Model and Verify Cyber-Physical Systems CPS with Communicating Sequential Processes (CSP)

Luca Brodo[1]

Contents

Abstract: Communicating Sequential Processes (CSP) is a mathematical notation which is used to describe the interaction of concurrent systems. Even though firstly proposed in 1978, this formal language is still subject of intensive research in both industry and in the academia. This paper is aimed to give the reader an introduction to the vast world of CSP. In addition to define the syntax, we will also describe an example and we will compare this formal language to a common model used in computer science, the actor model.

[1] luca.brodo@hshl.de

1 Motivation

Historically, the advancements in software development have mainly relied upon the improvements in hardware, hence on faster processing units and on larger available memory. However, as pointed out by Hoare in [Ho85], running code that consumes 10 times more resources in a machine that is 10 times faster than before is not really an improvement.
Concurrency, while advantageous, has always been avoided by programmers. Concurrency, unlike parallelism, which implies the use of n processors for n tasks, relies upon one processor to run multiple tasks. With such an approach, the amount of resources remains unchanged, which means that programs requiring the same resource must wait; therefore causing numerous errors, like a linear increase in waiting users and amount of storage as the number of jobs increases, but more notably, the difficulty in verifying program correctness. The latter being the main reason why programmers have shied away from parallel programming, as errors that arise in this method are notoriously hard to track down. The complex nature of concurrency often made programmers resort to exhaustive search tactics to find bugs in their code, like rigorous testing, which is however not guaranteed to cover all scenarios. The use of Communicating Sequential Processes (CSP) allows programmers to mathematically guarantee programs to be free of the most common concurrency-related problems and to write logically-guaranteed correct programs, rather than exhaustively search for bugs in the code.
The aim of this paper is to explore the vast world of CSP to give the reader a thorough understanding of the topic.
To reach this objective, in the next section we will introduce them and give an overview of what they are able to do. In the next section, we will introduce some of the most used operators in CSP. In section 4 we will use some of these operators to describe a real-world system using CSP.
Subsequently, we will explore the differences between the actor model and CSP.
Finally, in the last section, we will discuss some of the most common tools used for the simulation and the verification of systems described with CSP.

2 Core Concepts

Communicating Sequential Processes (CSP) is a formal language used to describe interactions between concurrent systems and it is part of the mathematical theories of concurrency known as process algebras, or process calculi, based on messages passed through channels. [Ro97]
It has been introduced by Hoare in 1978 in his paper ”Communicating Sequential Processes” and since then it has been evolving substantially. CSP has been highly influential for the design of programming languages like GO or the occam programming language. [Ro97]
CPS allows to describe systems in terms of component processes acting independently that communicate based solely on message-passing. The way these processes are related and the way they interact with the environment are described using various process algebraic

operators, which allow the description of quite complex processes from few primitives. The "Sequential" part of the name might be misleading, since, in the modern definition, CSP allows processes to be both sequential and a parallel composition of more primitive processes. In CSP there are two types of primitives: events and processes. Events represent communication or interactions and they are assumed to be instantaneous. Some examples for an event could be for e.g. *on, off* or *player.MoveRight*. Processes, on the other hand, represent fundamental behaviors, like for e.g. *STOP* (the process that communicates nothing, also called deadlock) or *SKIP* (which represents successful termination).
In CSP events are observed when modelling processes. To better understand the difference, let's take as example Fig. n 1. This figure depicts a model in which a device is in the middle of a 3x3 board and it is capable of moving in 4 directions (up, down, right, left) by one grid at the time.

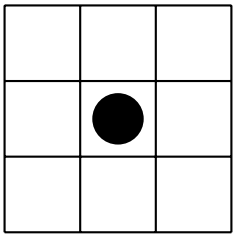


Fig. 1: 3x3 Grid with a device inside

The set of events which are considered relevant for a particular description of an object is called its *alphabet*. [Ho85]
The alphabet of such model is :

$$\alpha Board\{up, down, right, left\}$$

The most simple constructor to model a behavior is :

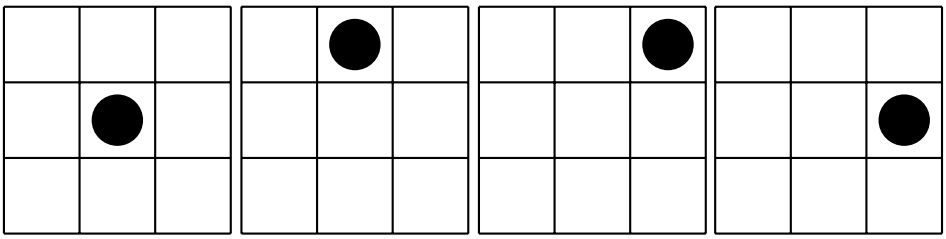
$$a- > P$$

With a being an event and P being a process, it means that a is a prefix of P. An example of a behavior which can be modelled in this scenario is

$$up- > right- > down- > STOP$$

and it can be seen in Fig n. 2.

The aforementioned example, even though particularly trivial, expresses perfectly one of the strength of CPS: its simplicity. CSP are notorious to be easy to understand and to apply, but are yet sufficiently expressive to enable reasoning about deadlock and livelocks. [Ro15]



(a) Initial state (b) up (c) right (d) down

Fig. 2: Simulation of the environment

3 Informal Definition

As we delineated in the previous sections, CSP allows to describe a system as processes that operate independently and that can communicate with each other solely relying on massage-passing. Such relationships and also the communication between each process and the environment is described by various *process algebraic operators*. Thanks to those operators, quite complex descriptions can be achieved starting from a few primitive elements. In the previous section we already defined the primitives provided by CPS in its process algebra, namely process and event. For convenience, the definition is repeated below:
Events represent communication or interactions and they are assumed to be instantaneous. Some examples for an event could be for e.g. *on, off* or *player.MoveRight*.
Processes, on the other hand, represent fundamental behaviors, like for e.g. *STOP* (the process that communicates nothing, also called deadlock) or *SKIP* (which represents successful termination). Moreover, the word process also stands for the behaviour pattern of an object, i.e. the limited set of events selected as its alphabet. [BHR84]
As for the operators, CPS includes a wide range of it. The first one, and the one that we introduced before, is the **prefix**.
Let x be an event and let P be a process. Then

$$(x \rightarrow P)$$

pronounced "x then P ", implies that an object, or system, after finishing an event x will behave as described by process P.
The example described by Fig n.1 shows a behavior described by prefixes that eventually terminates.
Even though easy to grasp and quite powerful, it would be tedious to describe the whole lifetime of bigger systems using only self-terminating methods with prefixes.[BHR84]
Therefore, CPS allows the description of systems using also recursion. For example, the behavior of a clock can be described as

$$CLOCK = tick- > CLOCK$$

to symbolize that the clock will not finish eventually, but rather will keep working indefinitely. In fact, if we try to explicit the recursion, as shown before, the process is will go on indefinitely:

$$CLOCK = tick- > CLOCK$$

$$CLOCK = tick- > (tick- > CLOCK)$$

$$CLOCK = tick- > (tick- > (tick- > CLOCK))$$

$$CLOCK = tick- > (tick- > (tick- > (tick- > CLOCK)))$$

With recursion and prefix it is only possible to define objects with only a sing stream of behavior. To allow objects to have their behaviour influenced by the environment, the operator **choice** has been introduced. [BHR84]
If x and y are distinct events

$$(x \rightarrow P|y \rightarrow Q)$$

describes an object which initially engages in either of the events x or y and afterwards will behave as either process P or Q. The bar | should be pronounced "choice": "x then P choice y then Q ". [BHR84]
In this case the choice is deterministic as the events are distinct, but there might be occasions where the events are the same:

$$(x \rightarrow P|x \rightarrow Q)$$

We refer to these situations as non-deterministic. The non-determinism comes from the fact that, as the event x is reached, there is no precise why to decide which process to undergo next.

CPS offers two operators to describe concurrency. If P and Q are processes with the same alphabet, we introduce the **concurrency** operator:

$$P||Q$$

to denote the process which behaves like the system composed of processes P and Q. This operator is used to combine interacting processes with differing alphabets into systems exhibiting concurrent activity, but without introducing non-determinism. However, sometimes is beneficial to join processes processes with the same alphabet to operate concurrently without directly interacting or synchronising with each other In this case, each action of the system is an action of exactly one of the processes and if one of the processes cannot engage in the action, then it must have been the other one; but if both processes could have engaged in the same action, the choice between them is non-deterministic. [BHR84]
This form of combination is defined by the **interleaving** operator as:

$$P|||Q$$

pronounced as P "interleave" Q.
Finally, the communication between processes is achieved through **channels** and **messages**.

$$channel(c.v) = c$$

$$message(c.v) = v$$

Messages are exchanged through channels and channels are used only in one direction and between only two processes.A channel which is used only for output by a process will be called an output channel of that process; and one used only for input will be called an input channel. [BHR84]
Let v be a message, a process which first outputs v on the channel c and then behaves like process P is defined as:

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

The only event engaged by this process is the communication event $c.v$.

A process which is initially prepared to input any value x communicable on the channel c, and then behave like P(x), is defined

$$(c?x \rightarrow P(x))$$

The way the communication is achieved in CPS is not dissimilar to the one defined for Timed Automata. [OD08] As a matter of fact, Timed Automata use channel as well and the semantic is not for off. The operator "!" is used to **output** on a channel c, while the operator "?" is used for **input** on the same channel.

4 Case Study

One of the archetypal CSP examples is an abstract representation of a vending machine that sells only chocolate and can interact with only one person at the time.
The first step to do when describing a system with CPS is to define its alphabet. In this case, we know that the machine accepts both card and coins and gives out either chocolate or a toffee. Therefore the alphabet for this example will be:

$$\alpha VENDINGMACHINE = \{coin, card, choc, toffee\}$$

We can describe the simplest form of interaction with this machine using the prefix and choice operators:

$$VENDINGMACHINE = (coin|card) \rightarrow (choc \rightarrow STOP|toffee \rightarrow STOP)$$

This description of the system is not wrong and defines perfectly the vending machine, however we can improve it by adding a customer. The customer can only insert a coin or a card and choose between a toffee or a chocolate.

$$\alpha CUST = \{coin, card, choc, toffee\}$$

To enable the customer to communicate with the vending machine, we need to modify it a bit, starting with its alphabet:

$$\alpha VENDINGMACHINE = \{coin, card, choose\_c\_t, choc, toffee\}$$

We introduced a new event, namely "choose_c_t" which defines the moment when the machine gives the customer the possibility to choose between chocolate and toffee.
The vending machine can now be expressed as:

$$OUT\_CHOCOLATE = choc? \rightarrow choc! \rightarrow VENDINGMACHINE$$

$$OUT\_TOFFEE = toffee? \rightarrow toffee! \rightarrow VENDINGMACHINE$$

$$VENDINGMACHINE = (coin?|card?) \rightarrow choose\_c\_t! \rightarrow (OUT\_CHOCOLATE|OUT\_TOFFEE)$$

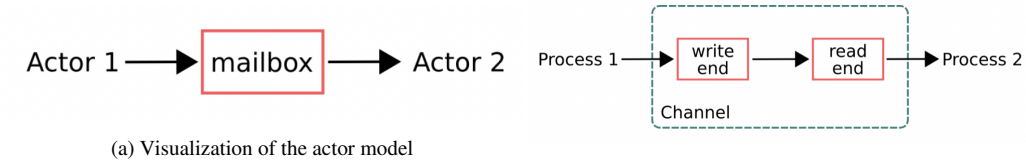
We also used recursion here to signify that the machine will never stop selling sweets.
According to this new description of the system, we can also define the behavior of the customer as :

$$CUST = (coin!|card!) \rightarrow choose\_c\_t? \rightarrow (choc!|toffee!) \rightarrow (choc?|toffee?) -> CUST$$

Now, in this augmented version, the customer is able to communicate with the vending machine using simple channels. Since we know that in CPS the communication with channels, we know that this behavior is deterministic. As a matter of fact, the machine can sell the chocolate only when the customer makes his choice, since it's expecting either *choc* or *toffee* from the customer as input. In addition, the customer can make the choice only when the machine offers him the possibility.
The example, even though trivial, is able to express amazingly the main characteristics of CPS and shows also how powerful they are. With only three of the many operators available, we were able to describe a real-world system composed of the interaction between two different sub-systems.

5 Comparison With The Actor Model

The actor model in computer science is a mathematical model of concurrent computation that treats actor as the universal primitive of concurrent computation.[CHS14]
The actor model is based on the idea of actors reacting to the messages they receive. The actor model and CSP are isomorphic, or equivalent, as they are both based on the concept of message passing. Thanks to this property, they both allow to solve the major problems related to shared memory. Programs realized with the concept of shared memory are realized based on primitives like mutexes and locks, which can create various issues. First of all,



(a) Visualization of the actor model

(b) Visualization of CSP

Fig. 3: Differences between CSP and the actor model [Ni99]

it's not uncommon for such programs to crash due to corrupted memory in critical region. In addition, is very difficult to track down the bug and to reason about it. Furthermore, nowadays it is not even clear where this shared memory is actually located, which makes recovering from failure even harder.

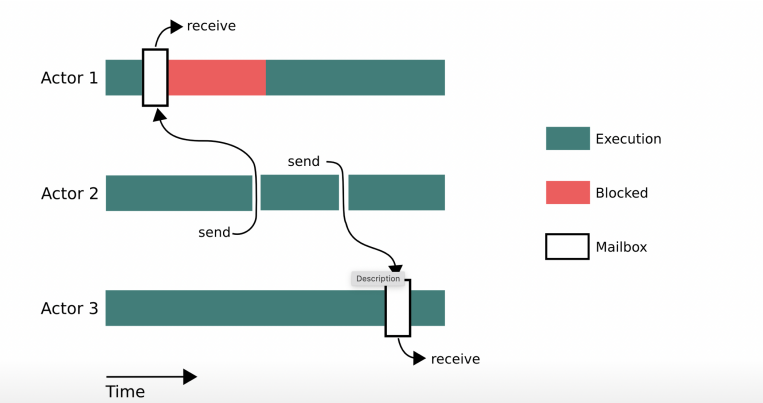
This is not the case for the Actor Model and CSP, since being based on message passing implies no shared resources between processes. The way these messages are passed, however, is the first instance in which those two models start to differ. In the actor model, actors can receive messages from any sender; CSP processes must name the channel over which they are sending (or receiving) a message.[Cl81]

This difference is shown in Fig n.3. In this picture, we can see that, to insure a non-blocking communication, every actor stores the received messages in a mailbox, while in the case of CSP there is a channel in between with a writing and a receiving end.

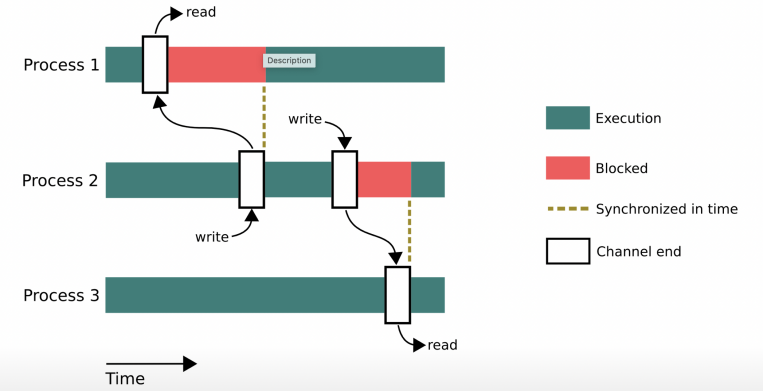
Moreover, the actor model requires fairness - no message can be delayed indefinitely. CSP message passing implies a rendezvous between the processes involved in the communication, i.e. the sender cannot transmit a message until the receiver is ready to accept it. In contrast, message-passing in actor systems is asynchronous, i.e. message transmission and reception do not have to happen at the same time.[Cl81] Such behavior is perfectly described in Fig n. 4.

In Fig. n. 4a, we can see how Actor 2 is able to send messages to both Actor 1 and Actor 3 without being blocked. Moreover, as it sends the message to the mailbox of Actor3, actor 3 is not stopped and continues to operate. The only actor being blocked is Actor 1, and that's because it needs to wait for Actor 2 to send the message before continuing.

Fig. n. 4b, on the other hand, shows an example of communication between processes described by CPS. Similar to Actor 1, Process 1 is blocked until Process 2 writes something on the channel and only then the two processes are synchronized. Differently from Actor 2, however, Process 2 is blocked when trying to write something on the channel with Processes 3. That's because the communication between the two processes is achievable only when both are ready, therefore Processes 2 is blocked and the two processes are synchronized when Process 3 is ready to read from the channel.



(a) Communication in the actor model



(b) Communication in CSP

Fig. 4: Differences in the communication methods between CSP and the actor model [Ni99]

6 Supported Tools

Over the years a lot of tools has been developed to model, simulate, check and refine systems described by CPS. One of the most known piece of software is Failures/Divergence Refinement (FDR) developed by the University of Oxford. FDR has been undergoing different release, namely FDR, FDR2, FDR3 and the last one FDR4, which was published in February of 2019. It is mainly used to check the determinism of a state machine and to check security properties. [Fa12]
Similarly to FDT, the Process Analysis Toolkit (PAT) is also a CSP analysis tool able to perform refinement checking and model checking [Su09]. It has been developed in the School of Computing at the National University of Singapore in 2009. In addition to the model checking, PAT also provides the users with the tools to simulate CSP and

Timed-CSP processes. The PAT process language extends CSP with support for mutable shared variables, asynchronous message passing, and a variety of fairness and quantitative time related process constructs such as deadline and waituntil. [Su09]
Lastly, Visualnets is a relatively simple tool which is able to simulate systems described with CSP and also offers an animated visualization of them. In addition to CSP, this tools supports also timed CSP.
Finally, thanks to the flexibility of CSP, there are also available libraries for the most common programming languages that enables the description of systems using CSP.
For instance, PyCSP is a Python library that implements core CSP primitives using standard python modules.[BVA07] Moreover, JCSP and CTJ are similar libraries developed for Java. [SHW00]

7 Conclusion

CSP is a very flexible and powerful powerful tool and allows to describe complex systems using only simple primitives. This formal language has been particularly influential for the development of many other tools and programming languages and it is considered a stepping stone for the verification and validation of concurrent systems. CSP revolvs around the idea of communication through message passing and, thanks to this concept, it is able to solve many problems related to shared memory. In other words, while the validation of systems based on shared memory is incredibly complex, the concept of communication through messages solves many of the problem with shared memory, hence allowing the engineers to track down errors in a significantly easier way.

Bibliography

[BHR84] Brookes, S. D.; Hoare, C. A. R.; Roscoe, A. W.: A Theory of Communicating Sequential Processes. J. ACM, 31(3):560–599, June 1984.

[BVA07] Bjørndalen, John; Vinter, Brian; Anshus, Otto: PyCSP - Communicating Sequential Processes for Python. volume 65, pp. 229–248, 01 2007.

[CHS14] Charousset, Dominik; Hiesgen, Raphael; Schmidt, Thomas: CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications. 2014.

[Cl81] Clinger, William D: Foundations of Actor Semantics. Technical report, USA, 1981.

[Fa12] : Failures-divergence Refinement Fdr2 User Manual. 2012.

[Ho85] Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall, Inc., USA, 1985.

[Ni99] Arild Nilsen, Communicating Sequential Processes (CSP).

[OD08] Olderog, Ernst-Rüdiger; Dierks, Henning: Real-Time Systems: Formal Specification and Automatic Verification. 01 2008.

[Ro97] Roscoe, A. W.: The Theory and Practice of Concurrency. Prentice Hall PTR, USA, 1997.

[Ro15] Roscoe, A.: The Expressiveness of CSP With Priority. Electronic Notes in Theoretical Computer Science, 319:387–401, 12 2015.

[SHW00] Schaller, Nan; Hilderink, Gerald; Welch, Peter: Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. 07 2000.

[Su09] Sun, Jun; Liu, Yang; Dong, Jin Song; Pang, Jun: PAT: Towards Flexible Verification under Fairness. In (Bouajjani, Ahmed; Maler, Oded, eds): Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 709–714, 2009.

8 Affidavit

I Brodo Luca herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance. I agree that my work may be checked by a plagiarism checker.