Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

Recife

2020

Carlos Alberto da Silva Carvalho de Freitas

A Theory of Communicating Sequential Processes in Coq

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

Universidade Federal de Pernambuco

Centro de Informática

Bachelor in Computer Engineering

Supervisor: Gustavo Henrique Porto de Carvalho

Recife

2020

Acknowledgements

Abstract

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

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

Resumo

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

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

List of Figures

List of abbreviations and acronyms

CSP	Communicating Sequential Processes
FDR	Failures-Divergence Refinement
LTS	Labelled Transition System
SOS	Structured Operational Semantics

Contents

1 Introduction

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

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

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

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

1.1 Objectives

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

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

1.2 An overview of CSP_{Coq}

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

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

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

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

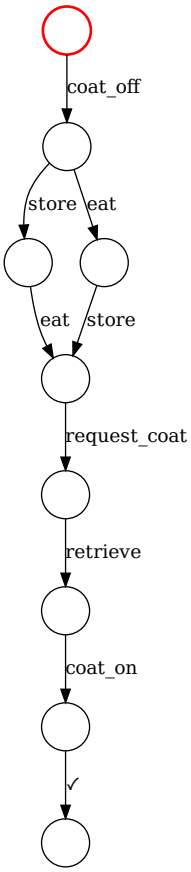
```
Definition example : specification.
Proof.
  solve_spec_ctx_rules (
    Build_Spec
    [ Channel {{"coat_off", "coat_on", "request_coat", "retrieve", "store", "eat"}} ]
    [ "SYSTEM" ::=
      "coat_off" -> "store" -> "request_coat" -> "retrieve" -> "coat_on" -> SKIP
      [| {{"coat_off", "request_coat",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

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

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

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

Figure 1 – The process LTS graph.



1.3 Main contributions

The main contributions of this work are the following:

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

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

1.4 Document structure

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

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

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

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

2 Background

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

2.1 Communicating sequential processes

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

Since the main goal of CSP is to provide a theory-driven framework for designing systems of interacting components an reasoning about them, we must introduce the concept of a component, or as we will be referencing it from now on, a *process*. Processes are self-contained entities that once combined they can describe a system, which is yet another larger process that may itself be combined as well with other processes. The way a process communicates with the environment is through its *interface*. The interface of a process is the set of all the events that the process has the potential to engage in. At last, an *event* represents the atomic part of the communication itself. It is the piece of information the processes rely on to interact with one another. A process can either participate actively or passively in a communication, depending on whether it performed or suffered the action. Events may be external, meaning they appear in the process interface; indicate termination, represented by the event $\checkmark$; or be internal, and therefore unknown for the environment, denoted by the event τ.

The most basic process one can define is *STOP*. Essentially, this process never interacts with the environment and its only purpose is to declare the end of an execution. In other words, it illustrates a deadlock: a state in which the process can not engage in any event or make any progress whatsoever. It could be used to describe a computer that

failed booting because one of its components is damaged, or a camera that can no longer take pictures due to storage space shortage.

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

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

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

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

CSP introduces two approaches for describing a parallel execution between processes: the *alphabetized parallel* and the *generalized parallel*. Let A be the interface of process P, and B the interface of process Q. An alphabetized parallel combination of these processes is described as $P \;{}_A\|_B\; Q$. Events in the intersection of A and B must be simultaneously engaged in by the processes P and Q. In other words, an event that appears in both process interfaces can only be communicated if the two processes are ready to perform this event. Any other event that does not match this criteria can be engaged in by its corresponding process independently. The semantics are similar for the generalized version of the parallel operator. The only change being its constructor, that takes the synchronization alphabet alone as the interface argument the processes must agree upon. Let C be the intersection of previously defined interfaces A and B. The generalized parallel between process P and Q is written as $P \underset{C}{\|} Q$.

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

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

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

Sometimes it is necessary to pass the control over execution from one process to another, and for that we use sequential composition. It means that the first process has reached a successful termination state and now the system is ready to behave as the second process in the composition. Parents can choose to let their children play only after completing their homework. That being the case, the process *CHILD* could be modeled as $HOMEWORK ; FUN$, where the process *HOMEWORK* is described as $choose\_subject \rightarrow study \rightarrow answer\_exercises \rightarrow SKIP$ and the process *FUN* as $build\_lego \rightarrow watch\_cartoons \rightarrow play\_videogame \rightarrow SKIP$. In this example, the process *FUN* can only be executed after the process *HOMEWORK* has successfully terminated.

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

2.1.1 Structured operational semantics

Since the early 1980s there have been three complementary approaches to understanding the semantics of CSP programs. These are algebra, where we set out laws that the syntax is assumed to satisfy, behavioral models such as traces that form the basis of refinement relations and other things, and operational models, which try to understand all the actions and decisions that process implementations can make as they proceed.

The operational semantics of CSP treats the CSP language itself as a (large!) LTS. It allows us to compute the initial events of any process, and what processes it might become after each such event. By selecting one of these actions and repeating the procedure, we can explore the state space of the process we started with. The operational semantics gives a one-state-at-a-time recipe for computing the transition system picture of any process. It is traditional to present operational semantics as a logical inference system: Plotkin's SOS, or Structured Operational Semantics style. A process has a given action if and only if that is deducible from the rules given.

2.1.2 Traces refinement

Imagine you are interacting with a CSP process. The most basic record you might make of what happens is to write down the trace of events that occur: the sequence of communications between you (the environment) and the process. In general, a trace might be finite or infinite: finite either because the observation was terminated or because the process and environment reach a point where they cannot agree on any event; infinite when the observation goes on for ever and infinitely many events are transacted. Traces are typical of the sort of behaviors we use to build models of CSP processes: they are clearly observable by the environment interacting with the process, and each of them is the record of single interaction with the process. We will see more details of the traces model in the next chapter, but we now introduce a fundamental way in which we can specify the correctness of a CSP process. Using traces(P) to denote P's finite traces we can write

and read this as Q *trace-refines* P. In other words, every trace of Q is a trace of P.

2.1.3 Machine-readable version of CSP

The main purpose of CSP is to describe communicating and interacting processes. But in order to make it useful in practice we have added quite a rich language of sub-process objects: as we saw in Chap.8, CSPM contains a functional programming language to describe and manipulate things like events and process parameters.

2.2 The Coq proof assistant

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming languages, the formalization of mathematics, and teaching.

2.2.1 Building proofs

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

2.2.2 The tactics language

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

2.3 QuickChick

QuickChick is a set of tools and techniques for combining randomized property-based testing with formal specification and proof in the Coq ecosystem.

3 A theory for CSP in Coq

3.1 Syntax

3.1.1 Abstract syntax

3.1.2 Concrete syntax

3.2 Structured operational semantics

3.3 Labelled transition systems

3.3.1 GraphViz integration

3.4 Traces refinement

3.4.1 QuickChick integration

4 Conclusions

4.1 Related work

4.2 Future work

Bibliography

CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2000. (ICFP '00), p. 268–279. ISBN 1581132026. Disponível em: <https://doi.org/10.1145/351240.351266>. Citado na página 12.

HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, ACM New York, NY, USA, v. 21, n. 8, p. 666–677, 1978. Citado na página 12.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733. Citado na página 9.