events, we have the types *channel* and *alphabet*. Apart from considering different names of constructors, they are syntactically equivalent; the difference between these two types being semantic. The type *channel* is used to declare external events that may be communicated in a CSP_{Coq} specification. The constructor provided by the type *alphabet* is applied, for example, to enumerate all external events in the process interface of an alphabetised parallel composition.

```
Inductive alphabet : Type :=
  | Alphabet (events : set event).
Inductive channel : Type :=
  | Channel (events : set event).
```

The types *proc_body* and *proc_def* define, respectively, constructors for the CSP language (SKIP, STOP, process referencing, event prefix, external choice, internal choice, alphabetised parallelism, generalised parallelism, interleaving, sequential composition, and hiding) and are used to declare new processes (process attribution statement). The constructors made available by *proc_body* can be combined in order to describe complex behaviour. The type *proc_def* provides a constructor that makes it possible to identify a process by an identifier, that is, to give it a name.

```
Inductive proc_body : Type :=
  | SKIP
  | STOP
  | ProcRef (name : string)
  | ProcPrefix (event : event) (proc : proc_body)
  | ProcExtChoice (proc1 proc2 : proc_body)
  | ProcIntChoice (proc1 proc2 : proc_body)
  | ProcAlphaParallel (proc1 proc2 : proc_body) (alph1 alph2 : alphabet)
  | ProcGenParallel (proc1 proc2 : proc_body) (alph : alphabet)
```

```
  | ProcInterleave (proc1 proc2 : proc_body)
  | ProcSeqComp (proc1 proc2 : proc_body)
  | ProcHiding (proc: proc_body) (alph : alphabet).
Inductive proc_def : Type :=
  | Proc (name : string) (body : proc_body).
```

The last definition of the CSP abstract syntax is *specification*. A CSP specification can be perceived as a file containing multiple channels of events (*ch_list*) and declarations of processes (*proc_list*). Ultimately, it is a context that holds information such as all the events that can be performed, and all processes 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, which must be proved to hold when creating an instance of the record.

```
Record specification : Type := Build_Spec {
  ch_list : list channel;
  proc_list : list proc_def;
  non_empty_proc_ids : ¬ In EmptyString (map get_proc_id proc_list);
  non_empty_events : ¬ In EmptyString (concat_channels ch_list);
  no_dup_events_proc_ids :
    NoDup ((concat_channels ch_list) ++ (map get_proc_id proc_list));
  no_missing_proc_defs : incl (get_proc_refs proc_list) (map get_proc_id proc_list);
  no_missing_events : incl (get_events proc_list) (concat_channels ch_list)
}.
```

Therefore, when creating a CSP specification in CSP_{Coq}, one needs to prove that the following properties are met. As mentioned in Chapter 1, we have developed a special-purpose tactic (*solve_spec_ctx_rules*) that automates the proof of such properties.

- The empty string is not a valid identifier for the name of a process (property: *non_empty_proc_ids*).
- The empty string is not a valid identifier for the name of an external event (property: *non_empty_events*).
- The name of processes and external events are unique (property: *no_dup_events_proc_ids*).
- There are no references to undefined processes (property: *no_missing_proc_defs*).
- There are no references to undefined external events (property: *no_missing_events*).

To illustrate the CSP_{Coq} abstract syntax, consider the following CSP_M processes:

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

and their representations in CSP_{Coq}:

```
Proc "PRINTER" (ProcPrefix (Event "accept") (ProcPrefix (Event "print") STOP))

Proc "MACHINE" (
  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 can see 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 specification and understanding of CSP_{Coq} processes.

3.1.2 Concrete syntax

In order to define a more appropriate notation for the CSP_{Coq} language, the command `Notation` was used. It 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 lines above, the prefix operator has the higher precedence among all three operators and has right associativity. Differently, the external choice has left associativity while the alphabetised parallel operator does not associate at all, meaning that parentheses are necessary to create a compound expression with multiple parallel operations. Now, we can use these symbols to rewrite the process examples from the previous section in a much more friendly way.

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

Table 5 displays a comparison between the CSP_M operators we have discussed and the CSP_{Coq} language concrete syntax.

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"

Apart from some different symbols (necessary to avoid conflicts with reversed keywords – e.g., "-->" and ";;"), and how to refer to other CSP processes (ProRef "P"), the concrete syntax of CSP_{Coq} is close to that of CSP_M.

3.2 Structured operational semantics

Now that we have a good understanding of how the abstract syntax and also a convenient notation of the CSP_{Coq} language were implemented in Coq, it is time to discuss the language semantics in Coq. All declarations presented in this section are based on the inference rules discussed in Section 2.1.1. More specifically, we will address how those SOS rules were defined in Coq's environment.

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 a property in terms of an inductive declaration in Coq, where each rule was translated into a propositional statement. We will use the same approach to define the semantic rules of CSP_{Coq}.

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,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 our inductive definition, let us revisit the inference rule for the prefix operator. It states that, after performing the event a, the process $a \rightarrow P$ behaves as P.

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

This structure can be translated into Coq as a single constructor in the SOS relation. Note that this definition relates a specification (*specification*), a behaviour (*proc_body*), and an event (*event_tau_tick*) with another behaviour (*proc_body*).

```
Inductive sosR : specification → proc_body → event_tau_tick → proc_body → Prop :=
  | prefix_rule (S : specification) (P : proc_body) (a : event) :
    S # (a --> P) // Event a ==> P
```

Considering the defined notation, 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, it resolves to P.

The constructors *ext_choice_left_rule* and *alpha_parall_joint_rule* illustrate part of the formalisation of the external choice and the alphabetised parallelism operations, respectively. Note that each one of these operations need more than one inference rule in order to fully describe their semantics (see Section 2.1.1). The following definitions consider two of these rules.

The *ext_choice_left_rule* constructor encodes the inference rule that solves the external choice operation for the left operand. As we have explained before, this behaviour is described by the following rule

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

which translates into the following logical proposition

```
| 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')
```

In this statement, we can see that the first clause corresponds to the side condition of the inference rule, which guarantees that the event a is not the internal event τ. The second clause 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 the hypothesis establish the necessary conditions to resolve the external choice operation to the left-hand operand.

Similarly, the constructor *alpha_parall_joint_rule* formalises the inference rule that defines the synchronous communication of an event by two parallel processes (considering alphabetised parallelism). This rule, as we can see from its sequent notation below, has a side condition which guarantees that the event a belongs to the intersection of the processes interfaces. Furthermore, it has two premises, which ensure that this event is able to evolve both operands of the combination, that is, the processes to the left- and to the right-hand side of the parallelism operator can communicate this 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.

```
| 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'
```

Note that this constructor does not consider the ✓ event as a possible synchronisation event, even though it is included in the inference rule (side condition) we saw

above. In Coq, external events and ✓ are defined using different constructors (*Event* and *Tick*), and, thus, cannot be part of the same set. We need to create different rules for each one. Therefore, the constructor *alpha_parall_tick_joint_rule* formalises this joint step when the performed event is ✓. The term *event_dec* provides evidence that the comparison of events is decidable.

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 rule 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, P must be of the form *ProcRef name*, where *ProcRef* is a constructor of type *proc_body*, and *name* is a string that identifies a process. Second, the process referenced by *ProcRef name* must be defined in the specification. The auxiliary function *get_proc_body* searches for this definition inside the specification, while the equality ensures that this search results in "*Some* Q", where Q is a process body. If such a process name is not found in the specification, this auxiliary function yields "*None*".

Although we reflect the original reference rule, it is important to emphasise that our definition in Coq differs from the one implemented by the FDR tool. As explained in Roscoe (2010, p. 203, footnote 7), it does not introduce this τ action to prevent from increasing the size of the state space. However, as a consequence, FDR diverges (stops responding) when analysing definitions such as $P = P$. This does not happen in CSP_{Coq}.

3.3 Labelled transition systems

Now, considering the SOS presented before and implemented in Coq, we discuss another way of representing processes, based on the idea of labelled transition systems (LTSs). This section will cover how the LTS concept is embedded in Coq via functional and inductive definitions, and how CSP_{Coq} provides support to the GraphViz software, which enables a graphical visualisation 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 (a set of triples of the form (P, x, Q), meaning that the state P can perform an action labelled as x and move to state Q). In Coq, we define an 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. Differently, if the event "c" gets communicated instead, then the composition resolves to the process STOP and also terminates. Therefore, we can list three transitions for "P": from state *("a" --> "b" --> STOP) [] ("c" --> STOP)* to state *"b" --> STOP* by performing "a", from state *("a" --> "b" --> STOP) [] ("c" --> STOP)* to state STOP by performing *"c"*, and from state *"b" --> STOP* to state *STOP* by performing "b". To better describe these transitions, we will use 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 LTSs in Coq. Starting from the initial state S_0, compute all the immediate transitions available from this state, that is, all 3-tuples where the target states can be reached in one step from S_0. Mark the initial state as "visited" and mark 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.

Let *compute_ltsR* be the function that implements the idea described in the previous paragraph, the following Coq command computes the set of transitions of "P". We omit here the formal definition of this function, but it is available at our GitHub repository.

```
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 need to provide a natural number to the function *compute_ltsR* (e.g., 1000). In Coq, to ensure termination, every recursive definition (`Fixpoint`) should have

an argument that explicitly decreases in each recursive call. During recursion, since new states may be reached, Coq cannot automatically infer what is the decreasing argument. The natural number ensures this premise (decreasing argument). Therefore, the number of recursive calls is limited by the provided number. However, it is also important to emphasise that, from the yielded result, one can easily assess whether the transition relation has been fully constructed. If the provided argument is not enough for computing all transitions, the function yields *None* instead of *Some* list.

We can prove the correctness of our functional definition for computing LTSs by means of the SOS presented in Section 3.2. If T is the set of transitions in the LTS representation of 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 this in mind, we provide an inductive definition for LTSs, which is based on the underlying SOS. Nevertheless, the prove of correctness of *compute_ltsR* is left as future work.

Given a specification (S), a process identifier (*name*), and a set of transitions (T), the propositional function *ltsR* yields a predicate that is true if T is the set of transitions of P, according to the CSP_{Coq} SOS. First, *name* needs to be a process defined in the specification, and T may not have duplicate entries. If these two conditions hold, we say that T is the correct of transitions if it is related to the definition of P (i.e., its *body*) by the inductively defined proposition *ltsR'*.

```
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 inductively defined proposition (LTS relation) is formalised as follows *ltsR'*. It has two constructors, which capture (inductively) a breadth-first search. Its last two arguments (`set` *proc_body*) control the visited and the to-visit states (represented as process behaviours), respectively. The first constructor (*lts_empty_rule*) states that when no states remain to be visited (*nil*), the corresponding LTS is empty (*nil*).

```
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
```

The second constructor (*lts_inductive_rule*) captures the induction step of our definition. Provided that:

- P is an arbitrary state/process;
- T is a set of transitions;
- T' is the set of transitions emanating from P in T, such that a triple (P, a, P') is in T' if, and only if, according to the SOS, P behaves as P' after performing a;
- T'' is defined as $T - T'$ (i.e., the transitions not emanating from P in T)

we can prove that T is the set of transitions of $P :: tl$ (i.e., the process P follower by an arbitrary list of other states/processes still to be visited – the tail tl), if we prove that T'' is the set of transitions of *to_visit*, which is the list of states to visit updated by including the states reached by T' and excluding the already visited ones (*visited'*), which includes the previously visited states in addition to P.

```
| 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.
```

The successive application of the induction step will progressively relate a set of transitions with all states reached from the initial one. This formal definition can be used to prove that a particular result of *compute_ltsR* is correct, as illustrated in the following example, where *ls* is the list yielded by *compute_ltsR SPEC* "P" 1000.

```
Example lts1_is_valid :
  ltsR SPEC "P" ls.
Proof. ... Qed.
```

During the development of this work, we have used this idea to validate the function *compute_ltsR*; proving that the computed LTSs are correct indeed.

3.3.1 GraphViz integration

GraphViz is an open source graph visualisation software that takes descriptions of graphs in simple textual languages – in our case, the DOT language – and creates diagrams in useful formats, such as SVG and PDF. Moreover, this tool allows the customisation of these diagrams, modifying colors, fonts, hyperlinks, and using custom node shapes. Providing support for this software enables CSP_{Coq} users to generate a graph from the description of an LTS with one command line. That way, the behaviour of a process may be inspected and validated more easily, since a visual representation is available.

In order to build the description of a graph in the DOT language, it is necessary to define a mapping of the constructors of the types *event_tau_tick* and *proc_body* to the *string* type. That way, it is possible to convert each tuple from the LTS into a statement that corresponds to a transition of a graph in the 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. For example, consider the following coercion from the type *event_tau_tick* to the 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 yields a string representation of this 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 possible coercions are stated, we define in Coq the function *generate_dot* 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) is translated to the string "<P> -> <Q> [label = <e>];" by this function, as shown in the definitions below. Note that the function *generate_dot* relies on the auxiliary and recursive function *generate_dot'*.

```
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 these definitions, recall the MACHINE example from Section 3.1.2. In what follows, we present its complete definition 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.
```

Considering the process MACHINE, to generate the graph description in the DOT syntax from the corresponding LTS, we simply run the following command in Coq:

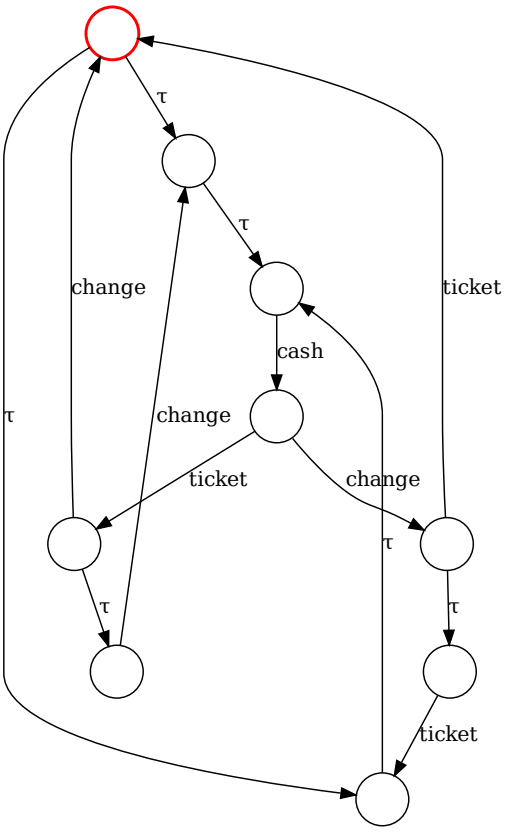
```
Compute generate_dot (compute_ltsR PARKING_PERMIT_MCH "MACHINE" 1000).
```

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

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

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

Figure 2 – Example LTS: The process MACHINE.



In CSP$_{Coq}$, it is also possible to generate such a graph displaying the behaviours (process bodies) associated with each state. Although this decreases the graph readability, it permits the user to know precisely the behaviour represented by each state.

3.4 Traces refinement

As we have seen in Section 2.1.2, a simple but effective way to analyse the behaviour of a process is through the sequence of externally visible events that such a process is capable of performing over time. This communication history is called *trace*. We may perceive a trace as a list of actions that takes from one state to another in an LTS. In Section 3.4.1 we present our characterisation of traces in Coq, and in Section 3.4.2 we explain how we use QuickChick to analyse traces-refinement expressions.

3.4.1 Traces-related concepts

Like the other concepts of CSP we have discussed so far, traces are also formalised in CSP_{Coq}. This concept is defined in terms of a list of events that can be observed by the environment and, thus, τ is not considered here. Although our definition of *trace* allows the presence of τ, the way we build traces prevents its appearance.

```
Definition trace := list event_tau_tick.
```

After defining this new type, we can formalise being a trace as a relationship between a process P and a list of events L, so that it is possible to create logical propositions that involve assessing whether L is a trace of P. This relationship encodes the following inference rules:

- the empty list is a trace of any process;

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

- a non-empty list $L = h :: tl$ is a trace of a process P if P behaves as Q after communicating h, which cannot be τ, and the remaining events of this list is a trace of the process resulting from this communication;

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

- 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.

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

To illustrate this concept, consider once again the process MACHINE, whose LTS is represented by the graph in Figure 2. Examining this figure, one can note that the sequence of events <cash, ticket, change> is a trace of this 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 MACHINE components (processes TICKET and CHANGE). Finally, the left-hand side of the parallel composition may communicate first, performing the event ticket; followed by event change, communicated by the right-hand side of the parallel composition. Remember that a trace consists of visible events only (also considering ✓), therefore the first two communicated events, which both happen to be τ, do not appear in the trace.

Now, we create an inductively defined proposition to formalise the traces notion, encoding the aforementioned rules, which are translated to three constructors of *traceR'*.

The propositional function *traceR* is responsible for retrieving the process body associated with the provided process name.

```
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_tau_tick) (tl : trace) :
    ¬ eq h Tau →
    (S # P // h ==> P') →
    traceR' S P' tl →
    traceR' S P (h::tl)
  | 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 before, in the order they were presented, correspond, respectively, to the constructors *empty_trace_rule, event_trace_rule*, and *tau_trace_rule* of the *traceR'* inductive definition.

This definition of traces as a relation 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 of traces and SOS.

It turns out that, the proof that L is a trace of P grows proportionally to the size of the list (trace). Additionally, these proofs also become quite repetitive. Fortunately, this repetition allows us to define an algorithm (decision procedure) to automatically prove assertions of this kind. Therefore, we have created a tactic macro for this 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 finish the proof. Otherwise, try to make progress by applying each of the other two constructors of *traceR'*, and then repeating this procedure until the proof is finished. For goals that involve expressions in terms of the CSP_{Coq} semantics (SOS), based o pattern matching, try to make progress using the SOS constructors applicable for the current SOS operator, making once again a recursive call at the end of this step.

The definition *solve_trace'*, partially presented in what follows, uses the Ltac language to describe this algorithm as a tactic macro that automatically proves statements of the kind *traceR S "P" L*. Note that the first case deals with the situation when the list is empty (*nil*). The second case tries to make progress by applying the *tau_trace_rule* and *event_trace_rule* constructors. Then, we have many cases considering the application of SOS constructors. In the end, we have some additional cases for proving some specific propositions, involving *set_In* and disjunctions. Finally, the tactic *solve_trace* works unfolding the definition *traceR*, performing simplification and, then, invoking the tactic *solve_trace'*.

```
Ltac solve_trace' :=
  multimatch goal with
  | ⊢ traceR' _ _ nil ⇒ apply empty_trace_rule
  | ⊢ traceR' _ _ _ ⇒
    (eapply tau_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'
  ⋮
  | ⊢ _ ≠ _ ⇒ 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 have 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 have 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 have explained in Section 2.1.2, this definition states 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 Checking refinement with QuickChick

In the context of this work, the QuickChick library was used to random test the refinement property according to the traces model. Our goal is to obtain a simple and automated way to test this property that relates two processes, Imp and Spec, eventually finding counterexamples for the traces refinement relation. In other words, we have developed a checker in QuickChick that tries to find a counterexample of a refinement expression by means of random property-based testing.

In order to create this checker, we first need to define a generator of random inputs, which, in our case, consist of traces for a given process. Thus, we define a random generator that takes a "specification" process and an arbitrary natural number to limit the maximum number of events in the trace. This generator returns a trace for the given process, whose size is limited by the third parameter. Proving that this generator is correct (i.e., all traces generated for a process P are indeed traces of P according to definition *traceR*) remains as a future work.

The function *gen_valid_trace* defines our generator by yielding an instance of the G Monad[1]. This function, as others presented before, retrieves the process body associated with the provided process identifier. Then, it calls the auxiliary function *gen_valid_trace'*.

```
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.
```

The function *gen_valid_trace'* operates as follows. While the third parameter is greater than zero, decide – based on a given frequency – whether the generation should stop.

[1] More information available at <https://softwarefoundations.cis.upenn.edu/qc-current/QC.html>

If 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, if the event is not τ. 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 $((size/(size+1))*100)\%$.

```
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 (Event e :: ts)
              )
            | (Tick, Q) :: _ ⇒
              bind (gen_valid_trace' S Q size') (
                fun ts ⇒ ret (Tick :: ts)
              )
            | (Tau, Q) :: _ ⇒
              bind (gen_valid_trace' S Q size') (
                fun ts ⇒ ret ts
              )
            end
          )
        )
      )
    ]
  end.
```

The command that samples traces according to this generator, as well as part of

the output of this sampling process are exemplified below.

```
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"]; ...]

As one can see, the definition of *gen_valid_trace'* relies on a generator of random transitions from a given process P (*gen_valid_trans*). The latter generator receives a specification and a process body, and yields a list containing exactly one valid random transition from the given state, where the transition is represented by the pair (action, target_state). The generation of all emanating transitions from a given process P is performed by *get_transitions*, which is a function used in our computable definition of LTSs.

```
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 exemplify the usage of *gen_valid_trans*.

```
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 may appear as actions, it is necessary to hide them, so that the generated trace does not consider these internal events. The generator *gen_valid_trace* does so by concatenating only external events and ✓ to the list that will be returned.

Once the random generator of traces is defined, we need an executable property that uses the generated traces to assess a refinement expression. Therefore, we define the function *check_trace'* 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 within 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 for the sequence of events to be accepted by the process.

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 (trace), or the internal event. Then, we try to make progress with each of these valid events, performing a recursive call considering each target state obtained from the transitions in the previous step, but removing the corresponding event from the list when applicable – that is, except τ. If any of the recursive calls reach an empty list (trace), then the provided trace is indeed a trace of the process. Differently, 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.

```
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 negb (is_equal (fst t) (Tau))
          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.
```

The function *check_trace'* is called by *check_trace*, which retrieves the process body associated with the provided process identifier.

```
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.
```

Finally, the function *traceP* provides the given trace to *check_trace* and propagates the boolean yielded by the latter function.

```
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.
```

Provided these definitions, we define the refinement-property checker. This checker tests whether every randomly generated trace of an implementation process is also a trace of a specification process. This definition reflects the formal definition of the trace-refinement relation presented in Section 3.4.1.

```
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).
```

To illustrate the verification of this executable property, 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.
```

If we may want to check whether "Q" trace-refines "P", that is, if every trace of "Q" is also a trace of "P", we invoke our checker with the command *QuickChick*, passing, for instance, 5 as the maximum trace size and the arbitrary integer 1000 as the fuel argument (to limit the number of performed recursions).

```
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 violates the refinement relation, showing evidence that "Q" does not trace-refine "P", since ["c"] is a trace of "Q", but it is not a trace of "P". Differently, note 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 show evidence that the refinement relation does not hold between two processes. Passing tests must be considered nothing but a mere indication of a possibly valid refinement expression. In other words, this refinement-checking approach is sound (if the checked property does not hold, the trace-refinement relation does not hold), but it is not complete (if the checked property holds, we can not guarantee whether the trace-refinement relation holds).

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 using inductive types, while the concrete syntax relies on the concept of notations. In addition, the SOS was declared using inductively defined propositions. The concept of LTSs was represented both in an inductive and functional approach, supporting a third-party tool that allows a custom graphic visualisation of this structure.

Finally, the notion of traces of a process was declared, along with a tactic macro that automates the proof of the is-a-trace 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 (QuickChick).

4.1 Related work

Other studies have shown how to embed the theories of many CSP models in theorem proving tools such as Isabelle (NIPKOW; WENZEL; PAULSON, 2002), and then prove both general laws of CSP and other coherency properties. Particularly, we want to discuss two implementations: the CSP-Prover (ISOBE; ROGGENBACH, 2005) and the Isabelle/UTP verification toolbox (FOSTER; ZEYDA; WOODCOCK, 2015).

CSP-Prover is an interactive theorem prover dedicated to refinement proofs involving CSP processes, which is 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's syntax and semantics, and semi-automatic proof tactics for the verification of refinement relations.

Isabelle/UTP is an implementation of Hoare and He's *Unifying Theories of Programming* (HOARE; JIFENG, 1998) framework based on Isabelle/HOL. UTP is a framework for construction of denotational semantic meta-models for a variety of languages based on an alphabetised relational calculus. Therefore, this implementation in Isabelle can be used to formalise semantic building blocks for different language paradigms and languages (e.g., CSP), prove algebraic laws of programming, and then use these laws to construct program verification tools.

The main distinguishing features of CSP_{Coq} are the support for producing graphical representations of LTSs, in addition to checking traces refinement relations using property-based testing, which may scale better than the traditional model-checking approach.

However, it is important to remember that this testing-based approach is sound, but not complete.

4.2 Future work

This work is a first representation of CSP in Coq and, thus, the topics listed below describe relevant activities that extend these first results and, thus, should be addressed in future developments.

- Extend CSP_{Coq} to include the remaining CSP operators (e.g., indexed forms of operators, compound events, interrupt operator, among others), along with the functional language supported by CSP_M.
- Check for invalid recursions in the presence of hiding and parallelism operations. In order to prevent the creation of LTSs with an infinite number of states, some forms of recursion is not supported when using hiding and parallelism operations.
- Provide optimisation means for compressing the LTS computed by *compute_ltsR*. For instance, the process referencing operation introduces the communication of an internal event τ as a way to take into account the "effort" of unfolding a process body inside another. Omitting these communications may reduce the LTS size.
- Define a tactic similar to *solve_trace* to automate proofs involving the relation *ltsR*. This tactic would automate the proof of whether a computed LTS is indeed a valid LTS for a given process.
- Consider the inductively defined proposition *ltsR* as the specification of the computable definition *compute_ltsR* in order to prove the correctness of the latter.
- Consider the inductively defined proposition *traceR* as the specification of the traces generator *gen_valid_trace* in order to prove the correctness of the latter.
- Define traces refinement in terms of bi-simulation. According to the notion of strong bi-simulation, 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. This would enable proving trace refinement for processes with an infinite number of traces (but a finite number of states).
- Perform empirical analyses comparing CSP_{Coq} with other tools for specifying and analysing CSP processes, such as FDR.
- Perform a more in-depth comparison of related work.

Bibliography

BERTOT, Y.; CASTRAN, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642058809. Citado 2 vezes nas páginas 9 and 22.

BRADY, E. C. IDRIS: Systems Programming Meets Full Dependent Types. In: *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*. New York, NY, USA: Association for Computing Machinery, 2011. (PLPV '11), p. 43–54. ISBN 9781450304870. Disponível em: <https://doi.org/10.1145/1929529.1929536>. Citado na página 22.

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 14.

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 2 vezes nas páginas 9 and 57.

HOARE, C.; JIFENG, H. *Unifying Theories of Programming*. Prentice Hall, 1998. (Prentice Hall series in computer science). ISBN 9780134587615. Disponível em: <https://books.google.com.br/books?id=WpdQAAAAMAAJ>. Citado na página 57.

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 14.

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 2 vezes nas páginas 9 and 57.

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 31.

NIPKOW, T.; WENZEL, M.; PAULSON, L. C. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 3540433767. Citado 2 vezes nas páginas 22 and 57.

NORELL, U. *Towards a practical programming language based on dependent type theory*. Tese (Doutorado) — Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007. Citado na página 22.

OWRE, S. et al. PVS: A prototype verification system. In: KAPUR, D. (Ed.). *11th International Conference on Automated Deduction (CADE)*. Saratoga, NY:

Springer-Verlag, 1992. (Lecture Notes in Artificial Intelligence, v. 607), p. 748–752. Disponível em: <http://www.csl.sri.com/papers/cade92-pvs/>. Citado na página 22.

PARASKEVOPOULOU, Z. et al. Foundational Property-Based Testing. In: URBAN, C.; ZHANG, X. (Ed.). *Interactive Theorem Proving*. Cham: Springer International Publishing, 2015. p. 325–343. ISBN 978-3-319-22102-1. Citado na página 10.

ROSCOE, A. *Understanding Concurrent Systems*. 1st. ed. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 184882257X. Citado na página 40.

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