UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA - CAMPUS DE RECIFE
PROGRAMA DE PÓS-GRADUAÇÃO CIÊNCIA DA COMPUTAÇÃO

VITÓRIA MARIA PENA MENDES

An Updated Theory for Communicating Sequential Processes in Coq

Recife

2024

VITÓRIA MARIA PENA MENDES

An Updated Theory for Communicating Sequential Processes in Coq

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestra em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador: Gustavo Henrique Porto de Carvalho

Recife

2024

Catalogação na fonte
Bibliotecária: Luiza de Oliveira/CRB 1316

M538a Mendes, Vitória Maria Pena.
An updated theory for communicating sequential processes in Coq / Vitória Maria Pena Mendes.– 2024.
81f.: il.

Orientador: Gustavo Henrique Porto de Carvalho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. Centro de Informática, Programa de Pós-graduação em Ciência da Computação, Recife, 2024.
Inclui referências.

1. Communicating sequential processes. 2. CSP. 3. Coq. 4. Assistente de provas. 5. Extensão para o VSCode. I. Carvalho, Gustavo Henrique Porto de. II. Título.

005.1 CDD (23. ed.) UFPE - CIN 2024 – 80

Vitória Maria Pena Mendes

"An Updated Theory for Communicating Sequential Processes in Coq"

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestra em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação

Aprovado em: 30/04/2024.

ACKNOWLEDGEMENTS

I appreciate the invaluable support and guidance that I have received from those around me. Firstly, I would like to thank God for providing me with the strength that I needed throughout this journey. I would also like to thank my parents, Lucicleide and Cleto, for believing in me and giving all the necessary support that enabled me to reach this point. I am also grateful to my fiancé, João Victor, for standing by me during every step of the way. My advisor, Gustavo, thank you for your mentorship and patience throughout this process. Finally, I would like to express my gratitude to my friends and family, who have also motivated me.

ABSTRACT

The ability of a system to perform operations simultaneously is known as concurrency. In concurrent systems, the extensive number of ways in which components can interact with one another significantly elevates the complexity of analysing the behaviour of such systems. CSP (Communicating Sequential Processes) introduces a convenient notation to accurately describe concurrent systems. Over the years, computational tools have been developed to enable the analysis of specifications in CSP, such as: the Failures-Divergence Refinement (FDR) tool, and theories in Isabelle (e.g., CSP-Prover, HOL-CSP). Previously, an initial characterisation of CSP has been developed in Coq: CSP_{Coq}. Here, we significantly extend the possibilities of using CSP to reason about concurrency in Coq. Now, we support compound communications, parametrised processes, and CSP operators that were not considered before. Well-formedness conditions are formalised in Coq and proof automation tactics are provided. The notions of Structured Operational Semantics (SOS), Labelled Transitions Systems (LTS), traces refinement, and deadlock of CSP specifications have also been captured in Coq. Graphical representation of LTSs is enabled via the DOT language and the Graphviz visualisation software. Moreover, we have developed a VSCode extension that automatically converts specifications in CSP_{M} (the machine-readable dialect of CSP) to CSP_{Coq}.

Keywords: communicating sequential processes; CSP; coq; proof assistant; vscode extension.

RESUMO

A habilidade de um sistema realizar operações simultâneas é conhecida como concorrência. Em sistemas concorrentes, o grande número de maneiras nas quais os componentes podem interagir entre si eleva significativamente a complexidade de analisar o comportamento desses sistemas. CSP (*Communicating Sequential Processes*) introduz uma notação conveniente para descrever precisamente sistemas concorrentes. Ao longo dos anos, ferramentas computacionais foram desenvolvidas para permitir a análise de especificações em CSP, tais como: a ferramenta *Failures-Divergence Refinement* (FDR) e teorias em Isabelle (por exemplo, CSP-Prover, HOL-CSP). Anteriormente, uma caracterização inicial de CSP foi desenvolvida em Coq: CSPCoq. Aqui, estendeu-se significativamente as possibilidades de usar CSP para raciocinar sobre concorrência em Coq. Agora, há suporte para comunicações compostas, processos parametrizados e operadores de CSP que não foram considerados previamente. Condições de boa formação são formalizadas em Coq e táticas de automação de prova são fornecidas. As noções de Semântica Operacional Estruturada (SOS), Sistemas de Transição Rotulada (LTS), refinamento no modelo de *traces* e *deadlock* de especificações CSP também foram capturadas em Coq. É ainda possível criar representações gráficas de LTSs através do uso da linguagem DOT e da ferramenta de visualização Graphviz. Por fim, foi desenvolvida uma extensão para o VSCode que converte automaticamente especificações em CSPM (o dialeto em ASCII de CSP) para CSP_{Coq}.

Palavras-chaves: communicating sequential processes; CSP; coq; assistente de provas; extensão para o vscode.

LIST OF FIGURES

LIST OF CODES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The ability of a system to perform operations simultaneously is known as concurrency. In concurrent systems, the extensive number of ways in which components can interact with one another significantly elevates the complexity of analysing the behaviour of such systems. These interactions can lead to phenomena such as deadlock, divergence, nondeterminism, and race conditions, which must be addressed to prevent the occurrence of undesirable behaviour. Generally, tests cannot provide sufficient evidence to guarantee the absence of these undesirable phenomena.

With this challenge in mind, Hoare (1978) proposed CSP (Communicating Sequential Processes), which is a convenient notation to accurately describe concurrent systems. The theories that support CSP make it possible for the analysis and proof of properties of interest in the described systems. For example, it is possible to demonstrate that a system is free from deadlock. Over the years, computational tools have been developed to enable the analysis of specifications in CSP. The Failures-Divergence Refinement tool, FDR (GIBSON-ROBINSON et al., 2014), is tailored for CSP_M (a combination of CSP with a functional language); and the Process Analysis Toolkit, PAT (SUN et al., 2009), is tailored for CSP# (a combination of CSP with an object-oriented language). A comparison between these two dialects of CSP has been carried out by Shi et al. (2012).

Despite the undeniable utility of tools such as FDR and PAT in the automatic analysis of concurrent systems described in CSP, since they base their analyses on the model checking technique, they suffer from a common problem: the state space explosion problem when analysing complex, large-scale systems.

An alternative approach consists of the use of a different analysis technique, based on theorem provers and proof assistants. The tools CSP-Prover (ISOBE; ROGGENBACH, 2005), HOL-CSP (CRISAFULLI; TAHA; WOLFF, 2023) and Isabelle/UTP (FOSTER; ZEYDA; WOODCOCK, 2015) follow this approach and are based on the Isabelle theorem prover. However, the same development has not been carried out in the context of the Coq proof assistant.

In the Coq Package Index[1], within the category "Theory of concurrent systems", the closest related contribution is coq-ccs. It formalises different equivalence notions

[1] Link: <https://coq.inria.fr/coq-package-index>

on labelled transitions systems underlying CCS – the Calculus of Communicating Systems (MILNER, 1980). A closer related contribution is the of work of Freitas (2020), which provides an initial characterisation of CSP in Coq: CSP_{Coq}. There was a little reuse of the first version of CSP_{Coq} in the development of this work, in Section 2.3 we present which parts were reused and the main additions we provide.

Despite the important progress in the direction of formalising CSP in Coq developed by Freitas (FREITAS, 2020), there is still significant room for improvement and expansion of the theory, particularly, by addressing syntactic and semantic aspects that were left behind by this first contribution. With this in mind, the research question addressed in this work is the following: how can we further extend the CSP_{Coq} theory to support a more comprehensive syntax and semantics?

1.1 GOALS

The main goal of this work is to extend the existing theory of CSP in Coq, considering both new syntactic and semantic elements. This unfolds into subgoals as follows:

- Goal 1 (G1): Extend the CSP_{Coq} syntax to consider a wider range of CSP operators.
- Goal 2 (G2): Update the structured operational semantics of CSP_{Coq} accordingly.
- Goal 3 (G3): Update the Graphviz visualisation of labelled transition systems.
- Goal 4 (G4): Update the definition of traces refinement accordingly.
- Goal 5 (G5): Formalise the notion of deadlock freedom.
- Goal 6 (G6): Implement a translator from CSP_{M} to CSP_{Coq}.
- Goal 7 (G7): Illustrate the possibilities of CSP_{Coq} via classical examples.

1.2 SOLUTION OVERVIEW

This section introduces a small example to illustrate the solution made available by this work. To begin, we have adapted the CSP_{M} example proposed by Schneider (1999) that concerns a parking permit machine. It is a simple system that allows users to insert cash and receive a ticket or change. The system is represented by Code 1.1.

Code 1.1 – Specification of the parking permit machine.

```
channel cash, ticket, change

TICKET = cash -> ticket -> TICKET
CHANGE = cash -> change -> CHANGE

MACHINE = TICKET [{cash, ticket} || {cash, change}] CHANGE
```

Source: Current Author

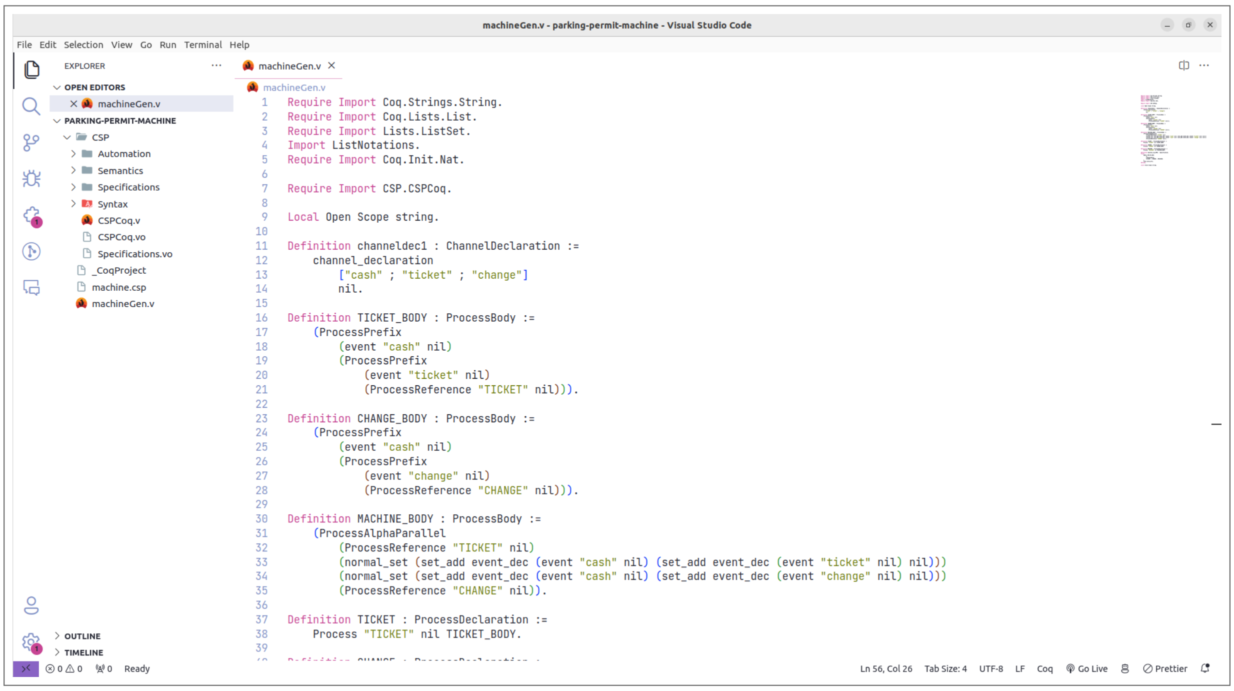
There are three events in the specification: *cash*, *ticket*, and *change*. The *TICKET* process receives *cash*, issues a *ticket* to the user, and then resets to its initial state. The *CHANGE* process works in the same way, but instead, it receives *cash*, and provides *change* to the user. The *MACHINE* process is defined in terms of the parallel composition of the *TICKET* and *CHANGE* processes. The alphabetised parallel operator is used to define the events each side of the parallel composition is allowed to perform: *TICKET* can perform *cash* and *ticket*, whereas *CHANGE* can perform *cash* and *change*; both processes synchronise on the occurrence of *cash*, which is the event shared by both of them.

To facilitate the representation of CSP specifications in CSP_{Coq}, we developed an automatic translator from both representations. This translator is provided as an extension to Visual Studio Code (VSCode). Figure 1 shows the CSP_{Coq} code produced for this example. The extension also copies to the directory the definitions that allow for reasoning about CSP specifications.

In Code 1.2, we highlight a fragment of the generated CSP_{Coq} code. The process *Machine* is defined referring to the *MACHINE_BODY* definition; *nil* indicates that the process *Machine* takes no arguments (i.e., it receives an empty list of arguments). The definition *MACHINE_BODY* comprises the alphabetised parallelism of processes references to *TICKET* and *CHANGE*. The alphabets (set of events) that drive the parallel composition are defined as $\{cash, ticket\}$ for *TICKET*, and $\{cash, change\}$ for *CHANGE*. Here, the alphabets are defined in terms of a normal set (*normal_set*) of events; opposed to production sets, as explained later. Additionally, we also provide *nil* to the definition of the events, since we are not referring to compound events (i.e., those that involve communications).

When creating a CSP specification in Coq, we ensure by proof that a great number of well-formedness conditions are satisfied. A number of custom automation tactics are invoked to automatically discharge the associated proofs.

Figure 1 – Specification of the parking permit machine in $\mathrm{CSP}_{\mathrm{Coq}}$



```
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import Lists.ListSet.
Import ListNotations.
Require Import Coq.Init.Nat.

Require Import CSP.CSPCoq.

Local Open Scope string.

Definition channeldec1 : ChannelDeclaration :=
    channel_declaration
        ["cash" ; "ticket" ; "change"]
        nil.

Definition TICKET_BODY : ProcessBody :=
    (ProcessPrefix
        (event "cash" nil)
        (ProcessPrefix
            (event "ticket" nil)
            (ProcessReference "TICKET" nil))).

Definition CHANGE_BODY : ProcessBody :=
    (ProcessPrefix
        (event "cash" nil)
        (ProcessPrefix
            (event "change" nil)
            (ProcessReference "CHANGE" nil))).

Definition MACHINE_BODY : ProcessBody :=
    (ProcessAlphaParallel
        (ProcessReference "TICKET" nil)
        (normal_set (set_add event_dec (event "cash" nil) (set_add event_dec (event "ticket" nil) nil)))
        (normal_set (set_add event_dec (event "cash" nil) (set_add event_dec (event "change" nil) nil)))
        (ProcessReference "CHANGE" nil)).

Definition TICKET : ProcessDeclaration :=
    Process "TICKET" nil TICKET_BODY.

```

Source: Current Author

Code 1.2 – *MACHINE* process in $\mathrm{CSP}_{\mathrm{Coq}}$.

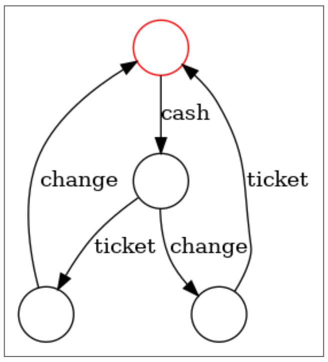
```
Definition MACHINE : ProcessDeclaration :=
  Process "MACHINE" nil MACHINE_BODY.

Definition MACHINE_BODY : ProcessBody :=
  (ProcessAlphaParallel
        (ProcessReference "TICKET" nil)
        (normal_set (... (event "cash" nil) (... (event "ticket" nil) ...)))
        (normal_set (... (event "cash" nil) (... (event "change" nil) ...)))
        (ProcessReference "CHANGE" nil)).
```

Source: Current Author

We support graphical representation of the Labelled Transition Systems (LTS) of CSP processes via the DOT language and the Graphviz visualisation software. The function *generate_dot* (*lts* : *LTS*) : *string* yields a textual (string) representation of a given LTS. To achieve this, we formalise in Coq the Structured Operational Semantics (SOS) of CSP, along with how an LTS is constructed after a given CSP process.

After invoking *generate_dot*, Graphviz can be used to render the corresponding image. Figure 2 shows the graphical representation of the LTS of the parking permit machine; actually, a simplification of the real LTS, since internal events (associated with unfolding process references) are hidden. In Figure 2, the red circle denotes the initial state of the LTS.

Figure 2 – LTS of the parking permit machine.



Source: Current Author

Furthermore, our CSP characterisation in Coq allows for formal reasoning about classical properties of concurrent systems, namely: traces refinement and deadlock freedom.

1.3 MAIN CONTRIBUTIONS

In summary, the main contributions of this work is the following ones.

- A significantly expanded syntax of CSP in Coq (relates to G1);
- Formalisation of well-formedness conditions (relates to G1 and G2);
- Automation tactics for proving well-formedness conditions (relates to G1 and G2);
- Formalisation of the structured operational semantics of CSP (relates to G2);
- Formalisation of labelled transition systems of CSP processes (relates to G3);
- Graphical representation of LTSs via Graphviz (relates to G3);
- Formalisation of traces and traces refinement (relates to G4);
- Formalisation of deadlock freedom (relates to G5);
- Translation from CSP_{M} to CSP_{Coq} using ANTLR (relates to G6);
- VSCode extension for translating from CSP_{M} to CSP_{Coq} (relates to G6);
- Illustration of CSP_{Coq} considering the dining philosophers example (relates to G7).

1.4 DOCUMENT STRUCTURE

This document is arranged as follows. In Chapter 2, the fundamental background information necessary to understand the work is presented, including the theoretical foundations of CSP, the Coq proof assistant, the preliminary CSP_{Coq} theory by Freitas (2020) and the ANTLR framework. In Chapter 3, we present our updated theory for CSP in Coq. In Chapter 4, we describe the tool support for CSP_{Coq}, including the translator from CSP to CSP_{Coq}, and the VSCode extension. Lastly, in Chapter 5, we bring up our final thoughts taking into consideration related and future work.

2 BACKGROUND

This chapter overviews the fundamental concepts required to understand this work. Section 2.1 presents some fundamental concepts related to the theory of Communicating Sequential Processes and its machine-readable language ($\mathrm{CSP_M}$). In Section 2.2, we discuss the Coq proof assistant and its role in formalising and verifying theorems. Once we have covered these aspects, we progress to Section 2.3, where we explore the preliminary theory of CSP in Coq proposed previously by Freitas (2020). Lastly, in Section 2.4, we discuss the ANTLR tool, which is used in this work for implementing the translator from CSP to $\mathrm{CSP_{Coq}}$.

2.1 COMMUNICATING SEQUENTIAL PROCESSES

Communicating Sequential Processes (HOARE, 1978) was introduced by Tony Hoare as a theory to describe concurrent systems. Considering the time when computer systems started to become more complex, his theory provided theoretical means for reasoning about concurrent execution, particularly, when dealing with deadlock and nondeterminism.

Hoare's initial work describes how processes interact and exchange messages across channels. In his foundational work, CSP emerged as a framework for modelling the interaction between different processes within a system. Understanding elements such as processes and communication events is indispensable before moving on.

Communication events are the basic building blocks of CSP. They represent atomic actions one or more processes can perform, such as sending or receiving a message through a channel. In addition to normal events, CSP has two special ones. The first one is *tau* (τ). It represents an internal, an invisible action within a process, that is not observable from the external environment. Additionally, there is also the *tick* ($\checkmark$) event that indicate the successful termination of a process. The $\checkmark$ acts as a marker of this completion, offering a clear endpoint to a sequence of communications or actions within a system.

Processes can execute events independently or with other processes when performing a coordinated action. The two fundamental processes of CSP are $SKIP$ and $STOP$. $SKIP$ represents a process that ends successfully, only performing $\checkmark$. $STOP$ is a process that cannot engage in any event and represents a deadlock situation.

The basic processes mentioned above can be combined using a set of operators in order to describe complex behaviours. We talk about some of them later on as an example. But first, let us understand how the theory of CSP was extended into a set of practical tools.

In the 1990s, under the supervision of Roscoe, Scattergood (1998) made an important contribution by introducing CSP$_{\mathrm{M}}$, a machine-readable dialect of CSP. This version featured a functional programming language that aimed to facilitate the use of tools for automated analysis using CSP. One of the central features of CSP$_{\mathrm{M}}$ is its support by FDR (Failures-Divergences Refinement), a refinement checker that facilitates the practical implementation of formal verification techniques and identifies potential issues such as deadlock, livelock, and other undesirable behaviours.

To demonstrate the utilisation of this dialect, we present a CSP$_{\mathrm{M}}$ specification of the dining philosophers' problem, which will serve as the running example for this work. The dilemma of the dining philosophers is a classic illustration in the study of concurrent systems, it serves as a metaphor for illustrating the issues of resource allocation and process synchronisation that can lead to deadlock conditions. In Code 2.1, we present the CSP$_{\mathrm{M}}$ specification of this classical problem.

Code 2.1 – Specification of the dining philosophers in CSP$_{\mathrm{M}}$.

```
N = 5

PHILNAMES = {0..N-1}
FORKNAMES = {0..N-1}

channel thinks, sits, eats, getsup : PHILNAMES
channel picks, putsdown : PHILNAMES.FORKNAMES

PHIL(i) = thinks.i -> sits!i -> picks!i!i -> picks!i!((i+1)%N) ->
eats!i -> putsdown!i!((i+1)%N) -> putsdown!i!i -> getsup!i -> PHIL(i)

FORK(i) = picks!i!i -> putsdown!i!i -> FORK(i)
[]  picks!((i-1)%N)!i -> putsdown!((i-1)%N)!i -> FORK(i)

PHILS = ||| i : PHILNAMES @ PHIL(i)
FORKS = ||| i : FORKNAMES @ FORK(i)

SYSTEM = PHILS [| {|picks, putsdown|} |] FORKS
```

Source: (ROSCOE, 2010)

N is a constant that states the number of philosophers around the table, five in the example. *PHILNAMES* and *FORKNAMES* define sets of identifiers for the philosophers and forks ranging from 0 to $N-1$, which turns to be 4 in this situation. Subsequently, there are channels declarations, in which *thinks*, *sits*, *eats*, and *getsup* represent the states of a philosopher by their identifiers; *picks* and *putsdown* uses a combination of philosopher and fork identifiers to indicate which philosopher is picking up or putting down which fork. It is important to note that the aforementioned channels communicate values and, thus, are called examples of compound events.

$PHIL(i)$ defines the behaviour of the i-th philosopher. The process describes an unending cycle where a philosopher thinks, sits down, picks up the left and then the right fork, eats, puts down the right and then the left fork, and gets up to repeat the cycle. The `->` operator, called *prefix*, ensures that these actions will happen one after another in the order that was specified. At the end, the definition of $PHIL(i)$ refers to itself, implying recursion.

The $FORK(i)$ process allows the i-th fork to be picked up by its right-hand or left-hand philosopher and then to be put down, repeating the processes. Since the fork is a shared resource, it is going to be picked up by the philosopher who attempts the *picks* action first. The `[]` operator, called *external choice*, models a decision that has to be made between two possible actions, meaning that after being picked up, the fork must be put down by the same philosopher before it can be picked up again. Additionally, this choice is external to the process $FORK$, reflecting an environmental choice, unlike the *internal choice* `|~|`, which represents a different behaviour. In the case of *internal choice*, the decision of what happens next is made by the process itself, rather than being influenced by the environment's inputs.

PHILS introduces the use of an indexed, also called replicated, version of the *interleaving operator*. It allows the parallel execution of multiple instances of a process, each of which is differentiated by an index. The operator $|||$ indicates that the processes are run in parallel without requiring synchronisation among them. The expression `i : PHILNAMES` denotes that the interleaving is indexed over the set *PHILNAMES*: for each i in this set, a process instance will be created. Lastly, `@ : PHIL(i)` specifies which process template is used for each instance and how the identifier i is passed to this template. *FORKS* will operate similarly, considering the *FORKNAMES* set and the process behaviour of a *FORK*.

Ultimately, *SYSTEM* uses the *synchronous parallel* operator `[| |]` to combine the *PHILS* and *FORKS* processes. They operate independently, except for the actions within the synchronisation set that is `{| picks, putsdown|}`. The symbol `{| |}` denotes a production set, which comprises all values that can be communicated over the provided channels. This synchronisation is required to ensure that the philosophers can only pick up or put down forks when the corresponding forks are available.

If every philosopher picks up his left fork at the same time, no philosopher will be able to pick up his right fork because it will have already been taken by another philosopher. This results in a circular wait condition, which is one of the conditions for a deadlock. Tools like FDR can verify properties like deadlock freedom of CSP models. As a model checker, it computes all possible states of the model while searching for the property of interest.

While handling large or complex systems, due to the exhaustive state space exploration, this search approach typically leads to the so called state explosion problem; the computation does not render to be feasible in a reasonable amount of time. In the following section, we introduce the utilisation of proof assistants such as Coq as an alternative that can provide a more scalable approach, at the expense of semi-automated proof effort, to verifying concurrent systems. By considering such tools, for instance, verification can be performed by relying on structural induction principles.

Finally, it is important to note that this brief introduction to CSP does not cover all available operators of the language; for instance, the dining philosophers example does not use the *hiding* and *interrupt* operators. Therefore, CSP_{Coq} considers a more comprehensive set of operators than the ones covered by this example.

Beyond that, it is also essential to understand the concepts of traces and refinement. Traces are sequences of events a process engages in during its interactions with the environment, recorded chronologically. These traces, which can be either finite or infinite, offer a means to gather information regarding a process's behaviour. Moreover, refinement ensures that a more detailed specification meets the conditions of a more abstract one. CSP uses the traces and refinement model to verify that a refined process conforms to the specified behaviours and constraints, allowing the substitution of a less desirable component with a superior one without compromising the system's characteristics. In this model, a process is said to refine another one if, and only if, the set of traces of the former is a subset of the set of traces of the latter.

2.2 THE COQ PROOF ASSISTANT

Proof assistants, also known as interactive theorem provers, are tools for writing proofs and verifying the correctness of software systems. At the heart of proof assistants is the capacity to formalise mathematical theorems and their proofs by utilizing formal languages. The theory of Coq (BERTOT et al., 2010) was developed in France in the 1980s by a group of researchers led by Thierry Coquand and Gérard Huet. It is based on the Calculus of Inductive Constructions (COQUAND; HUET, 1988), a powerful type theory for constructive logic. Gallina is the native functional programming language of Coq. It provides a solid foundation for reasoning about correctness. Its type system is important for making sure that specifications are well-formed and free of errors.

Coq's environment contains an extensive library that provides predefined theorems, lemmas, and proof tactics that enable development upon existing formalisations rather than starting from scratch. To facilitate proof construction, Coq employs tactics, which are small programs designed to guide the development of proofs by breaking them down into smaller, more manageable steps. Some tactics help decompose complex goals into straightforward subgoals, while others facilitate applying established logical rules.

Code 2.2 shows how enumerated types, such as *bool*, can be created in Gallina. Besides the name of the type, one defines a finite number of constructors. In this case, we have just two constructors: *true* and *false* that take no arguments. This type definition covers only syntactic aspects. The associated semantics comes from how other definitions manipulate elements of the type.

Code 2.2 – Definition of the type *bool*.

```
Inductive bool : Type :=
  | true
  | false.
```

Source: (PIERCE et al., 2018)

In Code 2.3, we have the function *negb*. Given a boolean *b* as argument, it yields another boolean with the opposite semantic value. This function is defined by pattern-matching on *b*. If *b* is a value created from the application of the *true* constructor, the function yields a value created from the application of the *false* constructor. Conversely, if *b* is *false*, the function yields *true*.

Code 2.3 – Definition of the function *negb*.

```
Definition negb (b:bool) : bool :=
  match b with
    | true ⇒ false
    | false ⇒ true
  end.
```

Source: (**PIERCE et al., 2018**)

The function defined in Code 2.3 is non-recursive and, thus, uses the keyword `Definition`. Recursive functions are defined using the keyword `Fixpoint`. When defining recursive functions, the automatic mechanism embedded in Coq needs to be able to prove that the function always terminate. Generally speaking, if this cannot be proved, the recursive function cannot be defined. There are some strategies to address this situation. We cover some of them in the next chapter on demand.

To illustrate constructors that have parameters, let us examine the following definition of natural numbers: *nat* (see Code 2.4). This type is not considered enumerated, since the set of natural numbers is infinite. Nevertheless, its definition covers a finite number of constructors. In this non-binary representation of natural number, we have two constructors. The underlying concept is that every natural number is either zero or the successor of a preceding natural number. Therefore, it introduces two constructors, O and S, the capital letter O represents zero and the S stands for successor. Note that the second constructor has a parameter of type *nat*.

Code 2.4 – Inductive definition of type *nat*.

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

Source: (**PIERCE et al., 2018**)

So, if the number zero is represented by O, $S\ O$ represents 1, $S\ (S\ O)$ represents 2, and so on. Any natural number can be constructed using just these two constructors thanks to inductive principle associated with this recursive definition. It embodies the essence of mathematical induction, where you have a base case O and an inductive step, which is: if n is a natural number, then $S\ n$ is also a natural number.

After a brief introduction to data types and functions, let us dive into the most common tactics, such as *intros*, *simpl* and *reflexivity* through a proof example. As mentioned before, under the hood, a tactic is a function designed to manipulate proof objects and, thus, guide the development of proofs. Considering the *nat* definition presented before, it is possible to prove that adding 0 to any number yields the same number. In other words, 0 is the neutral element for adding numbers. Code 2.5 shows the formalisation and proof of this statement in Coq. The proof tactics are provided between the commands `Proof` and `Qed`; the former initiates the proof environment, whereas the latter concludes the proof when no proof obligations are left.

Code 2.5 – Theorem *plus_O_n*.

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity.
Qed.
```

Source: (**PIERCE et al., 2018**)

Table 1 shows the step-by-step evolution of the proof object by the application of the proof tactics. The proof object encompasses the proof hypotheses (context), which are listed at the top, and the proof goals, which are listed under a horizontal bar. In this proof, firstly, universal instantiation is performed by the *intros* tactic. After executing *intros n*, the variable n is moved from the goal area to the hypothesis area, and the goal is updated. Now, one needs to prove that $0 + n = n$ holds for an arbitrary n of type *nat*. The *simpl* tactic simplifies the current goal by reducing expressions. From the definition of $+$, it follows that $0 + n$ yields n. At this point, the proof goal is $n = n$. The proof is completed by using the *reflexivity* tactic, since both sides of the equality encompass the same term. Finally, *Qed* finishes the proof environment provided that there are no more goals to prove.

Let us bring another proof to illustrate the destruct tactic, which performs case analysis. For inductively defined data types, such as booleans, destruct can break down the proof into separate cases (subgoals), considering the different possible ways of constructing booleans. Code 2.6 shows the proof that *negb* is involutive.

Code 2.6 – Theorem *negb_involutive*.

```
Theorem negb_involutive : ∀ b : bool, negb (negb b) = b.
Proof.
  intros b. destruct b eqn:E.
  - reflexivity.
  - reflexivity.
Qed.
```

Source: (**PIERCE et al., 2018**)

Table 1 – Proof steps of the theorem `plus_O_n`.

Command	**Proof situation**
`Proof.`	1 subgoal ——— $\forall n : \text{nat}, 0 + n = n$
`intros n.`	1 subgoal $n : \text{nat}$ ——— $0 + n = n$
`simpl.`	1 subgoal $n : \text{nat}$ ——— $n = n$
`reflexivity.`	No more subgoals.
`Qed.`	

Source: Current Author

Table 2 shows how the proof object evolves along the proof script, and each line shows the proof situation after finishing the corresponding commands. After entering on proof mode, we perform universal instantiation concerning the variable b. Then, the tactic *destruct* performs case analysis on the possible values of b. As the *bool* has two constructors, two subgoals are created: one when b is *true*, another when b is *false*. The original proof goal is updated accordingly. In the context, a hypothesis E is created to store the assumption associated with each subgoal. In Table 2 (fourth row), note that only $E : b = \text{true}$ is shown, since this is the information available at this moment (i.e., just after performing the *destruct* tactic). The hypothesis $E : b = \text{false}$ is made available when the second proof goal is focused. The command `-` focuses on the first subgoal, which is finished by applying *reflexivity*. The same is done for the second subgoal, which terminates the proof.

Coq encompasses a big number of built-in tactics: almost 400 tactics are provided by default[1]. Additionally, new custom tactics can be defined by the user. This is done by using a specific language called Ltac.

Regarding Integrated Development Environments (IDE), CoqIDE provides a user-friendly interface for specification and proof development. There is also an extension to VSCode[2]. It is possible to use Coq in the browser with an online IDE called jsCoq (ARIAS;

[1] Tactic index: <https://coq.inria.fr/doc/V8.19.0/refman/coq-tacindex.html>
[2] Link: <https://github.com/coq-community/vscoq>

Table 2 – Proof steps of the theorem `negb_involutive`.

Command	**Proof situation**
`Proof.`	1 subgoal ______________________ $\forall\, b : \text{bool}, \text{negb}\ (\text{negb}\ b) = b$
`intros b.`	1 subgoal $b : \text{bool}$ ______________________ $\text{negb}\ (\text{negb}\ b) = b$
`destruct b eqn:E.`	2 subgoals $b : \text{bool}$ $E : b = \text{true}$ __________________ (1/2) $\text{negb}\ (\text{negb true}) = \text{true}$ __________________ (2/2) $\text{negb}\ (\text{negb false}) = \text{false}$
`- reflexivity.`	This subproof is complete, but there are some unfocused goals: __________________ (1/1) $\text{negb}\ (\text{negb false}) = \text{false}$
`- reflexivity.`	No more subgoals.
`Qed.`	

Source: Current Author

PIN; JOUVELOT, 2017). The Coq community has also contributed to a rich ecosystem of plugins and extensions that enhance Coq's functionality.

2.3 FIRST VERSION OF CSP$_{\text{Coq}}$

A preliminary theory for CSP in Coq was proposed by Freitas (2020). It provides an abstract and concrete syntax based on CSP$_{\text{M}}$ for a subset of CSP operators. The abstract syntax describes types such as events, channels, alphabets, and processes that are wrapped up into a specification that also takes into account well-formedness conditions (WFC). The proof of WFCs is automated by custom tactics. To get closer to CSP$_{\text{M}}$, the authors used the `Notation` feature provided by Coq to define a custom concrete sytax. In Table 3, one can see a comparison made between the syntax of CSP$_{\text{M}}$ and this first version of CSP$_{\text{Coq}}$.

As it can be seen, the syntax of CSP$_{\text{M}}$ and the previous version of CSP$_{\text{Coq}}$ are quite close. However, due to reserved symbols in the Coq language, it was necessary to adapt

Table 3 – The syntax of CSP_M and the first version of CSP_Coq.

Operator	$\textbf{CSP}_\textbf{M}$	$\textbf{CSP}_\textbf{Coq}$
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 reference	`P`	`ProcRef "P"`

Source: (FREITAS, 2020)

some notations to ensure that the Coq compiler is capable of parsing CSP_Coq code. In Code 2.7, there are two examples illustrating the concrete syntax of the first version of CSP_Coq. The first one defines the *PRINTER* process that performs the events *accepts*, *print*, and then deadlocks. In the second example, we have a *MACHINE* process, similar to the one explained in Section 1.2 (see Code 1.1): a process that combines the behaviour of *TICKET* and *CHANGE* via an alphabetised parallelism. The syntactic differences with respect to CSP_M are evident.

Code 2.7 – Concrete syntax of the first version of CSP_Coq.

```
“PRINTER” ::= “accept” --> “print” --> STOP

“MACHINE” ::= ProcRef “TICKET”
              [[ {{“cash”, “ticket”}} \\ {{“cash”, “change”}} ]]
              ProcRef “CHANGE”
```

Source: (FREITAS, 2020)

In this work, we significantly extend the previous CSP_Coq theory. We support a much richer syntax, considering constants, arithmetic and boolean expressions, compound events, parametrised processes, and more CSP operators: conditional guards, if-then-else conditionals, interruption, and replicated operators such as replicated external choice, internal choice, alphabetised parallel, generalised parallel, interleave and sequential composition. This brings new challenges to the formalisation of the CSP theory in Coq, since it is necessary to have static and dynamic typing systems, in addition to reason about the process behaviour evolution in light of the associated context; now, we may have local variables introduced by process parameters, channel communication, and replicated operators. These changes impacts the whole theory, for instance: the number of well-formedness conditions, the definition of the structured operational semantics, the notion of labelled transitions systems, and the Graphviz integration. In the upcoming chapter, we will discuss in details the updated theory of CSP_{Coq}.

Table 4 provides an overview of the amount of the reused code from the first version of CSP_{Coq}. The table lists the main concepts of CSP_{Coq}, and their reuse level. It is not straightforward to determine an accurate number of reused LOCs. There is almost no direct correspondence between the files structure now and then; additionally, a *diff* command would not yield a reliable output, since that, even when there was a high level of reuse, small changes were required. Therefore, we summarise the reuse level using a categorical mensuration. A low level of reuse implies that the file was utilised as a reference, but the code was rewritten from scratch. A high level of reuse means that the file was adapted to the updated version of CSP_{Coq}. None indicates that there was no reuse at all.

Table 4 – Level of reused code from the first version of CSP_{Coq}

File	**Level of reuse**
Constants syntax	None
Expressions syntax	None
Channels syntax	Low
Processes syntax	Low
Well-formedness conditions	None
SOS	Low
LTS	High
Traces	High

Source: Current Author

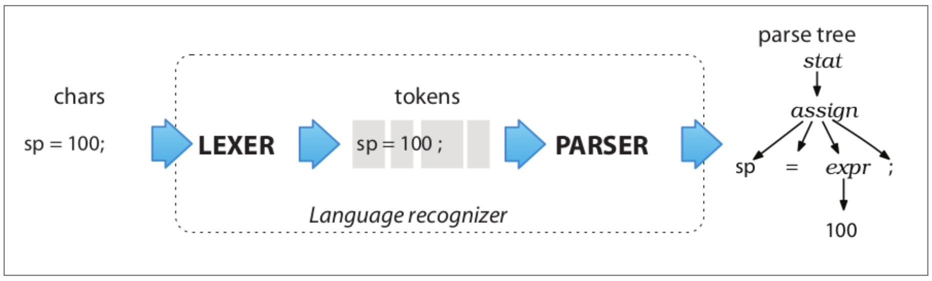
Now, the syntax of CSP_{Coq} is so elaborate that, due to the limitations of `Notation`, it is not straightforward to define a suitable concrete syntax. The differences with respect to the syntax of CSP_{M} would become even greater. This would impose an adoption barrier, since users of CSP_{Coq} would need to become familiar with this new syntax. Therefore, we opted to provide a translator from CSP_{M} to CSP_{Coq} implemented using ANTLR.

2.4 ANTLR

ANother Tool for Language Recognition (ANTLR) (PARR, 2013) is a powerful parser generator used in the development of compilers, interpreters, and frameworks for processing complex languages. ANTLR allows developers to define grammars using an intuitive syntax, then, with the grammatical descriptions, ANTLR generates the source code for a parser in a target language, such as Java, Python, and JavaScript.

In Figure 3, a diagram illustrates the typical data flow of a language recogniser. First, a lexer processes the input text and, after iterating over each character, identifies tokens. Then, the parser analyses whether the tokens are provided in an order accepted by the grammar of the source language. If this is the case, it yields the corresponding parse tree. As said before, the code for the lexer and the parser is automatically generated from the given grammars.

Figure 3 – Typical language recogniser data flow.



Source: PARR (2013)

ANTLR's set of features goes beyond just parsing, it also enables the execution of custom actions by traversing parse trees, such as building abstract syntax trees (ASTs) or even performing semantics. The two methods for traversing and processing parse trees generated from an ANTLR grammar are visitors and listeners. Both are based on the design patterns of same name, and allow the developers to implement custom behaviour when certain nodes in the parse tree are reached. Listeners are suited for a more passive way of processing the tree since they are automatically called by the ANTLR parser. Visitors, however, offer a more complex and customisable way of implementation, requiring a higher level of manipulation since they may not strictly follow the tree structure defined by the grammar.

In Code 2.8, we illustrate how to define an ANTLR4 grammar using the EBNF notation. The example defines a grammar called `Expr`, which provides a simple language for arithmetic expressions. The first rule (prog) serves as an entry point for parsing, indicating that a valid program consists of an expression followed by an end-of-file token. This ensures that every input must result in a complete expression to be considered valid. The expr rule defines the syntax for arithmetical expressions. As the rule is recursive, it allows for nested expressions.

During the parsing process, the NEWLINE rule indicates that return and newline characters should be ignored and not considered when interpreting expressions. The INT rule defines how numerical literals within expressions are recognised.

Code 2.8 – ANTLR4 Grammar for Expressions

```
grammar Expr;

prog: expr EOF ;

expr: expr ('*'|'/') expr
      | expr ('+'|'-') expr
      | INT
      | '(' expr ')'
      ;

NEWLINE : [\r\n]+ -> skip;

INT     : [0-9]+ ;
```

Source: Current Author

Provided the Expr grammar, the ANTLR4 tools generate automatically the lexer and parser files for the desired target language. In this work, Java is selected as the preferred option. Depending on the flags used to compile the grammar, a few other files, such as listeners and visitors, can be made available.

Considering the Expr grammar, in Figure 4, one can see part of the file tree structure associated with the generated files. The file Expr.g4 contains the input grammar of this toy expression language. The file Expr.tokens lists the types of tokens of this language. The files ExprLexer.java and ExprParser.java comprise the lexer and parser code, respectively. Finally, ExprBaseVisitor.java implements the ExprVisitor.java interface, and it can be adapted to provide a custom visitor for the language. For instance, one that would traverse the parse tree and evaluate the expression.

Figure 4 – Files generated from the Expr grammar.

```
example/
├── Expr.g4
├── ..
├── Expr.tokens
├── ExprBaseVisitor.java
├── ExprLexer.java
├── ExprParser.java
└── ExprVisitor.java
```

It is also important to note that the ANTLR ecosystem includes a rich set of documentation and community-contributed resources, such as grammar repositories and tutorials. There are plug-ins available for popular IDEs like IntelliJ IDEA, Eclipse, and VSCode.

3 AN UPDATED THEORY FOR CSP IN COQ

In this chapter, we present our updated theory for CSP in Coq. The formalisation of CSP_{Coq} comprises more than 3,300 lines of Coq code, and is organised into the folders: `Syntax`, `Semantics`, `Automation`, and `Specification`. The first two folders contain the definition of the syntax and semantics of CSP_{Coq}. The folder `Automation` contains useful custom automation tactics. Finally, the folder `Specification` formalises the notion of a CSP specification, gluing together syntactic and semantic aspects. The code is publicly available at: <https://doi.org/10.5281/zenodo.11228166>.

In Section 3.1, we present the concrete syntax of CSP_{Coq}, covering how expressions, channels, constants, and processes are defined. In Section 3.2, we present the static and dynamic typing system of CSP_{Coq}. The typing system forms the basis for the verification of many well-formedness conditions, as described in Section 3.3, along with the associated custom verification tactics. In Section 3.4, we present the structured operational semantics (SOS) of CSP_{Coq}. The SOS is used to define the notion of labelled transition systems, which is discussed in Section 3.5, in addition to the integration with Graphviz. In Section 3.6, we formalise the notions of refinement according to the traces semantic model. Finally, in Section 3.7, we define the property of deadlock freedom.

3.1 SYNTAX

A CSP specification consists of the declaration of constants, channels, and processes. The following sections describe how these concepts are formalised in Coq, in addition to expressions. At the end, we show how all the elements are combined into a CSP specification.

3.1.1 Expressions

The *Exp* type defines various kinds of arithmetical and boolean expressions. It is important to mention that we follow a shallow embedding approach to avoid having to reuse Coq's types to represent CSP's basic types (e.g., natural numbers and boolean values). Furthermore, this design decision enables the use of Coq's functions over basic types in

CSP$_{\text{Coq}}$ specifications, to some extent, similar to the functional language supported by CSP$_{\text{M}}$. In Code 3.1, we present a fragment of the inductive definition of *Exp*.

Code 3.1 – Definition of Expression.

```
Inductive Exp : Type :=
| EBool (b : bool)
| ENum (n : nat)
| EId (id : CSPId)
| EPlus (e1 e2 : Exp)
| ...
| EEq (e1 e2 : Exp)
| ...
| EAnd (e1 e2 : Exp)
| ...

Coercion EBool : bool >-> Exp.
Coercion ENum : nat >-> Exp.
Coercion EId : CSPId >-> Exp.
```

Source: Current Author

Boolean literals and natural numbers are created by the constructors *EBool* and *ENum*, respectively. The constructor *EId* allows one to refer to CSP identifiers (e.g., a constant, a local variable) within expressions. *CSPId* is basically a string. In what follows, we have typical constructors for arithmetical (e.g., *EPlus*), comparisons (e.g., *EEq*), and boolean (e.g., *EAnd*) expressions. `Coercion`s are created to automatically promote basic types from the Coq system (i.e., boolean literals, natural numbers, and strings) to elements of the type *Exp*.

Our formalisation of expressions also accounts for sets and sequences of expressions, as it can be seen in Code 3.2. A set of expressions is defined using the Coq's *set* type, whereas sequences are represented as lists. It is not shown here, but to use sets, it is necessary to prove the decidability of the equality of expressions (theorem *exp_dec*). When adding an expression to a set, it is necessary to test whether this expression is already a member of the set.

Code 3.2 – Definition of sets and sequences of expressions.

```
Definition ExpSet : Type := set Exp.
Definition ExpSeq : Type := list Exp.
```

Source: Current Author

One might wonder whether it would be possible to represent sequences of sequences.

As just defined, *ExpSeq* represents a sequence of expressions. An expression can refer to a constant via the *EId* constructor, and constants can be defined as sequences of expressions. Therefore, it is indeed possible to represent sequences of sequences of expressions. Nevertheless, at this moment, although it is syntactically possible to represent such a structure, it is still not possible to reason (i.e., give semantics) about it due to limitations of the current typing system. Tuples are not directly supported, but they can be modelled using sequences.

To illustrate the above definitions, the expression $(i + 1)\%N$ is represented in the CSP_{Coq}'s concrete syntax as follows: ((*EMod* ((*EPlus* (*EId* "i") (*ENum* 1))) (*EId* "N"))).

3.1.2 Channels

Channels are the next building block of our theory. They are used to represent the communication (events) between processes. Code 3.3 shows how channels are declared.

Code 3.3 – Definition of Channel.

```
Inductive ChannelDeclaration : Type :=
| channel_declaration : list ChannelName → list GeneralSet → ChannelDeclaration.
```

Source: Current Author

The constructor *channel_declaration* has two arguments: a list of names (*ChannelName*) and a list of types (*GeneralSet*). The name of a channel is a string; that is, *ChannelName* refers to the Coq's string type. This constructor receives a list of names, since it is possible to declare multiple channels at once, all of them sharing the same type signature. In Code 3.4, we exemplify the declaration of the channels thinks, sits, eats, and getsup, from the dining philosophers example.

Code 3.4 – Declaration of channels thinks, sits, eats, and getsup.

```
Definition channeldec1 : ChannelDeclaration :=
  channel_declaration ["thinks" ; "sits" ; "eats" ; "getsup"] [(id_set "PHILNAMES")].
```

Source: Current Author

The type of a channel is defined by a list of *GeneralSet*. If an empty list is provided, we say that this channel declaration is actually declaring a single event. Otherwise, we will have compound events, where data is transferred between the communicating processes. Code 3.5 presents the definition of *GeneralSet*. A channel may communicate boolean values (*bool_type*), arbitrary natural numbers (*nat_type*) or restricted to a given interval (*nat_interval_type_set*). The communicated values may belong to an explicitly defined set of expressions (*exp_set*) or events (*event_set*). It is also possible to refer to CSP identifiers that were previously used to define constants (*id_set*). For instance, in the previous example, the declaration of the channel thinks refers to the constant *PHILNAMES*.

Code 3.5 – Definition of GeneralSet.

```
Inductive GeneralSet : Type :=
| nat_type : GeneralSet
| nat_interval_type_set : Exp → Exp → GeneralSet
| bool_type : GeneralSet
| exp_set : ExpSet → GeneralSet
| event_set : EventSet → GeneralSet
| id_set : CSPId → GeneralSet.
```

Source: Current Author

As said before, channels are used to represent the communication between processes, giving rise to single and compound events. Code 3.6 shows the formalisation of *Event*s. An event refers to the name of a channel, and to a list of communications. These are the parameters of the constructor *event*.

Code 3.6 – Definition of Event.

```
Inductive Event : Type :=
| event : ChannelName → list Communication → Event.
```

Source: Current Author

The *Communication* type represents different kinds of messages that can be communicated via channels. This includes input (?), output (!) and default (.) communications. When performing an input communication, one provides the name (*CSPId*) of the local variable that will store the read value. For output and default communications, one provides the communicated value, which is any valid *Exp*.

Code 3.7 – Definition of Communication.

```
Inductive Communication : Type :=
| input_communication : CSPId → Communication
| output_communication : Exp → Communication
| default_communication : Exp → Communication.
```

Source: Current Author

To illustrate, consider the following example: the event *picks*!*i*!*i* is declared in CSP_{Coq} as *event* "picks" [*output_communication* (*EId* "i") ; *output_communication* (*EId* "i")]. For an event that has no communication, such as the event *cash* considered in Section 1.2, we would have *event* "cash" *nil*. This indicates that the *cash* event does not involve any data transfer. Sets of events can be declared following the definition of *EventSet* (see Code 3.8).

Code 3.8 – Definition of EventSet.

```
Inductive EventSet : Type :=
| normal_set : set Event → EventSet
| production_set : set Event → EventSet.
```

Source: Current Author

A set of events may be created using *normal_set* and *production_set* constructors. Considering the channel declaration, a normal set comprises only complete events; whereas a production may contain incomplete events. Assuming that *PHILNAMES* refers to the interval $\{0..N-1\}$, where $N = 5$ (see Code 2.1), the CSP_M declaration $\{|thinks|\}$ is equivalent to the following set of events $\{thinks.0, thinks.1, thinks.2, thinks.3, thinks.4\}$. In CSP_Coq, we would have *production_set* (*set_add event_dec* (*event* "thinks" *nil*) *nil*). We pass to the constructor *production_set* an incomplete event (i.e., *thinks* with no communications), and add it to the empty set (*nil*). Note that the function *set_add* relies on the theorem *event_dec* that proves the decidability of the equality of events. This is necessary to ensure that a set has no repetition of elements. *Alphabet*s are basically sets of events.

To conclude our formalisation of channels and events, we have the types *SpecialEvent* and *GeneralEvent* (see Code 3.9). The former defines the events τ and $\checkmark$, whereas the latter formalises that an event may be a normal (*normal_event*) or a special (*special_event*) one.

Code 3.9 – Definition of normal and special events.

```
Inductive SpecialEvent : Type :=
| tau : SpecialEvent
| tick : SpecialEvent.

Inductive GeneralEvent : Type :=
| normal_event : Event → GeneralEvent
| special_event : SpecialEvent → GeneralEvent.
```

Source: Current Author

3.1.3 Constants

Constants are used to represent fixed values that can occur elsewhere in CSP specifications. The *ConstantDeclaration* (see Code 3.10) defines how constants are declared in CSP$_{\text{Coq}}$. A constant may refer to an expression (*exp_declaration*), to a *GeneralSet* (*set_declaration*), or to a *GeneralSeq* (*seq_declaration*). Although not presented before, a *GeneralSeq* is similar to a *GeneralSet*, but comprises sequences of elements instead of sets. When declaring a constant, as expected, one needs to provide its name (*CSPId*).

Code 3.10 – Definition of ConstantDeclaration.

```
Inductive ConstantDeclaration : Type :=
| exp_declaration : CSPId → Exp → ConstantDeclaration
| set_declaration : CSPId → GeneralSet → ConstantDeclaration
| seq_declaration : CSPId → GeneralSeq → ConstantDeclaration.
```

Source: Current Author

To illustrate, consider the example shown in Code 3.11. It declares the constant *PHILNAMES*, which consists of the general set formed by the numbers within the interval (*nat_interval_type_set*) ranging from 0 to $N - 1$, where N is another constant, whose definition is not shown here.

Code 3.11 – Definition of the constant *PHILNAMES*.

```
Definition PHILNAMES : ConstantDeclaration := set_declaration "PHILNAMES"
  (nat_interval_type_set (ENum 0) (EMinus (EId "N") (ENum 1)))
```

Source: Current Author

3.1.4 Processes

The type *ProcessDeclaration* (see Code 3.12) defines how a CSP process is declared. Its single constructor receives the process name (*CSPId*), the name of its parameters, which can be none (i.e., an empty list), and the definition of the associated process body (an element of *ProcessBody*).

Code 3.12 – Definition of ProcessDeclaration.

```
Inductive ProcessDeclaration : Type :=
| Process (name : CSPId) (ids: list CSPId) (body : ProcessBody).
```

Source: Current Author

Code 3.13 shows a fragment of the definition of *ProcessBody*. It includes various constructors to represent the basic CSP processes (*SKIP* and *STOP*), along with the different CSP operators, namely: process reference (*ProcessReference*), process prefix (*ProcessPrefix*), external choice (*ProcessExtChoice*), internal choice, alphabetised parallelism, generalised parallelism, interleaving, sequential composition, hiding, conditional guards, if-then-else conditionals, interruption, and replicated operators. Each constructor has the adequate parameters. For example, when referring to a process, the constructor *ProcessReference* takes as arguments the name of the referred process (*CSPId*), and a list of arguments (a list of *Exp*).

Code 3.13 – Definition of ProcessBody.

```
Inductive ProcessBody : Type :=
| SKIP
| STOP
| ProcessReference (name : ProcessName) (args : list Exp)
| ProcessPrefix (event : Event) (proc : ProcessBody)
| ProcessExtChoice (proc1 proc2 : ProcessBody)
| ...
| ProcessReplicated (op : ReplicatedOperator) (proc : ProcessBody).
```

Source: Current Author

The replicated operators are defined by an auxiliary type: *ReplicatedOperator*. In Code 3.14, we present a fragment of its definition. For example, the constructor *ReplicatedInterleave* is the one used to define the process *PHILS* (see Code 2.1).

Code 3.14 – Definition of ReplicatedOperator.

```
Inductive ReplicatedOperator : Type :=
| ReplicatedExtChoice (id : CSPId) (set : GeneralSet)
| ReplicatedIntChoice (id : CSPId) (set : GeneralSet)
| ...
| ReplicatedInterleave (id : CSPId) (set : GeneralSet)
| ...
```

Source: Current Author

To illustrate the above definitions, consider the process *PHILS = ||| i : PHILNAMES @ PHIL(i)* shown in Code 3.15. Its body (*PHILS_BODY*) consists of the replicated interleave (*ReplicatedInterleave*) of references (*ProcessReference*) to the process *PHIL* indexed by an *i* belonging to the set *PHILNAMES*. The process *PHILS* is then defined by invoking the constructor *Process* providing the process name, an empty list (*nil*) of arguments, and the previously defined process body.

Code 3.15 – Definition of the process PHILS.

```
Definition PHILS_BODY : ProcessBody := ProcessReplicated
  (ReplicatedInterleave "i" (id_set "PHILNAMES"))
  (ProcessReference "PHIL" [(EId "i")]).

Definition PHILS : ProcessDeclaration := Process "PHILS" nil PHILS_BODY.
```

Source: Current Author

3.1.5 Specifications

A CSP specification (in CSP$_{\text{Coq}}$, an element of *Specification* – see Code 3.16) combines the syntax (i.e., the declaration of constants, channels, and processes) and the static semantics (i.e., well-formedness conditions that can be statically verified) of CSP specifications. In Coq, a record is used to define a composite type that contains multiple fields. Records are similar to classes in object-oriented languages.

Code 3.16 – Definition of Specification.

```
Record Specification : Type := Build_Spec
{
  constants : list ConstantDeclaration;
  channels : list ChannelDeclaration;
  processes : list ProcessDeclaration;

  well_formedness_conditions : ...
}.
```

Source: Current Author

When one defines a record in Coq, it will automatically generate a constructor function for the Record (*Build_Spec* in Code 3.16). This constructor is used for creating instances of the record. The distinguishing feature of records in Coq is that the fields may be related to data (in our case, the declaration of CSP elements), but also to properties (in our case, static well-formedness conditions). Therefore, when creating an instance of the record, one needs to provide the expected data, but also prove all associated properties.

In Code 3.17, when creating the specification *phils_SPEC*, this combination of data and properties becomes evident. Note that the record instance is created within proof mode (`Proof`).

Code 3.17 – The CSP$_{\text{Coq}}$ specification of the dining philosophers example.

```
Definition phils_SPEC : Specification.
Proof.
  apply (Build_Spec
         [N ; PHILNAMES ; FORKNAMES]
         [channeldec1 ; channeldec2]
         [PHIL ; FORK ; PHILS ; FORKS ; SYSTEM]
  ).
  prove_wfcs.
Defined.
```

Source: Current Author

After invoking the constructor *Build_Spec*, the required properties are listed as proof obligations. The tactic *prove_wfcs* is a custom tactic defined in this work, and explained later, that automatically discharges these proofs. Finally, one can use the keyword `Defined` to conclude the definition of the record instance.

In the following sections, we detail the typing system (Section 3.2) that is used to define the statically verified well-formedness conditions (Section 3.3), but also other elements of the semantics (e.g., the SOS).

3.2 TYPING SYSTEM

Upon the introduction of compound events that communicate values between processes, it is necessary to consider a typing system. However, in our context, typing cannot always be performed statically. As we allow for parametrised processes and the types of parameters are left implicit, the types of arguments need to be dynamically inferred. Therefore, we account for a static and dynamic typing system. When checking static well-formedness conditions, we try to statically infer as many types as possible in order to anticipate problems. The dynamic typing system is used by the semantic models employed to analyse dynamic properties of the CSP specifications (e.g., deadlock freedom).

3.2.1 Values and types

At this moment, we support communications of natural and boolean values (see Code 3.18). The constructor *undefined_value* is used to represent that it is (still) not possible to infer the value of some expression or identifier. This occurs both during static

Code 3.18 – Definition of supported types and values.

```
Inductive Value : Type :=
| undefined_value : Value
| natural_value : nat → Value
| boolean_value : bool → Value.

Inductive InferredType : Type :=
| invalid
| undefined
| boolean
| natural.
```

Source: Current Author

(as we do not focus on evaluating expressions) and dynamic (when there is a typing error that prevents the evaluation) typing. For instance, during static analysis, if i is a parameter of a process, the value of $i + 1$ is left undefined, since it requires evaluating the associated process call. There are also elements that are not associated with an explicit value, such as processes.

The Coq type *InferredType* is used during type inference. If we are capable of inferring the type, we use the constructors *boolean* or *natural*. If it still not possible to type the term (i.e., during static typing), we use the constructor *undefined*. However, if we deduce that the term has an invalid type, we use the constructor *invalid*.

3.2.2 Execution context

Our typing system is directly associated with the notion of an execution context (see Code 3.19). This is necessary because we have both global definitions (such as constants and channels), but also local ones (such as variables introduced by input communications and replicated operators).

A context is defined as a list of *ContextEntry*, which is a record with the following five fields: *id* – the identifier of the global or local definition; *level* – a natural number indicating the declaration level (0 is used to denote the *GLOBAL_LEVEL* and increasing numbers denote inner declaration scopes); value – the value obtained after evaluating the identifier (if applicable); *type* – the inferred type; and, *decoration* – the syntactic structure associated with the current entry. In what follows, we provide more details about the last two fields.

Code 3.19 – Definition of ContextEntry and Context.

```
Definition GLOBAL_LEVEL : nat := 0.

Record ContextEntry : Type := Build_ContextEntry
{
  id : CSPId;
  level : nat;
  value : Value;
  type : list InferredType;
  decoration : DecoratedDeclaration;
}.

Definition Context : Type := list ContextEntry.
```

Source: Current Author

It is important to note that the inferred type is actually a list of *InferredType*. Consider the following examples: in $N = 0$, the inferred type of N would be [*natural*]; in $N = \{0, 1\}$, the inferred type of N would be [*natural*, *natural*]. That is the underlying reason for defining the field *type* as a list of *InferredType*.

In Code 3.20, we see the definition of *DecoratedDeclaration*, which is the type of the field *decoration* in *ContextEntry*. There are five syntactic constructions that are responsible for creating context entries. The first one (*event_decoration*) is associated with an input communication. Therefore, when we have an event such as *ev?x?y*, two entries are introduced in the context, and both of them are associated (decorated) with this event.

Code 3.20 – Definition of *DecoratedDeclaration*.

```
Inductive DecoratedDeclaration : Type :=
| event_decoration : Event → DecoratedDeclaration
| const_decoration : ConstantDeclaration → DecoratedDeclaration
| channel_decoration : ChannelDeclaration → DecoratedDeclaration
| replicated_decoration : ReplicatedOperator → DecoratedDeclaration
| parameter_decoration : ProcessDeclaration → DecoratedDeclaration
| process_decoration : ProcessDeclaration → DecoratedDeclaration.
```

Source: Current Author

The global declaration of constants, channels, and processes also introduce new entries in the context. In these situations, we use *const_decoration*, *channel_decoration*, and *process_decoration*, respectively. The parameters of a process also produce new context entries within its process body (*parameter_decoration*). Finally, the replicated operator (e.g., `||| i : {0,1} @ ...`) also creates new context entries (*replicated_decoration*).

The creation of context entries is facilitated by auxiliary functions; some of them are shown in Code 3.21. Provided the expected arguments, the functions *EventEntry* and *ProcessEntries* yield new entries for events and processes, respectively. In Code 3.21, note that `{| |}` is a Coq notation for creating record instances; it is not related to the notion of production sets of CSP. In *EventEntry*, we provide as argument a single type *t*, since each communicated value is associated with a single value/type. In *ProcessEntry*, note that no value is provided as argument, since a process is not associated with a single value. In such a situation, *undefined_value* is considered.

Code 3.21 – Auxiliary functions for creating context entries.

```
Definition EventEntry (name : CSPId) (l : nat) (v : Value)
(t : InferredType) (e : Event) : ContextEntry :=
  {| id := name ; level := l ; value := v ; type := [t] ;
     decoration := event_decoration e |}.

Definition ProcessEntry (name : CSPId) (l : nat)
(t : list InferredType) (p : ProcessDeclaration) : ContextEntry :=
  {| id := name ; level := l ; value := undefined_value ; type := t ;
     decoration := process_decoration p |}.
```

Source: Current Author

To illustrate the definitions presented in this section, let us consider the dining philosophers example, see Code 3.22 for a quick reference. In Code 3.23, we can see the context statically inferred for all global definitions; we omit the fields *value* and *decoration* for legibility purposes.

Code 3.22 – Fragment of the dining philosophers specification in CSP_M.

```
N = 5
PHILNAMES = {0..N-1}
FORKNAMES = {0..N-1}
channel thinks, sits, eats, getsup : PHILNAMES
channel picks, putsdown : PHILNAMES.FORKNAMES
PHIL(i) = ...
FORK(i) = ...
PHILS = ||| i : PHILNAMES @ PHIL(i)
FORKS = ||| i : FORKNAMES @ FORK(i)
SYSTEM = PHILS [| {|picks, putsdown|} |] FORKS
```

Source: (ROSCOE, 2010)

Code 3.23 – Global context for the dining philosophers example.

```
[
  {| id := "N" ; level := 0 ; ... type := [natural] ; ... |} ;
  {| id := "PHILNAMES" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "FORKNAMES" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "thinks" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "sits" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "eats" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "getsup" ; level := 0 ; ... type := [natural] ; ... |};
  {| id := "picks" ; level := 0 ; ... type := [natural; natural] ; ... |};
  {| id := "putsdown" ; level := 0 ; ... type := [natural; natural] ; ... |};
  {| id := "PHIL" ; level := 0 ; ... type := [undefined] ; ... |};
  {| id := "FORK" ; level := 0 ; ... type := [undefined] ; ... |};
  {| id := "PHILS" ; level := 0 ; ... type := []; ... |};
  {| id := "FORKS" ; level := 0  ; ... type := [] ; ... |};
  {| id := "SYSTEM" ; level := 0 ; ... type := [] ; ... |}
] : Context
```

Source: Current Author

The level of all entries is 0, since all of them are related to global declarations. As explained before, the types of parameters are not statically determined and, thus, are left *undefined* – see the entries for PHIL and FORK. As the processes FORKS, PHILS and SYSTEM do not have parameters, their entries are associated with an empty list of types.

Now, after explaining and exemplifying execution contexts, we are in a better position for getting into some details of static typing. Code 3.24 shows a fragment of the function that statically types expressions. It is defined by pattern-matching on the expression *exp*. If the expression concerns a boolean or a natural literal, the inferred type is *boolean* or *natural*, respectively.

If the expression refers to an identifier (i.e., *EId i*), we try to find the closest entry in the context for *i*. By closest we mean from the inner to the outer scope level. This is necessary since CSP allows for multiple local definitions of the same identifier, but with different scope levels. If no entry is found, the function *FindClosestEntry* yields *None*. In such a case, the inferred type for *exp* is *invalid* (i.e., it is referring to an undeclared identifier). If the identifier is found, *FindClosestEntry* yields *Some entry*. Then, the *entry* is going to be inspected to see whether this is a valid reference (e.g., the name of a process cannot be referred within an expression) and, then, retrieved its inferred type.

In Code 3.24, we can also see an example of static typing of arithmetic expressions. Assuming that *exp* is equal to $e1 + e2$, if the inferred types of $e1$ and $e2$ are *natural*, then, the type of *exp* is also natural. If the type of one of the inner expressions is *natural*, but the type of the other expression is *undefined*, such as the $i + 1$ in $picks!i!((i + 1)\%N)$ from Code 2.1, the type of *exp* is *undefined* as well. If the types of both expressions are *undefined*, the type of *exp* is also *undefined*. In all other situations, the type of *exp* is invalid.

Code 3.24 – Function for statically typing expressions.

```
Fixpoint StaticType_Exp (ctx : Context) (exp : Exp) : InferredType :=
  match exp with
  | EBool _ ⇒ boolean
  | ENum _ ⇒ natural
  | EId i ⇒ match FindClosestEntry ctx i with
              | None ⇒ invalid
              | Some entry ⇒ ...
              end
  | EPlus e1 e2 ⇒ match StaticType_Exp ctx e1, StaticType_Exp ctx e2 with
                    | natural, natural ⇒ natural
                    | undefined, natural ⇒ undefined
                    | natural, undefined ⇒ undefined
                    | undefined, undefined ⇒ undefined
                    | _, _ ⇒ invalid
                    end
  | ...
  end.
```

Source: Current Author

3.3 WELL-FORMEDNESS CONDITIONS

To be valid, besides adhering to the syntax of CSP$_{\text{Coq}}$, a CSP specification needs to meet a number of well-formedness conditions (WFC). Otherwise, the behaviour of processes may be undefined or invalid. All WFCs are first checked statically (i.e., when creating a CSP specification) to anticipate violations. Some of them are rechecked during dynamic semantic analysis; particularly, those related to types, since the static typing system is incomplete (i.e., some terms cannot be statically typed), as explained before. There are also WFCs that have already been handled by the ANTLR parser, but even so, we check them in the Coq implementation to ensure that specifications written directly in CSP$_{\text{Coq}}$ are also correct.

Here, it is important to say that CSP$_{\text{Coq}}$ anticipates problematic situations that are only identified by FDR during dynamic process analysis. For instance, let `P(i) = STOP`, if one declares `Q = P`, the specification is loaded successfully by FDR. However, when analysing the behaviour of `Q`, FDR raises an error. In CSP_{Coq}, this problem is statically detected. We have a total of 39 WFCs associated with expressions (Table 5), constants (Table 6), channels (Table 7), and processes (Table 8). In what follows, we list and explain all of them.

In Coq, a CSPId is defined as a string. However, in the context of CSP specifications, an identifier cannot be defined by the empty string (*wfc_exp_1*). Additionally, it should start with a letter, and be followed by letters, digits, underscores, or single quotes (*wfc_exp_2*). Although the identifier of these two WFCs starts with *wfc_exp*, they apply to all CSPIds, not only those embedded in expressions. Here, the identifier of WFCs reflects the location where the related Coq definition is. In this case, CSPId is declared when defining expressions. Other definitions such as constants, channels and processes refer to the same definition of CSPId.

Table 5 – Well-formedness conditions of CSP_{Coq} (expressions).

Identifier	**Description**
wfc_exp_1	CSPId must not be the empty string.
wfc_exp_2	CSPId must start with a letter, and be followed by letters, digits, underscores, or single quotes.
wfc_exp_3	The argument of EId must refer to a declared CSP identifier, which needs to be a constant, parameter or local variable.
wfc_exp_4	The arguments of EPlus, EMinus, EMult, EDiv, EMod, ELe, EGe, ELt, and EGt must be of type nat.
wfc_exp_5	The arguments of EEq, and ENeq must be of the same type.
wfc_exp_6	The arguments of ENot, EAnd, and EOr must be of type bool.
wfc_exp_7	All expressions in ExpSet must be of the same type.
wfc_exp_8	All expressions in ExpSeq must be of the same type.

Source: Current Author

In expressions, when referring to a CSPId, this identifier needs to be previously declared; additionally, it should be related to a constant, to a process parameter, or to a local variable (*wfc_exp_3*), which is introduced by input communications or replicated operators. Within an expression, for instance, we cannot refer to a process.

The other WFCs in Table 5 are related to type correctness: arithmetic expressions are about numbers (*wfc_exp_4*), equalities and inequalities should be applied to expressions of the same type (*wfc_exp_5*), boolean expressions are about boolean literal (*wfc_exp_6*), and we only accept homogenous sets and sequences of expressions (*wfc_exp_7* and *wfc_exp_8*).

Regarding the declaration of constants, the name of a constant must be unique at the global level (see Table 6 – *wfc_const_1*). At inner scope levels, local variables may be introduced with previously used identifiers.

Table 6 – Well-formedness conditions of CSP_{Coq} (constants).

Identifier	**Description**
wfc_const_1	In ConstantDeclaration, the name of the constant must be unique at the global scope.

Source: Current Author

Table 7 shows WFCs that are related to channels and events. When creating a channel declaration, one provides a list of channel names and a type signature (list of channel types) shared by all declared channels (e.g., `channel thinks, sits : PHILNAMES`). The type signature can be an empty list (i.e., an event is being declared), but the list of channel names must not be empty (*wfc_ch_1*). Similarly to constants, the name of channels must be unique at the global scope (*wfc_ch_2*).

Via the constructor id_set, the type of a channel can refer to an identifier defined elsewhere (e.g., `channel thinks : PHILLNAMES`). However, this identifier must be of a constant, and it must be equivalent to a set (*wfc_ch_3*). For instance, in the example just provided, it would not make sense if `PHILNAMES` were defined as a constant value (e.g., `N-1`). Recall that `PHILNAMES` is defined as `{0..N-1}`: a set of numbers ranging from `0` to `N-1`. A similar restriction applies to the constructor id_seq of GeneralSeq (*wfc_ch_4*), which is used in the context of the replicated sequential operator.

When creating sets and sequences of events (EventSet and EventSeq, respectively), any compound event (i.e., with communications) must not involve input neither output communication, but only default ones (*wfc_ch_5* and *wfc_ch_7*). Input and output communications only occur in the context of the prefix operator. When creating normal sets or normal sequences of events (opposed to production sets and production sequences), it is not possible to refer to incomplete compound events (*wfc_ch_6* and *wfc_ch_8*). For instance, `{|thinks|}` is a valid definition, since we have a production set. However, `{thinks}` would not be accepted.

As expected, all events must refer to declared channels (*wfc_ch_9*), and the types of communications must be compatible with the channel type signature (*wfc_ch_10*). Finally,

Table 7 – Well-formedness conditions of $\mathrm{CSP}_{\mathrm{Coq}}$ (channels).

Identifier	**Description**
wfc_ch_1	In ChannelDeclaration, the list of ChannelName must not be empty.
wfc_ch_2	In ChannelDeclaration, the name of the channels must be unique at the global scope.
wfc_ch_3	In GeneralSet, the argument of id_set must refer to declared constant, which comprises a set (i.e., it cannot refer to a sequence, neither to arithmetic and boolean expressions).
wfc_ch_4	In GeneralSeq, the argument of id_seq must refer to declared constant, which comprises a sequence (i.e., it cannot refer to a set, neither to arithmetic and boolean expressions).
wfc_ch_5	In EventSet, all compound events must not involve input neither output communication.
wfc_ch_6	In EventSet, the arguments of normal_set must not be incomplete compound events.
wfc_ch_7	In EventSeq, all compound events must not involve input neither output communication.
wfc_ch_8	In EventSeq, the arguments of normal_seq must not be incomplete compound events.
wfc_ch_9	In Event, the channel name must refer to a declared channel.
wfc_ch_10	In Event, the types of communications must be compatible with the channel declaration.
wfc_ch_11	In GeneralSet, the arguments of nat_interval_type_set must be of type nat.
wfc_ch_12	In GeneralSeq, the arguments of nat_interval_type_seq must be of type nat.

Source: Current Author

when creating intervals, the arguments of the constructor must be numbers (*wfc_ch_11* and *wfc_ch_12*).

Table 8 shows the WFCs associated with CSP processes. As for constants and channels, the name of a process must be unique at the global scope (*wfc_proc_1*). The name of the parameters must also be unique at their declaration scope (*wfc_proc_2*); in other words, two different parameters of the same process cannot have the same name. In the body of a process, when using the process reference operator, the name provided to the operator must be of a declared process (*wfc_proc_3*). Moreover, the number of arguments must be equal to number of declared parameters (*wfc_proc_4*).

The conditions *wfc_proc_5* to *wfc_proc_14* are about invalid recursions. Let us consider an example to illustrate the undesired situations: `P = (a -> b -> P) \ {a}`, where `a` and `b` are events, and `\ {a}` denote the hiding operator (i.e., the event `a` is hidden from outside this process). According to the SOS of CSP, which is detailed later, when unfolding the reference to `P`, we accumulate the hiding operator; that is, we would have: `P = (a -> b -> ((a -> b -> P) \ {a})) \ {a}`. This prevents the unfolding process from reaching a fix point. Therefore, such situations are not allowed in CSP for a variety of operators.

In FDR, this restrictions are checked only upon the dynamic process evaluation. Differently, in CSP_{Coq}, we do this statically, anticipating the detection of undesired situations. To do this, we statically traverse the process declarations, but also following processes references. While doing this, we accumulate which recursions are deemed valid, and which are not. When we find a process reference operator, we check whether the recursion is a valid one. This algorithm is implemented by the function *OnlyValidRecursion*.

An interesting point worth mentioning is that all Coq functions must terminate. Furthermore, in general terms, we would like the Coq infrastructure to be capable of detecting termination automatically. To accomplish this, we limit the number of recursive calls of *OnlyValidRecursion* to itself. This limit is equal to the number of declared processes. As this analysis is performed by traversing the syntactic structure of a process, we do not need to analyse any process more than once. Here, we omit further details of *Only-ValidRecursion*.

The WFCs *wfc_proc_15* and *wfc_proc_16* enforce that the expressions in guards and if-then-else commands must be of type boolean. The *wfc_proc_18* guarantees that the names introduced by input communications of an event must be locally unique; for instance, `c?v?v`, would not be accepted.

Regarding *wfc_proc_17*, we are being more strict than CSP and FDR. In CSP, let `ch` be a channel declared as `channel ch : Bool.Bool`, the following event is admissible: `ch?v`. In this situation, the variable `v` will hold the composite value of type `Bool.Bool`. At this moment, our typing system does not account for composite values and, thus, we impose the restriction that the number and types of communications in events must be precisely the same of the corresponding channel declaration. Nevertheless, the above example is possible in CSP_{Coq} with no loss of expressiveness by writing `ch?v1?v2`.

Table 8 – Well-formedness conditions of CSP_{Coq} (processes).

Identifier	**Description**
wfc_proc_1	In ProcessDeclaration, the name of a process must be unique at the global scope.
wfc_proc_2	In ProcessDeclaration, the name of the parameters must be unique at their declaration scope.
wfc_proc_3	In ProcessReference, the name must refer to a declared CSP process.
wfc_proc_4	In ProcessReference, the number of arguments of the referred process must be the same.
wfc_proc_5	In ProcessAlphaParallel, we cannot have a recursion to the current (previous) process(es).
wfc_proc_6	In ProcessGenParallel, we cannot have a recursion to the current (previous) process(es).
wfc_proc_7	In ProcessInterleave, we cannot have a recursion to the current (previous) process(es).
wfc_proc_8	In ProcessSeqComp, the LHS process cannot have a recursion to the current (previous) process(es).
wfc_proc_9	In ProcessHiding, we cannot have a recursion to the current (previous) process(es).
wfc_proc_10	In ProcessInterrupt, the LHS cannot have a recursion to the current (previous) process(es).
wfc_proc_11	In ReplicatedAlphaParallel, we cannot have a recursion to the current (previous) process(es).
wfc_proc_12	In ReplicatedGenParallel, we cannot have a recursion to the current (previous) process(es).
wfc_proc_13	In ReplicatedInterleave, we cannot have a recursion to the current (previous) process(es).
wfc_proc_14	In ReplicatedSeqComp, we cannot have a recursion to the current (previous) process(es).
wfc_proc_15	In ProcessGuard, the condition must be of type boolean.
wfc_proc_16	In ProcessIfElse, the condition must be of type boolean.
wfc_proc_17	In ProcessPrefix, the number and types of communications must be the same of the channel declaration.
wfc_proc_18	In ProcessPrefix, the names introduced by the event must be locally unique.

Source: Current Author

3.3.1 Formalisation of well-formedness conditions

Differently from tools like FDR and PAT, where WFCs are algorithmically verified from informal specifications, in CSP_{Coq}, all WFCs are formally stated as logical propo-

Code 3.25 – Definition of Specification with well-formedness conditions.

```
Record Specification : Type := Build_Spec
{
  constants : list ConstantDeclaration;
  channels : list ChannelDeclaration;
  processes : list ProcessDeclaration;

  well_formedness_conditions :
    let ...
    in
      (* wfc_exp_1, wfc_exp_2 *)
      ∀ (id : CSPId), In id ids → ValidFormat id
      ∧ ... ∧
      (* wfc_exp_4, wfc_exp_5, wfc_exp_6 *)
      ∀ (pair : Exp × Context)), In pair exps_context →
           StaticType_Exp (snd pair) (fst pair) ≠ invalid
      ∧ ... ∧
      (* wfc_ch_3 *)
      ∀ (id : CSPId), In id id_sets →
           ∃ (c : ConstantDeclaration) (g : GeneralSet),
             In c constants ∧ set_declaration id g = c

      ...
}.
```

Source: Current Author

sitions, and their verification is carried out by showing that the predicates are logically equivalent to *True*. Therefore, when we say that a given CSP specification meets all formally defined WFCs, we actually have a proof that it does.

In this section, we illustrate the formalisation of WFCs by showing some of the associated predicates. In Code 3.25, we have the definition of the record *Specification* along with the formalisation of the well-formedness conditions. As explained before, when creating an instance of this record, besides providing data for the fields *constants*, *channels*, and *processes*, one needs to prove that the predicate in *well_formedness_conditions* is logically equivalent to *True*.

In *well_formedness_conditions*, we use a let-in clause to introduce local variables that are shared among the various predicates. These definitions are omitted here due to space restrictions. The first shown predicate relates to *wfc_exp_1* and *wfc_exp_2*: the validity of CSP identifiers. The variable *ids* is a list with all CSPIds within the whole CSP specification. Then, we state that, for all *id* in *ids*, the format of *id* is valid. The auxiliary function *ValidFormat* yields a predicate that is true if, and only if, the identifier is not the empty string and its spelling adhere to the expected format.

The next shown predicate relates to the well-typedness of expressions (*wfc_exp_4*, *wfc_exp_5*, and *wfc_exp_6*). Since the type of expressions depends on the execution context, we traverse the syntactic structure of the CSP specification, and store in *exps_context* a list of pairs: the expression and the associated execution context. Then, we state that, for all pairs in *exps_context*, the statically inferred type is not invalid.

The last shown predicate states that, for all applications of *id_set* (the constructor that creates *GeneralSets* by referring to a previously declared identifier), there should exist a *ConstantDeclaration* in the CSP specification whose name is the one referred by the *id_set* application. Additionally, the predicate restricts this constant to the declaration of a constant set (see the application of the constructor *set_declaration*).

3.3.2 Automated verification

The verification of well-formedness conditions, that is, proving that the associated predicates are true, is facilitated by a set of custom tactics developed by this work. They are defined using Ltac: the Coq's language for designing new tactics. In this section, we briefly comment on some of them.

In different proof situations, we have in the proof context a hypothesis of the form $H : P_1 \lor \ldots \lor P_n$, where P_i denotes predicates. In such a situation, it is necessary to analyse the proof goal on a case basis analysis, assuming separately that each P_i is true. The tactic *flat_or_destruct* (see Code 3.26) was defined by us to aid on such a situation. It destructs H in a flat way; that is, generating n proof goals, each one of them with the respective P_i in the proof context. This tactic illustrates that proof automation can be defined using patterns, similarly to functions in Gallina. The tactic examines the proof environment (in Code 3.26, referred as *goal*), and searches for a hypothesis of the form $\_ \lor \_$. Then, it repeatedly applies the `destruct` tactic to perform case analysis.

Code 3.26 – Automation tactic: flat_or_destruct.

```
Ltac flat_or_destruct :=
  match goal with
  | H : _ ∨ _ ⊢ _ ⇒ repeat (destruct H as [H | H])
  end.
```

Source: Current Author

Code 3.27 illustrates another important feature of Ltac: proof automation with backtracking. For instance, when proving *wfc_ch_3* (see the previous section), one needs to find a *ConstantDeclaration* that meets certain criteria. To deal with the existential quantification, we apply the `eexists` tactic, which performs a symbolic existential instantiation; that is, introducing a symbol (e.g., $?x$) that will be bound later. In such a situation, we typically have a proof goal of the form $?x = e_1 \vee (x = e_2 \vee (x = e_3 \vee ...))$.

The tactic *find_declaration* (see Code 3.27) is used to faciliate the proof of such goals. First, it applies the `left` tactic that modifies the proof goal leaving only the left side of the disjunction (i.e., $?x = e_1$ in our example). Then, we apply `reflexivity` that assumes that $?x$ is equal to e_1 and substitutes this symbolic variable $?x$ by e_1 in the remaining proof goals. However, if for some reason proving the remaining goals fails, the tactic *find_declaration* backtracks; this is indicated by the symbol $+$. After backtracking, it performs `right`, which leaves the proof goal as $?x = e_2 \vee (x = e_3 \vee ...)$. By a recursive call to itself, the tactic now tries `left` and `reflexivity` again. In summary, we are iterating over the equalities in the disjunction and try whether we can complete the proof assuming that some e_i is the accurate witness for the existential quantification.

Code 3.27 – Automation tactic: find_declaration.

```
Ltac find_declaration :=
  (left ; reflexivity)
  +
  (right ; find_declaration).
```

Source: Current Author

Tactics such as the ones just presented are then combined to create decision procedures that try to prove all stated well-formedness conditions. The top-level tactic is `prove_wfcs`, which tries to prove all 39 WFCs. This is precisely the tactic that was shown in Code 3.17, which concludes the creation of the CSP_{Coq} specification for the dinning philosophers example.

Considering this running example, the average time to prove all WFCs is of about 10 seconds[1]. Table 9 shows the time required to prove each WFC for a given run. As it can be seen, about 80% of the proof time is spent while proving only three WFCs: *wfc_exp_3*, *wfc_ch_10*, and *wfc_proc_17*.

[1] Metrics collected on an i7 @ 2.40GHz x 4, with 8 GB of RAM running Ubuntu 20.04.2 LTS.

Table 9 – Well-formedness conditions of CSP_{Coq} (expressions).

Well-formedness conditions	**Proof time**
wfc_exp_1, wfc_exp_2	0.608 secs (0.586u,0.s) (success)
wfc_exp_3	1.203 secs (1.202u,0.s) (success)
wfc_exp_4, wfc_exp_5, wfc_exp_6	0.258 secs (0.257u,0.s) (success)
wfc_exp_7	0.01 secs (0.01u,0.s) (success)
wfc_exp_8	0.012 secs (0.012u,0.s) (success)
wfc_ch_1	0.001 secs (0.001u,0.s) (success)
wfc_ch_2, wfc_const_1, wfc_proc_1	0.408 secs (0.408u,0.s) (success)
wfc_ch_3	0.022 secs (0.022u,0.s) (success)
wfc_ch_4	0.015 secs (0.015u,0.s) (success)
wfc_ch_5	0.018 secs (0.018u,0.s) (success)
wfc_ch_6	0.004 secs (0.004u,0.s) (success)
wfc_ch_7	0.004 secs (0.004u,0.s) (success)
wfc_ch_8	0.003 secs (0.003u,0.s) (success)
wfc_ch_9	0.142 secs (0.141u,0.s) (success)
wfc_ch_10	4.696 secs (4.669u,0.019s) (success)
wfc_ch_11, wfc_ch_12	0.042 secs (0.038u,0.003s) (success)
wfc_proc_2	0.003 secs (0.003u,0.s) (success)
wfc_proc_3, wfc_proc_4	0.029 secs (0.029u,0.s) (success)
wfc_proc_5, wfc_proc_6, wfc_proc_7, wfc_proc_8, wfc_proc_9 wfc_proc_10, wfc_proc_11, wfc_proc_12, wfc_proc_13, wfc_proc_14	0.103 secs (0.102u,0.s) (success)
wfc_proc_15, wfc_proc_16	0.007 secs (0.007u,0.s) (success)
wfc_proc_17	2.289 secs (2.278u,0.007s) (success)
wfc_proc_18	0.05 secs (0.05u,0.s) (success)
total time:	**9.995 secs** (9.92u,0.039s) (success)

Source: Current Author

To prove these three WFCs, we need to do something similar to the *find_declaration* tactic. Optimising the search, which is currently linear, may reduce the time required. Furthermore, the most time consuming proof (i.e., the one associated with *wfc_ch_10*) involves traversing the syntactic structure and building pairs of contexts and events. We believe that there are also opportunities for improving how these elements computed are manipulated during the proof. Investigating these opportunities is left as future work.

3.4 STRUCTURED OPERATIONAL SEMANTICS

Since this work significantly extended and modified the syntax and static semantics of CSP_{Coq}, up to this point, there was little reuse of the work of Freitas (2020). Regarding the remaining sections of this chapter, namely the SOS, the definition of LTSs, and the notion of traces refinement, the level of reuse increased.

Concerning the structured operational semantics, rules about operators that are not affected by the context (e.g., internal and external choices) were just lifted to the current development. However, many other rules needed to be further updated, since they affect or are affected by the environment. Other rules were created from scratch, since they are related to operators not previously considered. In what follows, we detail some of these situations.

A structured operational semantics describes how individual steps of a computation takes place. In our case, given an initial state, how the behaviour of a CSP process evolves by the employed operators. The CSP_{Coq} SOS is defined as relation (*sosR*) associating *State*s (previous and after states) by means of a *GeneralEvent* (denoting the computation). A state is defined as the combination (pair) of a *ProcessBody* and its *Context* – see Code 3.28. We define the notation P @ C to represent the pair (P,C), where P is a *ProcessBody* and C is a *Context*.

The SOS is defined inductively by enumerating all rules that allow the creation of valid associations between states and events. In Code 3.28 the rule for successful termination is shown (*success_termination_rule*). If the state is characterised by the *SKIP* process and an arbitrary context *ctx*, by the application of this rule we reach the state (*STOP*, *ctx*). This is an example of a rule that was straightforwardly lifted from the work of Freitas (2020): it suffices to incorporate the context to the rule, and state that SKIP does not change it. Note that we use the notation S1 // ev ==> S2 to denote that, according to the SOS, state S1 is related to state S2 by the event ev.

Code 3.28 – The sosR relation.

```
Definition State : Type := ProcessBody × Context.
Notation "P '@' C" := (P,C) (at level 110).

Reserved Notation "S1 '//' a '==>' S2" (at level 150, left associativity).
Inductive sosR : State → GeneralEvent → State → Prop :=
| success_termination_rule (ctx : Context) :
  (SKIP @ ctx // special_event tick ==> STOP @ ctx)
```

Source: Current Author

A more interesting rule is the *reference_rule* one (see Code 3.29). Let *P* be a process reference to the process whose name is *name*, and *args* be the passed arguments, the source state of the rule is then represented as *P* @ *ctx*. The target state is *Q* @ *ctx'*, where *Q* is the process body associated with the process of name *name*. The context *ctx'* of the target state is defined by updating *ctx* in light of the provided arguments. This update introduces to the context local variables, whose names are those of the parameters of *Q*, and whose values are those obtained from the dynamic evaluation of *args*. Additionally, when updating the context, if the types of the parameters of *Q* were *undefined* up to this point, they are updated accordingly based on the values of *args*. Note that we also require *args* to be valid arguments (i.e., the obtained values are consistent with the types of the parameters). The state (*P*,*ctx*) is associated with (*Q*,*ctx'*) by means of the special event τ, which represents the unfolding of the process reference.

Code 3.29 – The reference_rule of the SOS.

```
| reference_rule (ctx : Context) (name : CSPId) (args : list Exp) :
  ∀ (ctx' : Context) (P Q : ProcessBody),
      P = ProcessReference name args →
      Some Q = ContextProcessBody ctx name →
      ValidArguments ctx name args →
      ctx' = UpdateContextByArguments ctx name args →
      (P @ ctx // special_event tau ==> Q @ ctx')
```

Source: Current Author

Code 3.30 shows the rule for the prefix operator. Here, the most interesting part concerns the evaluation of an event; particularly, when it involves input communications. First, the initial *ctx* needs to be updated; new variables are introduced to the context. Second, depending on the type signature of the channel, we may read different values from the input communication. Therefore, by the application of this rule, we may be able to reach different target states from the same source one. In Code 3.30, this is accomplished via the universally quantified variable *events*. The function *eval_Event* evaluates the event given the source context, and yields all possible concrete events we may have. For instance, let `ch` be defined as `channel ch : {0,2,4}`, when we evaluate `ch?x`, we obtain a set with three events: {`ev.0`, `ev.2`, `ev.4`}. We require that all events in *events* are valid with respect to the declaration of the associated channel. This verification is done with the aid of the function *Forall*. It creates a predicate by the application the *ValidEvent* function to each event in *events*. Finally, note in Code 3.30 that the source state is related to the target one by any *ev'* in *events*.

Code 3.30 – The prefix_rule of the SOS.

```
| prefix_rule (ctx : Context) (ev : Event) (P : ProcessBody) :
  ∀ (ctx' : Context) (events : set Event) (ev' : Event),
      events = (eval_Event ctx ev) →
      Forall (fun e ⇒ ValidEvent ctx e) events →
      In ev' events →
      ctx' = UpdateContextByEvent ctx ev ev' →
      (ProcessPrefix ev P @ ctx // normal_event ev' ==> P @ ctx')
```

Source: Current Author

Another interesting situation arises by the need to merge different contexts. See Code 3.31. This rule concerns the generalised parallelism when the behaviour of both processes evolve based on the occurrence of a synchronous event. Here, as before, we need to evaluate the events in the synchronisation alphabet; *events* stores all obtained events.

Code 3.31 – The gener_parall_joint_rule of the SOS.

```
| gener_parall_joint_rule (ctx : Context) (P Q : ProcessBody) (A : Alphabet) :
  ∀ (ctx' ctx'' ctx''' : Context) (events : set Event) (P' Q' : ProcessBody) (ev : Event),
      events = eval_Alphabet ctx A →
      Forall (fun e ⇒ ValidEvent ctx e) events →
      In ev events →
      (P @ ctx // normal_event ev ==> P' @ ctx') →
      (Q @ ctx // normal_event ev ==> Q' @ ctx'') →
      ctx''' = MergeContexts ctx' ctx'' →
      (ProcessGenParallel P A Q @ ctx // normal_event ev ==>
       ProcessGenParallel P' A Q' @ ctx''')
```

Source: Current Author

To illustrate the need to merge contexts, consider the example shown in Code 3.32. When the event `ch.0.false` occurs, the context of `P` evolves such that now there is a local variable called `v1` whose value is `0`. However, regarding `Q`, we have two different local variables: `v2`, and `v3`. Therefore, we need to merge both contexts. One process cannot access the variables of the other, since we have already statically checked that this does not occur. Our semantics has a limitation that, at this moment, in `Q`, we would not allow for the use of the name `v1`. We plan to lift this restriction in the future. In Code 3.31, the function *MergeContexts* is responsible for merging the contexts reached by the evolution of P and Q: *ctx'* and *ctx"* , respectively.

Code 3.32 – Illustrating the need to merge execution contexts.

```
channel ch : {0,1}.{false,true}

P = ch?v1.false -> STOP
Q = ch?v2?v3 -> STOP
R = P [| {|ch|} |] Q
```

Source: Current Author

Regarding the relation *sosR*, we still need to take into account the replicated operators. This was left as future work. We need to investigate which approach is better: to define specific SOS rules or to process the syntactic structure of replicated operators to replace it by an equivalent representation.

3.5 LABELLED TRANSITIONS SYSTEMS

The behaviour of a CSP process can be described by its labelled transition system, which is built from the SOS. Here, it was quite straightforward to lift the definition of Freitas (2020) to the current development of CSP_{Coq}. The notion of LTSs is also defined inductively (*ltsR* – see Code 3.33), as the SOS. It sufficed to adapt the types involved to consider a different characterisation of states, which now have an associated context. Therefore, we do not get into too many details here, but only give an overview.

A *Transition* can be seen as a triple (s, ev, s') relating two states s and s' by means of the event ev. In Coq, this is represented using pairs: $((s, ev), s')$. In Code 3.33, *prod* is the Coq's constructor of pairs. The relation *ltsR* is inductively defined, and follows a breath-first search principle. Its members are triples of the form (T, S, S'), where T is a set of transitions, S denotes the states still to be covered, and S' denotes the covered states.

Code 3.33 – The definition of LTS.

```
Definition Transition := prod (prod State GeneralEvent) State.
Definition LTS := set Transition.
Inductive ltsR : LTS → set State → set State → Prop :=
| lts_empty_rule (visited : set State) : ltsR nil nil visited
| lts_inductive_rule ...
```

Source: Current Author

Initially, S contains only the initial state of the corresponding CSP process, and S' is empty. The LTS is built by the application of two rules (*lts_empty_rule* and *lts_inductive_rule*). First, according to the SOS, we determinate the transitions emanating from the initial state. Then, we update T, S and S' accordingly: new transitions are added to T, the initial state is moved from S to S', and the new reached states are added to S. By following an inductive principle, we proceed until S becomes empty; that is, when there are no new states to be covered.

To finish, we lift the relation *ltsR* to deal with CSP$_{\text{Coq}}$ specifications (see Code 3.34; some details are omitted). We say that *specification_lts spec name lts* is true if, and only if, the following holds. There is a process declaration P in *spec*, whose name is equal to *name*, and *ltsR* holds from the initial state, which is characterised by the process body of P and the evaluation of the global context (*evaluated_ctx*).

Code 3.34 – Lifting the relation ltsR to CSP$_{\text{Coq}}$ specifications.

```
Definition specification_lts (spec : Specification)
(name : CSPId) (spec_lts : LTS) : Prop :=
let ... in
  ∃ (P : ProcessDeclaration),
    In P spec.(processes) ∧ NameProcess P = name
    ∧ ...
    ∧ ltsR spec_lts [(BodyProcess P, evaluated_ctx)] nil.
```

Source: Current Author

3.5.1 Graphviz integration

Graphical representation of LTSs is enabled using the DOT language and the visualisation software Graphviz. We define in Coq printing functions that yield textual (*string*) representations of the elements involved in the CSP$_{\text{Coq}}$ theory of labelled transition systems (i.e., from arithmetical and boolean expressions to transitions).

The top-level function is *generate_dot* (see Code 3.35). It calls the auxiliary function *style_initial_state* to style the initial state according to the DOT notation. We assume that the initial state is the source state of the first transition and, thus, we perform pattern-matching on this transition. The other states and transitions of the LTS are formatted by iterating over the complete list of transitions *lts*; this is done by the recursive function *generate_dot_aux*.

Code 3.35 – Functions to generate DOT representation.

```
Definition style_initial_state (s0 : State) : string :=
  "<" ++ (State2String s0) ++ "> [style=bold, color=red];".

Definition generate_dot (lts : LTS) : string :=
  match lts with
  | [] ⇒ ""
  | (s0, _, _) :: _ ⇒
      "digraph LTS { " ++ (style_initial_state s0) ++ (generate_dot_aux lts) ++ " }"
  end.
```

Source: Current Author

Code 3.36 shows a fragment of the DOT code generated for Figure 11. The first `< ... >` denotes the initial state, which is styled in bold and red. Afterwards, we have the definition of all transitions of the LTS. The source and target states are separated by `->`, and the associated label is provided using the keyword `label`.

Code 3.36 – Fragment of generated DOT code.

```
digraph LTS {
  < ... > [style=bold, color=red];
  < ... > -> < ...> [label=<tau>];
  < ... > -> < ... > [label=<thinks.0>];
} }
```

Source: Current Author

In the next chapter, when addressing our tool support for CSP$_{\text{Coq}}$, we show the graphical representation of the first states of the LTS for the dining philosophers example.

3.6 TRACES REFINEMENT

Similarly to LTSs, refinement in the traces model is another concept that was easily lifted from the work of Freitas (2020). A trace is a list of events (see Code 3.37) that can be performed from a given state. In a trace, we can have normal events, but also the special event ✓. It should be noted that, although Coq can handle infinite lists, we currently consider traces to have a finite number of events. Nevertheless, the semantics of a process can be describe by an infinite set of traces.

The relation *traceR* defines which traces are valid for a given source state. This relation is built from three rules. The first one (*empty_trace_rule*), states that the empty trace is a valid trace for any state. The second rule (*event_trace_rule*) accounts for the evolution of the process behaviour by the occurrence of visible events (normal ones and ✓). Provided that e is not τ, to show that *traceR s1* (e :: *tl*) holds, one needs to prove that, according to the SOS, the state *s2* is reached from *s1* by performing e, and that *traceR s2 tl* holds. The third rule (*tau_trace_rule*) allows for the evolution of the process behaviour by the occurrence of internal events (i.e., τ).

Code 3.37 – Definition of trace and traceR.

```
Definition Trace := list GeneralEvent.
Inductive traceR : State → Trace → Prop :=
| empty_trace_rule (s : State) :
  traceR s nil
| event_trace_rule (s1 s2 : State) (e : GeneralEvent) (tl : Trace) :
  ¬ eq e (special_event tau) →
  (s1 // e ==> s2) →
  traceR s2 tl →
  traceR s1 (e :: tl)
| tau_trace_rule (s1 s2 : State) (t : Trace) :
  (s1 // (special_event tau) ==> s2) →
  traceR s2 t →
  traceR s1 t.
```

Source: Current Author

The definition *traceRefinement* (see Code 3.38) states that a set of traces *T1* is refined by another set of traces *T2* if, and only if, *T2* ⊆ *T1*; *incl* is the Coq function for checking that *T2* is a subset of *T1*. Let *name1* and *name2* be the names of two CSP processes, when lifting the notion of traces refinement to CSP$_{\text{Coq}}$ specifications, we say that the process with name *name1* (P_{name1}) is refined by the process with name *name2* (P_{name2}), denoted by $P_{\text{name1}} \sqsubseteq_t P_{\text{name2}}$, if, and only if, *traces1* is the traces of P_{name1}, *traces2* is the traces of P_{name2}, and *traceRefinement traces1 traces2* holds. Here, we omit the definition of *specification_trace* that lifts the notion of traces to CSP$_{\text{Coq}}$ specifications.

Code 3.38 – Lifting the notion of traces refinement to CSP_{Coq} specifications.

```
Definition traceRefinement (T1 T2 : list Trace) : Prop := incl T2 T1.

Definition specification_traceRefinement
(spec : Specification) (name1 name2 : CSPId) : Prop :=
  ∀ (traces1 traces2 : set Trace),
    (∀ (t : Trace), In t traces1 ↔ specification_trace spec name1 t)
    ∧ (∀ (t : Trace), In t traces2 ↔ specification_trace spec name2 t)
    ∧ traceRefinement traces1 traces2.
```

Source: Current Author

3.7 DEADLOCK FAILURES FREEDOM

To conclude this chapter, let us address the deadlock (freedom) concept by focusing on deadlock analysis according to the *failures model*. The failures model is a structured approach to understanding and detecting deadlocks by looking at the sequences of events the process can perform and the sets of events they can refuse after a given trace. While the failures model is powerful, it does not account for infinite internal activities, known as divergences, which can also lead to unresponsive states in processes. Taking into account divergences requires considering the failures-divergence model, which will be explored in future work. With this in mind, see Code 3.39. A state *s* is in deadlock (*deadlock_state*) if, and only if, there are no emanating transitions from it; that is, according to the SOS, we cannot find a state *s'* that can be reached from *s* by performing an event *e*.

The definition *deadlock_lts* lifts this notion to LTSs. If the LTS is empty (i.e., there are no transitions emanating from the initial state), we have a deadlock. Another possibility arises if there is a transition *t* in the LTS that reaches a state in deadlock.

Code 3.39 – Lifting the notion of deadlock to CSP_{Coq} specifications.

```
Definition deadlock_state (s : State) : Prop :=
  ¬(∃ (s' : State) (e : GeneralEvent), s // e ==> s').
Definition deadlock_lts (lts : LTS) : Prop :=
    (length lts = 0) ∨
    (∃ (t : Transition), In t lts ∧ deadlock_state (snd t)).
Definition specification_deadlock
(spec : Specification) (name : CSPId) : Prop :=
  ∀ (lts : LTS), specification_lts spec name lts → deadlock_lts lts.
```

Source: Current Author

Finally, the definition *specification_deadlock* lifts the notion of deadlock to CSP_{Coq}

specifications. Let *name* be the name of a CSP process in *spec*, we say that there a deadlock in *name* if, and only if, the LTS of *name* has a deadlock according to the previous definition *deadlock_lts*. To prove deadlock freedom of a given CSP process, it suffices to prove the negation of *specification_deadlock*.

The running example of the dining philosophers is not deadlock free since the philosophers can reach a state where they are all holding a fork and waiting for the other fork to be released. However, the proof of this result in our Coq characterisation, along with an analysis of the required effort, is left as future work.

4 TOOL SUPPORT FOR CSP$_{\text{Coq}}$

In this chapter, we present the tools we have developed to foster the use and adoption of CSP$_{\text{Coq}}$. First, in Section 4.1, we discuss how we have used ANTLR4 to create an automatic translator from CSP$_{\text{M}}$ to CSP$_{\text{Coq}}$. Afterwards, in Section 4.2, we show how this translator is integrated into VSCode as an extension. Finally, in Section 4.3, we describe how Graphviz can be used to render graphical representations of labelled transition systems.

4.1 TRANSLATION FROM CSP$_{\text{M}}$ TO CSP$_{\text{Coq}}$

Our integration with the ANTLR4 tools is via a Java project. With the ANTLR support, provided a lexical and syntactic grammar for CSP, it automatically generates the Java code for a lexer and a parser. Within the lexer, we establish the guidelines that define how the input strings are broken down into meaningful tokens. These tokens include keywords, identifiers, literals, and CSP operators. In Figure 5, we show a fragment of our lexical grammar. We use regular expressions to define the format of valid `CSPID`s and natural literals. In the lexical grammar, we also define how block and line comments are delimited: `{- ... -}` and `-- ...`, respectively.

Figure 5 – Fragment of the supported Lexical grammar of CSP.

```
CSPID: [a-zA-Z][a-zA-Z0-9_]*;
NATLITERAL: [0-9]+;
...
LBRACE : '{';
RBRACE : '}';
LFATBRACE : '{|';
RFATBRACE : '|}';
...
BLOCKCOMMENT: '{-' .*? '-}' -> skip;
LINECOMMENT:  '--' ~[\r\n]* -> skip;
```

Source: Current Author

Once the input is broken down into tokens, the parser will determine how these tokens will come together to form the syntax of the language. The syntactic grammar is responsible for describing how tokens are combined to construct valid syntactic elements of the language. In Figure 6, we present a fragment of our syntactic grammar. A *spec* (in our case, a CSP specification) can be either empty or contain any of the following declarations, followed by the end of file marker: `constant_declaration`, `channel_declaration`, and `process_declaration`.

Figure 6 – Fragment of the supported syntactic grammar of CSP.

```
spec : (constant_declaration | channel_declaration | process_declaration)* EOF ;
...
channel_declaration :
  CHANNEL (CSPID (COMMA CSPID)*)
| CHANNEL (CSPID (COMMA CSPID)*) COLON (general_set (DOT general_set)*)
;
...
process_declaration :
  CSPID DEFSYMBOL process_body
| CSPID LPAR RPAR DEFSYMBOL process_body
| CSPID LPAR (CSPID (COMMA CSPID)*) RPAR DEFSYMBOL process_body
;
...
process_body :
  PSKIP
| PSTOP
| CSPID (LPAR (exp (COMMA exp)*)? RPAR)?
| event PREFIX process_body
| process_body EXTCHOICE process_body
| process_body INTCHOICE process_body
| process_body LBRACKET alphabet PARALLELISM alphabet RBRACKET process_body
| process_body LFATBRACKET alphabet RFATBRACKET process_body
| process_body INTERLEAVE process_body
| process_body SEQUENTIAL process_body
| process_body HIDING alphabet
| exp GUARD process_body
| IF exp THEN process_body ELSE process_body
| process_body INTERRUPT process_body
| replicated_operator process_body
;
```

Source: Current Author

It is possible to declare single events (channels with no communications), but also channels involving communications. In the latter, the term `general_set` represents the type of the i-th channel communication. Although not presented in Figure 6, the definition of `general_set` is aligned with the Coq formalisation discussed in Chapter 3. Actually, this applies to all elements of these grammars.

A `process_declaration` associates a `CSPID` with a `process_body`. The process may also have parameters. The definition of `process_body` mimics the definition of *ProcessBody* (see Code 3.13).

To integrate with ANTLR4 tools, a number of configurations are required in our Java project. We need to specify the project's main class, such that it is known as the entry point for the execution of the application; in our case, it is the class `Translate.java`. Additionally, we include ANTLR 4.13.1 as a dependency for generating the lexer, the parser, and the default visitor codes. Finally, aiming at facilitated distribution and deployment, we configure the creation of a JAR file (`cspcoq`) that bundles all required components, including the Coq files for our CSP_{Coq} theory.

The translation from CSP_{M} to CSP_{Coq} is implemented as methods of the visitor class `CSPEvalVisitor.java`: while traversing the syntax tree obtained from parsing the source file, we generate the corresponding Coq code that represents the CSP specification in CSP_{Coq}. The translator was extensively tested with JUnit5[1]. We used EclEmma[2] to analyse the coverage of our test campaign. As shown in Figure 7, we achieved a 100% statement coverage on our custom visitors, which means that our manually written CSP examples successfully tested (covered) all code paths implemented within the different types of visitors for translating CSP specifications.

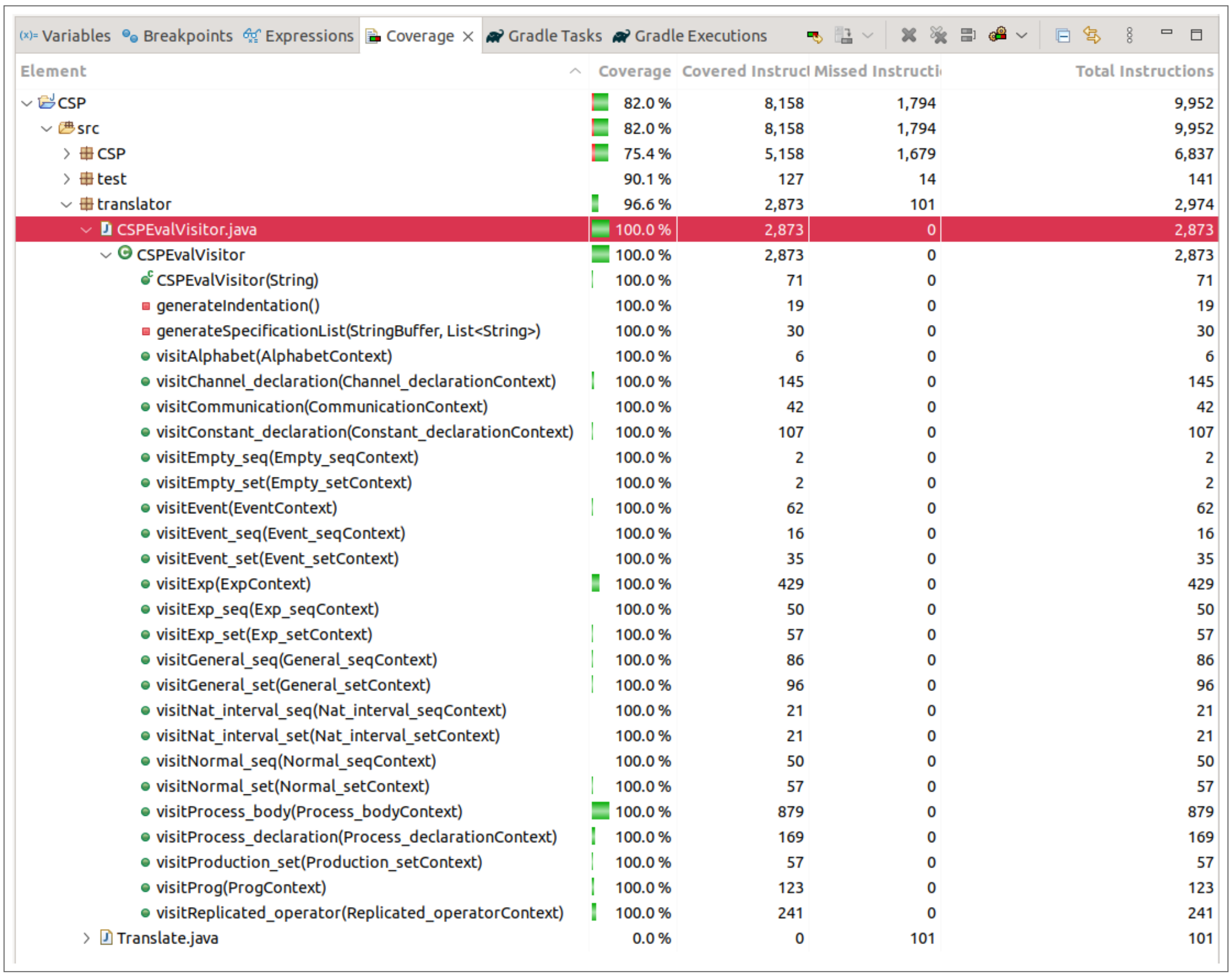
4.2 AN EXTENSION FOR VSCODE

Visual Studio Code, commonly called VSCode, is a popular open-source code editor that Microsoft maintains. According to the latest Stack Overflow survey (OVERFLOW, 2023), it was the preferred Integrated Development Environment (IDE) among professional developers and those learning to code. Due to its rich extensibility model, VSCode supports many languages and tools with extensions.

Given this community-driven approach and an updated documentation of VSCode, we have decided to develop a VSCode extension (called `CSPcoq`) to facilitate the use of our translator from CSP_{M} to CSP_{Coq}. First, we started the extension project following the standard guidelines using their Yeoman generator. A few project configurations, such as extension type and package manager, can be established upfront through the generator.

[1] Link: <https://junit.org/junit5/>
[2] Link: <https://www.eclemma.org/>

Figure 7 – Coverage metrics of our test campaign.



Element	Coverage	Covered Instruct	Missed Instructi	Total Instructions
CSP	82.0 %	8,158	1,794	9,952
src	82.0 %	8,158	1,794	9,952
CSP	75.4 %	5,158	1,679	6,837
test	90.1 %	127	14	141
translator	96.6 %	2,873	101	2,974
CSPEvalVisitor.java	100.0 %	2,873	0	2,873
CSPEvalVisitor	100.0 %	2,873	0	2,873
CSPEvalVisitor(String)	100.0 %	71	0	71
generateIndentation()	100.0 %	19	0	19
generateSpecificationList(StringBuffer, List<String>)	100.0 %	30	0	30
visitAlphabet(AlphabetContext)	100.0 %	6	0	6
visitChannel_declaration(Channel_declarationContext)	100.0 %	145	0	145
visitCommunication(CommunicationContext)	100.0 %	42	0	42
visitConstant_declaration(Constant_declarationContext)	100.0 %	107	0	107
visitEmpty_seq(Empty_seqContext)	100.0 %	2	0	2
visitEmpty_set(Empty_setContext)	100.0 %	2	0	2
visitEvent(EventContext)	100.0 %	62	0	62
visitEvent_seq(Event_seqContext)	100.0 %	16	0	16
visitEvent_set(Event_setContext)	100.0 %	35	0	35
visitExp(ExpContext)	100.0 %	429	0	429
visitExp_seq(Exp_seqContext)	100.0 %	50	0	50
visitExp_set(Exp_setContext)	100.0 %	57	0	57
visitGeneral_seq(General_seqContext)	100.0 %	86	0	86
visitGeneral_set(General_setContext)	100.0 %	96	0	96
visitNat_interval_seq(Nat_interval_seqContext)	100.0 %	21	0	21
visitNat_interval_set(Nat_interval_setContext)	100.0 %	21	0	21
visitNormal_seq(Normal_seqContext)	100.0 %	50	0	50
visitNormal_set(Normal_setContext)	100.0 %	57	0	57
visitProcess_body(Process_bodyContext)	100.0 %	879	0	879
visitProcess_declaration(Process_declarationContext)	100.0 %	169	0	169
visitProduction_set(Production_setContext)	100.0 %	57	0	57
visitProg(ProgContext)	100.0 %	123	0	123
visitReplicated_operator(Replicated_operatorContext)	100.0 %	241	0	241
Translate.java	0.0 %	0	101	101

Source: Current Author

In our case, we selected the Extension TypeScript type, since this is one of the types that facilitates manual installation and installation via the marketplace. The generator offers alternative types of extensions like Colour Theme and Code Snippets as well.

Our extension follows the default file structure of an extension. The main files we have changed are the following ones: (i) the `package.json` file, which is called the extension manifest and contains all the information and configuration of the extension; and (ii) the `extension.ts` file, which has the extension source code. The anatomy of our extension can be seen in Figure 8.

An extension can immediately work when activated, but there are ways to configure commands the user can trigger under certain conditions. Such commands are possible through the VSCode API, which contains a set of JavaScript APIs for developers. Every command has a unique identifier, and they can be manually accessed from the UI inside the command palette or have a key binding configured to trigger the command directly.

Figure 8 – Files structure of our extension to VSCode.

```
cspcoq-extension/
├── .vscode/
├── lib/
│   └── cspcoq-1.0.0.jar
├── src/
│   └── extension.ts
├── .gitignore
├── README.md
├── cspcoq-1.0.0.vsix
├── package.json
└── tsconfig.json
```

Following the first possibility, inside the `package.json` file, a command with the name `cspcoq.translate` is created, which is referenced in the `extension.ts` file. Through a `registerCommand` function, we integrate the command with the translator we developed. When the translator is called, two conditions must be met: a file has to be open, and its extension must be .csp. After fulfilling these conditions, there is a call to the `runJavaProcess` function. Here, we access the lib folder to find the `.jar` file generated from the ANTLR Java-based project, which handles the translation. We also extract from the `.jar` and copy to the current folder the Coq files that formalise the CSP_{Coq} theory.

Along with our extension, we recommend using the official VSCode extension for Coq, which is called VsCoq[3]. It allows for Coq syntax highlight and enable proof checking within the VSCode environment. Since our project uses Coq versions under 8.17, we recommend using the VsCoq Legacy extension that is compatible with versions greater than or equal to 8.7 until 8.17. For CSP_{M} syntax highlighting, we suggest the use of the extension called CSP Language Support[4]. The combination of these three VSCode extensions (CSP Language Support, VsCoq, and ours – CSPcoq) provides an integrated environment for writing CSP specifications and analysing their behaviour with the support of the Coq proof assistant.

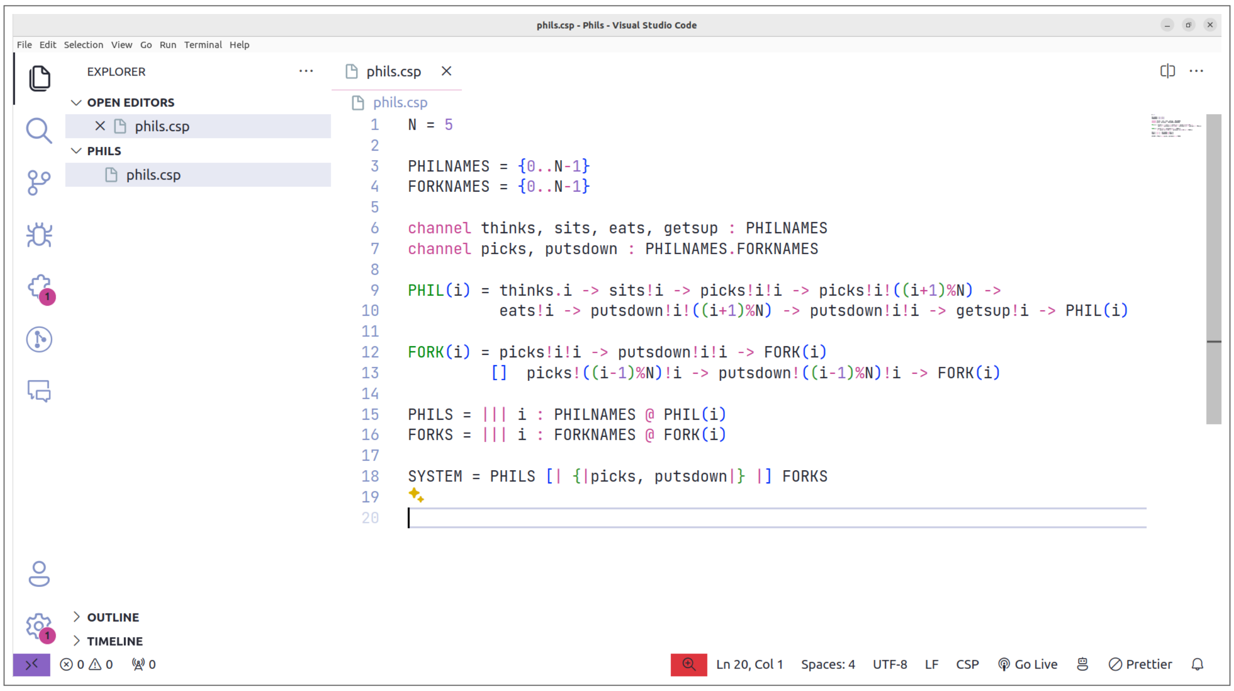
To publish our extension to the VSCode marketplace, it is necessary to create an or-

[3] Link: <https://marketplace.visualstudio.com/items?itemName=coq-community.vscoq1>
[4] Link: <https://marketplace.visualstudio.com/items?itemName=LongPham.cspsupport>

ganisation inside Azure DevOps, besides a few other configuration steps. At this moment, for simplification, we opted for manual installation via a `.vsix` file. The manual installation process works as follows: (i) one should click on the extensions icon, (ii) click on the kebab button and choose "*Install from VSIX...*", and (iii) select the desired `.vsix` file; in our case, `cspcoq-1.0.0.vsix`.

After installing and activating the extension, it is necessary to reload the VSCode window. By openning a `.csp` file with CSP$_M$ code, our translator becomes available via a custom command, as explained before. Figure 9 shows the CSP$_M$ specification for the dining philosophers example in VSCode. Note on the left panel that there are no other files in the current folder.

Figure 9 – The dinning philosophers example in VSCode.

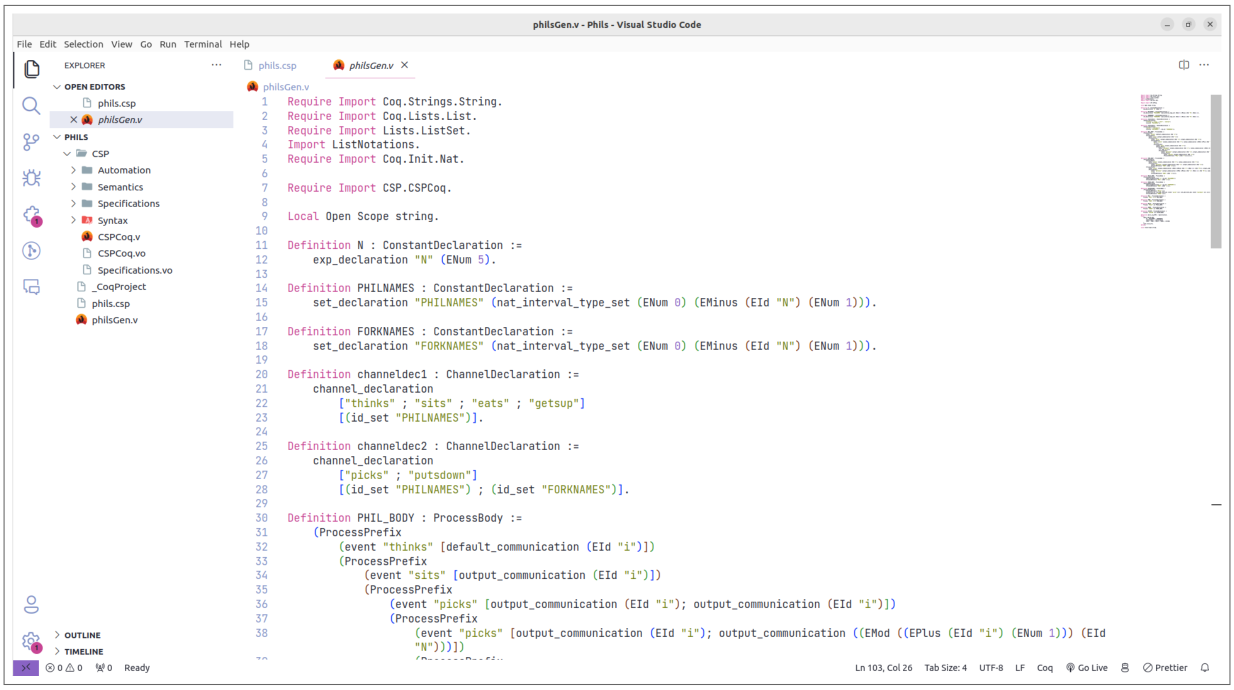


```
N = 5

PHILNAMES = {0..N-1}
FORKNAMES = {0..N-1}

channel thinks, sits, eats, getsup : PHILNAMES
channel picks, putsdown : PHILNAMES.FORKNAMES

PHIL(i) = thinks.i -> sits!i -> picks!i!i -> picks!i!((i+1)%N) ->
          eats!i -> putsdown!i!((i+1)%N) -> putsdown!i!i -> getsup!i -> PHIL(i)

FORK(i) = picks!i!i -> putsdown!i!i -> FORK(i)
         []  picks!((i-1)%N)!i -> putsdown!((i-1)%N)!i -> FORK(i)

PHILS = ||| i : PHILNAMES @ PHIL(i)
FORKS = ||| i : FORKNAMES @ FORK(i)

SYSTEM = PHILS [| {|picks, putsdown|} |] FORKS
```

Source: Current Author

To translate the file `phils.csp` to the corresponding CSP$_{Coq}$ representation, one should start the command palette, search for the "*CSPcoq: Translate*" command and press enter. The translation process starts, and a toast appears on the screen. If the translation is successful (i.e., the CSP$_M$ code does not contain syntactic errors), on the left panel, the user will see a new file called `philsGen.v`, in this case, along with the files of the CSP$_{Coq}$ theory. Now, the user can open the `philsGen.v`, and use the VsCoq extension to analyse desired properties of her CSP specification. In Figure 10, we can see the generated and copied `.v` files on the left panel. On the main panel, we have the file `philsGen.v` open. If the CSP$_M$ code is updated, the user can repeat the translation process and the `philsGen.v` file will be overwritten accordingly.

Figure 10 – Files produced for the dinning philosophers example.



```
Require Import Coq.Strings.String.
Require Import Coq.Lists.List.
Require Import Lists.ListSet.
Import ListNotations.
Require Import Coq.Init.Nat.

Require Import CSP.CSPCoq.

Local Open Scope string.

Definition N : ConstantDeclaration :=
    exp_declaration "N" (ENum 5).

Definition PHILNAMES : ConstantDeclaration :=
    set_declaration "PHILNAMES" (nat_interval_type_set (ENum 0) (EMinus (EId "N") (ENum 1))).

Definition FORKNAMES : ConstantDeclaration :=
    set_declaration "FORKNAMES" (nat_interval_type_set (ENum 0) (EMinus (EId "N") (ENum 1))).

Definition channeldec1 : ChannelDeclaration :=
    channel_declaration
        ["thinks" ; "sits" ; "eats" ; "getsup"]
        [(id_set "PHILNAMES")].

Definition channeldec2 : ChannelDeclaration :=
    channel_declaration
        ["picks" ; "putsdown"]
        [(id_set "PHILNAMES") ; (id_set "FORKNAMES")].

Definition PHIL_BODY : ProcessBody :=
    (ProcessPrefix
        (event "thinks" [default_communication (EId "i")])
        (ProcessPrefix
            (event "sits" [output_communication (EId "i")])
            (ProcessPrefix
                (event "picks" [output_communication (EId "i"); output_communication (EId "i")])
                (ProcessPrefix
                    (event "picks" [output_communication (EId "i"); output_communication ((EMod ((EPlus (EId "i") (ENum 1))) (EId
                    "N"))))])
```

Source: Current Author

4.3 GRAPHVIZ INTEGRATION

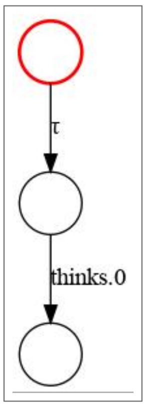
At this moment, the integration with Graphviz is via command line. First, one needs to invoke the function *generate_dot* (see Section 3.5.1). This needs to be done using some IDE for Coq. For instance, one could open the file `philsGen.v` in VSCode (with the VsCoq extension), and use the Coq keyword `Compute` to call the aforementioned function.

Provided an LTS (i.e., a set of *Transitions*), the function yields a textual (string) representation of the given labelled transition system in the DOT language. This string is sent to the default output stream of the IDE being used. Then, it is necessary to copy the contents of the string and save into a file. Finally, via command line, we can use Graphviz to generated a graphical representation as a `jpeg` file, for instance.

To illustrate, let us consider the part of the LTS obtained from a process reference to `PHIL(0)` – see Code 2.1. By executing the following command, Figure 11 is generated. Assume that the output of *generate_dot* has been save in the file `PhilsTest.gv`.

```
dot -Nlabel="" -Nshape=circle -Tjpeg PhilsTest.gv -o PhilsTest1.jpeg
```

Figure 11 – Graphical representation of LTSs.



Source: Current Author

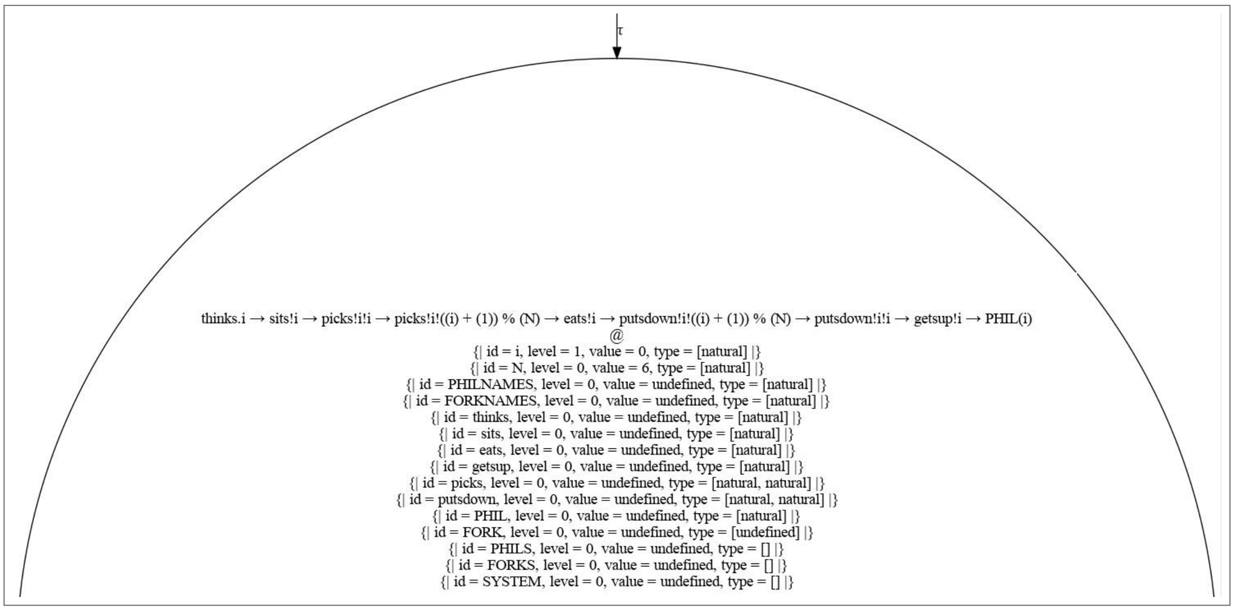
According to the SOS, from the associated initial state, we have a τ transition representing the unfolding of the process reference. Then, the first event to occur is *thinks.0*. This is precisely what is seen in Figure 11.

A distinguishing feature of our integration with Graphviz is that it is also possible to render the LTS showing the inner structure of nodes. This facilitates inspecting and debugging a CSP specification. A node is characterised by a *State*, which associates a *ProcessBody* with a *Context*. By executing the following command, on the same file (`PhilsTest.gv`), we generate the image with the internal representation of nodes (see Figure 12).

```
dot -Nshape=circle -Tjpeg PhilsTest.gv -o PhilsTest2.jpeg
```

The difference between the two commands is that, in the latter, we removed the argument `-Nlabel=""`, which hides the contents of nodes. In Figure 12, we highlight the state reached after performing the τ. Before the `@`, we see the process body of `PHIL(0)`. After the `@`, we see the associated context. Note that we have in the context a local variable (i.e., level = 1) `i`, whose value is equal to `0`. This is the variable introduced by the process reference.

Figure 12 – Graphical representation of LTSs showing the state information.



```
thinks.i → sits!i → picks!i!i → picks!i!((i) + (1)) % (N) → eats!i → putsdown!i!((i) + (1)) % (N) → putsdown!i!i → getsup!i → PHIL(i)
@
{| id = i, level = 1, value = 0, type = [natural] |}
{| id = N, level = 0, value = 6, type = [natural] |}
{| id = PHILNAMES, level = 0, value = undefined, type = [natural] |}
{| id = FORKNAMES, level = 0, value = undefined, type = [natural] |}
{| id = thinks, level = 0, value = undefined, type = [natural] |}
{| id = sits, level = 0, value = undefined, type = [natural] |}
{| id = eats, level = 0, value = undefined, type = [natural] |}
{| id = getsup, level = 0, value = undefined, type = [natural] |}
{| id = picks, level = 0, value = undefined, type = [natural, natural] |}
{| id = putsdown, level = 0, value = undefined, type = [natural, natural] |}
{| id = PHIL, level = 0, value = undefined, type = [natural] |}
{| id = FORK, level = 0, value = undefined, type = [undefined] |}
{| id = PHILS, level = 0, value = undefined, type = [] |}
{| id = FORKS, level = 0, value = undefined, type = [] |}
{| id = SYSTEM, level = 0, value = undefined, type = [] |}
```

Source: Current Author

5 CONCLUSION

This work contributes to the development of an updated theory for Communicating Sequential Processes within the Coq proof assistant. A broader range of CSP specifications can now be formalised in Coq thanks to the addition of composite channels, parametrised processes, and a richer set of CSP operators. These additions brought new challenges that motivated the definition of static and dynamic typing systems. A set of 39 well-formedness conditions are verified by proof when creating a CSP specification in Coq. The associated proof goals are automatically discharged by custom automation tactics created by us. Our Coq formalisation allows for reasoning about the structured operational semantics, labelled transition systems, traces refinement, and deadlock freedom of CSP specifications.

In addition to the enhancements made to the language CSP_{Coq}, this project has developed tools that intend to streamline the use and adoption of CSP_{Coq}. Specifications written in CSP_{M} are seamlessly translated into their corresponding Coq characterisation, and this is made available for users via an extension to VSCode, a popular IDE. We also integrate our implementation with Graphviz to enable the graphical visualisation of labelled transitions systems obtained from CSP specifications. In the following sections, we address related (Section 5.1) and future (Section 5.2) work.

5.1 RELATED WORK

First, let us compare our contribution with the preliminary work of Freitas (2020). There was almost no reuse of the previous theory with respect to the syntactic definitions, as detailed in Section 2.3. Our account of WFCs is also distinct from this previous work. Concerning the SOS, there was some reuse, lifting rules that are not greatly affected by the new typing system. Differently, regarding the definition of LTSs, and traces refinement, we basically lifted the previous definitions to the current development of CSP_{Coq}. Deadlock-freedom analysis is also an exclusive contribution of our work.

In the following sections, we provide more details about other tools that also enable formal verification of CSP specifications via proof assistants; namely, CSP-Prover (ISOBE; ROGGENBACH, 2005), HOL-CSPM (BALLENGHIEN; TAHA; WOLFF, 2023), and Isabelle/UTP (FOSTER et al., 2019). Then, we summarise our comparison, also taking into account tools that rely on model checkers; namely, FDR (GIBSON-ROBINSON et al., 2014), PAT (SUN et al., 2009), and ProB (LEUSCHEL; BUTLER, 2008). Our comparison focuses on the following criteria.

- Strategy: if it is based on model checking or proof development.
- Verification of WFCs: how WFCs are verified – algorithmically vs. by proof.
- SOS: whether it takes into account an SOS.
- Denotational: whether it takes into account a denotational semantics.
- LTS: whether it supports graphical visualisation of LTSs.
- Traces refinement: whether it supports traces-refinement analysis.
- Deadlock freedom: whether it supports deadlock-freedom analysis.

CSP-Prover, HOL-CSPM, and Isabelle/UTP are all based on Isabelle (NIPKOW; PAULSON; WENZEL, 2002), which is designed to assist in the formal proof of logical theorems. The Archive of Formal Proofs (AFP), a repository of proofs, theories, examples, and scientific development in Isabelle, serves as a repository of verified proofs that can be used and referenced in new verification projects. Contributions to the AFP are peer-reviewed, ensuring that proof meets high standards of correctness, readability, and relevance. At this moment, CSP-Prover is the only one that is not available at AFP, however, its use has been demonstrated to be promising by the community, and reported on different publications.

5.1.1 CSP-Prover

CSP-Prover is an interactive theorem prover designed specifically for refinement proofs within the CSP process algebra. The CSP syntax is deeply encoded. A denotational semantics approach was chosen due to the focus on the CSP Failures model. The implementation of the theory considers traces and stable-failures semantics. The authors argue that the encoding of these semantics is well-formed, since Isabelle only allows for consistent theories.

5.1.2 HOL-CSPM

The HOL-CSPM (BALLENGHIEN; TAHA; WOLFF, 2023) framework offers support for the specification, analysis, and verification of concurrent systems using CSP within a framework that is backed by Higher-Order Logic (HOL). Isabelle/HOL is a specific instantiation of Isabelle used for Higher-Order Logic, and it provides support for a broad range of logical constructs, including HOL-CSP and HOL-CSPM.

HOL-CSP version 2.0 (TAHA; YE; WOLFF, 2019) is part of the Isabelle ecosystem and provides a foundational framework and tools for modelling and reasoning about CSP. HOL-CSPM enables the specification of concurrent systems in CSP_M and leverages the formal verification environment provided by HOL-CSP. HOL-CSPM is based on a denotational semantics. If one needs to reason about an operational semantics, an alternative is to use HOL-CSP-OpSem (BALLENGHIEN; WOLFF, 2023), which is built on top of HOL-CSPM.

5.1.3 Isabelle/UTP

The Unifying Theories of Programming (UTP) is a fundamental approach to bridge the different programming paradigms and specification languages (HOARE; JIFENG, 1998). Isabelle/UTP implements UTP within Isabelle/HOL, which serves as a comprehensive framework for studying and formalising various languages and semantics, including those of CSP. Isabelle/UTP enables the seamless combination of theories to model complex systems. Its integration with Isabelle's type system and proof tactics enhances the ability to perform rigorous verification. In the latest versions of Isabelle/UTP, the CSP theory is available through the Circus (WOODCOCK; CAVALCANTI, 2002) family of notations and not as a separate theory.

5.1.4 Summary of comparison

In Table 10, we summarise the comparison of CSP_Coq with its related work. The investigated tools support different variations of CSP. While ProB and FDR targets CSP_M, PAT is tailored for CSP#. The other tools combine CSP with Isabelle's language or Coq's one (i.e., Gallina).

Table 10 – Comparison with related work.

Work	Known as	Language	Tool	Strategy	WFCs verif.	SOS	Denot.	LTS	Traces	Deadlock
(GIBSON-ROBINSON et al., 2014)	–	CSP$_{\mathrm{M}}$	FDR	model check.	algorithmically	yes	yes	yes	yes	yes
(SUN et al., 2009)	–	CSP#	PAT	model check.	algorithmically	yes	yes	yes	yes	yes
(LEUSCHEL; BUTLER, 2008)	–	CSP$_{\mathrm{M}}$	ProB	model check. constraint sol.	algorithmically	yes	no	yes	yes	yes
(ISOBE; ROGGENBACH, 2005)	CSP-Prover	CSP+ Isabelle	Isabelle	proof dev.	by proof	no	yes	no	yes	yes
(BALLENGHIEN; TAHA; WOLFF, 2023)	HOL-CSPM	CSP+ Isabelle	Isabelle	proof dev.	by proof	yes	yes	no	yes	yes
(FOSTER et al., 2019)	Isabelle/UTP	Circus+ Isabelle	Isabelle	proof dev.	by proof	no	yes	no	yes	yes
Our contribution	–	CSP$_{\mathrm{Coq}}$+ Gallina	Coq	proof dev.	by proof	yes	no	yes	yes	yes

Source: Current Author

On the one hand, some approaches allow for automatic verification of CSP specifications (FDR, PAT, and ProB), in general, using model checking techniques, which suffer from scalability issues. ProB differentiates itself for allowing automated verification by using constrains solvers, which render sound but potentially incomplete results. On the other hand, our solution, CSP-Prover, HOL-CSPM, Isabelle/UTP and $\mathrm{CSP}_{\mathrm{Coq}}$ verifies CSP specifications by means of proof development, which, in general terms, is a semiautomatic procedure, highly dependent on the user's experience, but that can scale better than model checking approaches.

Tools like FDR, PAT, and ProB verifies well-formedness conditions of CSP specifications in an algorithmic way: procedures are implemented to verify (statically and dynamically) whether the associated WFCs hold. Differently, the other alternatives presented in Table 10 perform such a verification by creating proofs scripts.

Reasoning about traces and deadlock-freedom is supported by all investigated works. The same cannot be said about the formalisation of SOS, and graphical visualisation of LTSs. Although this is supported by all tools based on model checking, regarding those using proof assistants, apart from $\mathrm{CSP}_{\mathrm{Coq}}$, only an extension of HOL-CSPM encodes the operational semantics of CSP. However, this extension does not provide graphical visualisation of LTSs. Regarding the CSP denotational semantics, this is addressed by most works, apart from ProB and our CSP characterisation in Coq.

Different tools have their strengths and weaknesses when it comes to the verification of CSP specifications. Isabelle's ecosystem has several theories that can be used to model and verify CSP specifications. By proposing an alternative based on Coq, we allow for the integration of this theory into other projects that are based on Coq, besides benefiting from Coq's type theory and proof mechanisms.

5.2 FUTURE WORK

Despite the various contributions of this work, we envisage many interesting opportunities worth pursuing in the future. In what follows, we comment on the main ones.

- **Further validate $\mathbf{CSP}_{\mathbf{Coq}}$.** In this work, we validated our theory for Communicating Sequential Processes taking into account on a small number of examples. Therefore, considering more examples is an important next step.

- **Enrich the syntax of CSP$_{\mathbf{Coq}}$**. Although we support a rich set of CSP operators, we are aware of some specifications that are still not allowed; for instance, we do not account for the renaming operator.

- **Optimise the static verification of WFCs**. As discussed in Chapter 3, the static verification of some WFCs is time consuming, compared to others. We should investigate ways of reducing the time required by these verification steps.

- **Extend the integration with Gallina**. At this moment, when creating natural and boolean expressions, one can invoke arbitrary functions of Gallina that yield natural and boolean values; this is a consequence of our shallow embedding. In the future, we could extend this integration by enriching our representation of expressions. For instance, we could provide support for lists and sets. This would have an impact on the semantics (i.e., the defined typing system).

- **Address other semantic models**. CSP different semantic models. At this moment, we focused on an operation semantics, and on the traces model. In the future, other semantic models should be considered too.

REFERENCES

ARIAS, E. J. G.; PIN, B.; JOUVELOT, P. jsCoq: Towards hybrid theorem proving interfaces. In: AUTEXIER, S.; QUARESMA, P. (Ed.). *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016*. [S.l.]: Open Publishing Association, 2017. (Electronic Proceedings in Theoretical Computer Science, v. 239), p. 15–27. ISSN 2075-2180.

BALLENGHIEN, B.; TAHA, S.; WOLFF, B. HOL-CSPM - architectural operators for HOL-CSP. *Archive of Formal Proofs*, December 2023. ISSN 2150-914x. <https://isa-afp.org/entries/HOL-CSPM.html>, Formal proof development.

BALLENGHIEN, B.; WOLFF, B. Operational semantics formally proven in HOL-CSP. *Archive of Formal Proofs*, December 2023. ISSN 2150-914x. <https://isa-afp.org/entries/HOL-CSP_OpSem.html>, Formal proof development.

BERTOT, Y.; HUET, G.; CASTÉRAN, P.; PAULIN-MOHRING, C. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. [S.l.]: Springer Berlin Heidelberg, 2010. (Texts in Theoretical Computer Science. An EATCS Series). ISBN 9783642058806.

COQUAND, T.; HUET, G. The calculus of constructions. *Information and Computation*, v. 76, n. 2, p. 95–120, 1988. ISSN 0890-5401.

CRISAFULLI, P.; TAHA, S.; WOLFF, B. Modeling and analysing cyber-physical systems in HOL-CSP. *Robotics Auton. Syst.*, v. 170, p. 104549, 2023.

FOSTER, S.; ZEYDA, F.; NEMOUCHI, Y.; RIBEIRO, P.; WOLFF, B. Isabelle/UTP: Mechanised theory engineering for Unifying Theories of Programming. *Archive of Formal Proofs*, February 2019. ISSN 2150-914x. <https://isa-afp.org/entries/UTP.html>, Formal proof development.

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.

FREITAS, C. A. da Silva Carvalho de. *A Theory for Communicating Sequential Processes in Coq*. 2020. Trabalho de Conclusão de Curso, Centro de Informática, Universidade Federal de Pernambuco.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. W. FDR3 — a modern refinement checker for CSP. In: ÁBRAHÁM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 187–201. ISBN 978-3-642-54862-8.

HOARE, C.; JIFENG, H. *Unifying Theories of Programming*. [S.l.]: Prentice Hall, 1998. (Prentice Hall series in computer science). ISBN 9780134587615.

HOARE, C. A. R. Communicating Sequential Processes. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 8, p. 666–677, aug 1978. ISSN 0001-0782.

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.

LEUSCHEL, M.; BUTLER, M. J. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, v. 10, n. 2, p. 185–203, 2008.

MILNER, R. *A Calculus of Communicating Systems*. Springer, 1980. (Lecture Notes in Computer Science, v. 92). ISBN 3-540-10235-3. Available at: <https://doi.org/10.1007/3-540-10235-3>.

NIPKOW, T.; PAULSON, L. C.; WENZEL, M. *Isabelle/HOL a Proof Assistant for Higher-Order Logic*. Berlin and New York: Springer, 2002.

OVERFLOW, S. *Stack Overflow Developer Survey 2023*. 2023. [Accessed: 2024-02-27]. Available at: <https://survey.stackoverflow.co/2023/>.

PARR, T. *The Definitive ANTLR 4 Reference*. 2nd. ed. [S.l.]: Pragmatic Bookshelf, 2013. ISBN 1934356999.

PIERCE, B. C.; AMORIM, A. A. de; CASINGHINO, C.; GABOARDI, M.; GREENBERG, M.; HRITCU, C.; SJÖBERG, V.; YORGEY, B. *Logical Foundations*. [S.l.]: Electronic textbook, 2018. (Software Foundations series, volume 1).

ROSCOE, A. W. *Understanding Concurrent Systems*. [S.l.]: Springer, 2010.

SCATTERGOOD, J. B. *The semantics and implementation of machine-readable CSP*. Phd Thesis (PhD Thesis) — University of Oxford, UK, 1998. Available at: <https://www.cs.ox.ac.uk/activities/concurrency/theses/>.

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

SHI, L.; LIU, Y.; SUN, J.; DONG, J. S.; CARVALHO, G. An analytical and experimental comparison of CSP extensions and tools. In: AOKI, T.; TAGUCHI, K. (Ed.). *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*. [S.l.]: Springer, 2012. (Lecture Notes in Computer Science, v. 7635), p. 381–397.

SUN, J.; LIU, Y.; DONG, J. S.; PANG, J. Pat: Towards flexible verification under fairness. In: . [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5643), p. 709–714.

TAHA, S.; YE, L.; WOLFF, B. HOL-CSP version 2.0. *Archive of Formal Proofs*, April 2019. ISSN 2150-914x. <https://isa-afp.org/entries/HOL-CSP.html>, Formal proof development.

WOODCOCK, J.; CAVALCANTI, A. The semantics of Circus. In: BERT, D.; BOWEN, J. P.; HENSON, M. C.; ROBINSON, K. (Ed.). *ZB 2002:Formal Specification and Development in Z and B*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 184–203. ISBN 978-3-540-45648-3.