Communicating Sequential Processes - Theory, Implementation And Application

Syed Muhammad Saim[1]

Contents

Abstract: This paper elucidates fundamental concepts of communicating sequential processes (CSP) based on mathematical notations and self-explanatory diagrams. In addition, it talks about implementation of CSP with the aid of relevant tools and python programming (PyCSP). The paper concludes after discussing application of communicating sequential processes (CSP) in the fields of embedded system, automated driving, and neural network systems. Hence, the paper provokes to pursue further research and assists to apply the concepts in real world.

[1] syed-muhammad.saim@stud.hshl.de

1 Motivation

In the 1985, Tony Hoare published the article and explained communicating sequential processes in a simple theory. Communicating Sequential Processes is a branch of computer science that describes patterns of interaction in concurrent system. It is likely a formal language.

Initially, there is a description of how typical sequential programming operators can be integrated into the architecture of communicating sequential processes. Experienced programmers may be shocked to hear that these operators have the same exquisite algebraic characteristics as operators from well-known mathematical theories, and that sequential programs can be established to meet their specifications in the same fashion that concurrent programs do. The goal of this study is to create a mathematical domain that can define the semantics of communicative processes in the same way as the domain of partial functions can for sequential and deterministic programming languages. The domain has been kept as basic as possible, with the necessary operators over objects in the domain having elegant and intuitively valid features. [BHR84]

In addition, the application of automated driving system has been illustrated at first. The driver may be unable to take adequate control of the autonomous vehicle if they are not paying enough attention to the surrounding environment and driving circumstances. This research can uncover trends in a driver's state transition that result in the driver losing control of the automated vehicle. The ADS seeks to impact the driver's state during autonomous driving when appropriate, as the driver's state will influence how he or she reacts in an emergency. HAVEit (Highly Automated Vehicles for Intelligent Transportation) explored how the ADS can maintain the driver in the control loop of an automated vehicle in the event that intervention is needed. Consequently, an autonomous driving system (ADS) should be constructed in such a way that the driver is involved at the right times. It lays out the exact circumstances in which a driver can take control of an automated vehicle. HAVEit (Highly Automated Cars for Intelligent Transportation) looked at and validated the architecture of automated vehicles, including the driver-to-ADS interface. [Ki15a]

Moreover, this work provides an experimental system that uses a tablet to control a remote-controlled (RC) car in order to validate our suggested unique method for embedded system development. This study provides a new way for implementing an embedded system that uses communicating sequential processes (CSPs). The new XMOS, which was built using the latest LSI technology, allows the CSP principle to be implemented in an embedded system. [Ch18]

Lastly, the paper shows an example that illustrates the implementation of communication sequential process with Python programming language. The example I have shown in this paper is simply for the Hello World program to understand fundamental concept of PyCSP. [KO88]

2 Theoretical Background

2.1 Processes

Let x represent an event and let P represent a process. Then (x → P) (pronounced “x then P”) depicts an object that interacts in the event x before acting entirely as P says. The process (x → P) is intended to have the same alphabet as P, so this notation must not be used unless x is in that alphabet; more formally, $\alpha\ \alpha$ (x → P) = αP provided x αP [AJ05]

2.2 Concurrency

The notation P || Q is used when P and Q are processes with the same alphabet. [AJ05]

2.3 Nondeterminism

If P and Q are processes, we can use the notation P u Q (P or Q) to designate a process that behaves either like P or like Q , with the choice determined randomly and without awareness of the external environment’s control. The operands’ alphabets are presumed to be the same. α(P u Q) = αP = αQ [AJ05]

2.4 Communication

Set of all messages that P can communicate on channel c are known as αc(P) = v | c.v αP It has also been determine that functions which extract channel and message components of a communication channel (c.v) = c, message(c.v) = v [AJ05]

2.5 Sequential Processes

A sequential process is determined as one which has in its alphabet; and naturally this can only be the last event in which it engages. We stipulate that cannot be an alternative in the choice construct (x : B → P(x)) is invalid if B SKIPA is defined as a process which does nothing but terminate successfully SKIPA = A As usual, we shall frequently omit the subscript alphabet. [AJ05]

3 Application Of CSP Upon Automated Driving System

```
-- Driver's Behavior
SProcessing = (env?info -> com1!ok -> SProcessing)
                    [] (env?info -> sens!noperc ->
                    ((sens!reperc -> com1!ok -> SProcessing)
                    [] (NoPerceiving)))
NoPerceiving = sens!nores -> NoPerceiving
```

Figure 4. Driver's description about Sensor Processing process [Ki15b]

Part of the description of the CSP model used to depict driver processes is shown in Fig. 4. The driver's status is set to "Normal Perceivingör "No Perceivingïn Fig. 4 once data from the environment is obtained. It moves from "Normal Perceiving"to "Perceptionïf the state is "Normal Perceiving."The driver's status is conveyed to the ADS via the channel ßensïf the state is "No Perceiving."The ADS then seeks to reactivate the driver's participation in the driving process. The processes of the ADS are depicted in Figure 5. After obtaining the driver's status or behavior, those steps kick in. The ADS decides whether or not to take the motorist to an MRS based on his or her condition.When the driver's condition is "No Perceiving,"for example, the ADS tries to incorporate the driver into the driving process. If the ADS is unable to sense the environment after the effort, he or she is referred to the MRS. FDR3 features a feature that allows you to draw diagrams that depict the state changes of each operation. [Ki15b].

```
-- Automated Driving System
SensAuto = sens?noperc -> ((sens?reperc -> AutomatedDriving)
            [](sens?nores -> MinimumRiskState))
PercAuto = perc?drowsy -> ((perc?attentive -> AutomatedDriving)
            [](perc?noresR -> MinimumRiskState))
            [] perc?distracted -> ((perc?attentive -> AutomatedDriving)
            [](perc?noresR -> MinimumRiskState))
ResAuto = res?timedelay -> MinimumRiskState
FailureAD = adfail -> Failure
Failure = (failurerecovery -> AutomatedDriving)
            [] (err -> STOP)
MinimumRiskState = (mrsfail -> Failure) [] (mrs -> MinimumRiskState)
AutomatedDriving = SensAuto [] PercAuto [] ResAuto [] FailureAD
```

Figure 5. Automated driving system's description [Ki15b]

FDR3 was then used to check the CSP model for deadlock freedom. An assertion statement was added to the model to allow this. The model that resulted was free of deadlocks. This means that the ADS collaborates with the driver to keep the transportation system secure without coming to a complete stop. The failure state does not need to stop all processes at this point to verify if other concurrent processes are still running. The model is updated after model verification so that the failure status is STOP. The model verification in this situation reveals that the processes are no longer deadlock-free, as they stop at the failure state.[Ki15b].

4 Remote-Controlled Car using CSP

In this investigation, there are two major agents in the RC car controlled by a tablet. The first is the RC car, and the second is the tablet. The RC car adjusts its direction or speed by rotating or moving the tablet. These commands are sent from the tablet to the RC car via message-passing protocols. Sub-agents are divided into each main agent. Sub-agents in the RC car include a receiver for receiving commands from the tablet, a steering wheel for changing front-wheel direction, a main motor for rotating the driving wheels forward or backward, and sensors for detecting impediments. Figure 6 depicts the relationship between an agent and the hardware component that the agent controls. [MO12]

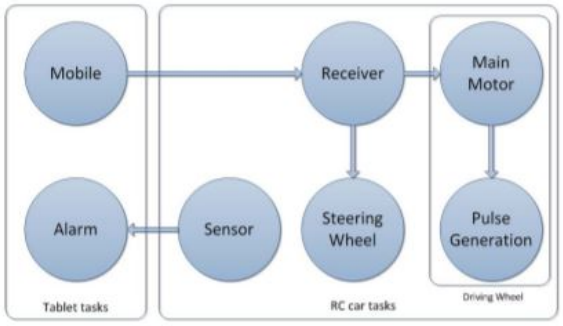

Figure 6. Communicating Sequential Process of RC Car [MO12]

- Mobile Process This process acts as the tablet's primary controller. Through a communication channel, the process sends a command to the RC car's Receiver process to change the speed and direction. The CSP model is described as follows: Mobile = in!x → Mobile, where in!x is a sending event x through channel in. [MO12]
- Receiver Process The commands sent from the tablet are managed by this process. This process sends an instruction to the Main Motor or the Steering Wheel processes after receiving a command. Receiver = in?x → (ch1!x|ch2!x) → Receiver, where in?x is a receiver event x through channel in. [MO12]
- Steering Wheel Process The direction of the RC car's steering wheel is controlled by this method. The Receiver process sends an instruction to this process, which controls the angle of the RC car's front wheel. SteeringW heel = ch2?x → (right|lef t) → SteeringW heel. [MO12]
- Main Motor Process Along with the Pulse Generation process, this process controls the RC car's driving wheels. The process receives a command from the Receiver process and instructs the Pulse Generation process to generate pulses for forward, backward, or stop motion. MainMotor = ch1?x → ch3!p → go|back|stop → MainMotor [MO12]

- Pulse Generation Process This process generates pulses and sends them to the driving motor in response to a request from the Main Motor process.. PulseGeneration = ch3?p → genP → PulseGeneration. [MO12]
- Sensor Process The Alarm process receives an alarm from one of the sensors and forwards it to this process. Sensor = analysis → ch4!s → Sensor. [MO12]
- Alarm Process The tablet emits alarm noises as a result of this operation. Alarm = ch4?s → ring → Alarm. [MO12]
- RC Car Process The complete process of the RC car, which is portrayed as a principal agent in the agent system concept, is expressed in this procedure. This process's sub-processes are all running at the same time. System = Mobile || Receiver || Handle || MainMotor || PulseGeneration || Sensor || Alarm where "||" denotes parallel processing in a CSP model. [MO12]

5 Implementation of Python Communication Sequential Process (PyCSP)

PyCSP is a Python library that can be used to create CSP-based concurrent applications. PyCSP was first released in 2007, and it has subsequently been updated in 2009 and 2012 to incorporate a distributed channel design. PyCSP currently comes in two different flavors: parallel and greenlets. Both implementations share a same API, making switching between them simple: compositional pieces can be written once and used in both implementations. When moving from parallel to greenlets, the PyCSP application may behave differently since processes are scheduled in a different order and are distributed less evenly. When all other processes are waiting to be scheduled, hidden latencies may become more apparent. It was proposed that several implementations share a single PyCSP API. For fine-grained CSP networks running on a single core, use pycsp.greenlets, and for coarse-grained CSP networks running on multi-core, many-core, and distributed architectures, use pycsp.parallel. [FVB10] The two implementations:

- pycsp.greenlets - Instead of threads, co-routines are used in this approach. Greenlets is a Python package that implements a simple co-routine implementation. It enables the creation of millions of CSP processes within a single CSP network. This version is best suited for single-core architectures since it delivers the quickest communication while excluding parallelism. [FVB10]
- pycsp.parallel- All channels have network connectivity and can communicate with both remote and local processes. For PyCSP programs, this is the default implementation. This version currently has four separate process types. [FVB10]
  - @process- As an OS thread, a CSP process is implemented. Thread-locking mechanisms handle the internal synchronization. This is best suited for programs

that spend the majority of their time in external procedures that release the Python Global Interpreter Lock1. [FVB10]

- @multiprocess- A CSP process that has been turned into an OS process. It's based on the multiprocessing module, which is available in Python 2.6 and later. The Global Interpreter Lock has no effect on this implementation, but it does have some constraints on a Windows OS, mostly due to the lack of a Windows fork() call. [FVB10]
- @sshprocess- A CSP process that runs on a remote host as an OS process. The procedure is run utilizing the SSH protocol, which necessitates the use of a set of additional arguments to start it. [FVB10]
- @clusterprocess- A CSP process that runs on a remote host identified in a node file as an OS process. The clusterprocess chooses the remote host depending on a distribution strategy provided to it. [FVB10]

Based on these commands, A simple Hello World program is depicted in the figure below.

```
@process
def HelloWorld(register):
    req_chan = Channel()
    cin = IN(req_chan)
    register(('/hello.html', OUT(req_chan)))
    while True:
        (req_str, cout) = cin()
        cout("Hello World")
```

Figure 7. HelloWorld Example of PyCSP [FVB10]

6 Conclusion

The paper discusses initially with a basic background of Communicational Sequenctial Processes. Furthermore, it applied the concepts to two of the projects. Finally, it showed the implementation of Communiaction Sequential Processes into Python programming language. This work builds the ground for those who are pursuing in understanding Communicational Sequential Processes and hence gives a concrete direcition.

Bibliography

[AJ05] Abdallah, Ali E; Jones, Cliff B: Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers, volume 3525. Springer Science & Business Media, 2005.

[BHR84] Brookes, Stephen D; Hoare, Charles AR; Roscoe, Andrew W: A theory of communicating sequential processes. Journal of the ACM (JACM), 31(3):560–599, 1984.

[Ch18] Chaturvedi, Kanishk; Matheus, Andreas; Nguyen, Son H; Kolbe, Thomas H: Securing spatial data infrastructures in the context of smart cities. In: 2018 International Conference on Cyberworlds (CW). IEEE, pp. 403–408, 2018.

[FVB10] Friborg, Rune; Vinter, Brian; Bjørndalen, John: PyCSP - Controlled concurrency. IJIPM, 1:40–49, 10 2010.

[Ki15a] Kinoshita, Satoko; Yun, Sunkil; Kitamura, Noriyasu; Nishimura, Hidekazu: Analysis of a driver and automated driving system interaction using a communicating sequential process. In: 2015 IEEE International Symposium on Systems Engineering (ISSE). IEEE, pp. 272–277, 2015.

[Ki15b] Kinoshita, Satoko; Yun, Sunkil; Kitamura, Noriyasu; Nishimura, Hidekazu: Analysis of a driver and automated driving system interaction using a communicating sequential process. In: 2015 IEEE International Symposium on Systems Engineering (ISSE). pp. 272–277, 2015.

[KO88] Koikkalainen, P; Oja, E: Proc. ICNN'88, Int. Conf. on Neural Networks. 1988.

[MO12] Mizutani, Ryo; Ohmori, Kenji: A Design and Implementation Method for Embedded Systems Using Communicating Sequential Processes with an Event-Driven and Multi-Thread Processor. In: 2012 International Conference on Cyberworlds. pp. 221–225, 2012.