FRI-ONLINE-1-CCT1-01

IMPLEMENTATION OF THE *CSP* SEMANTICS OF INTER-PROCESS COMMUNICATIONS USING THE *C++11* STANDARD LIBRARY

Assoc. Prof. Milen Loukantchevsky, PhD, IEEE Member, ACM Member
Department of Computer Systems & Technologies,
University of Ruse "Angel Kanchev"
Phone: 0877 303 850
E-mail: mil@ieee.org

Abstract: *The CSP semantics of inter-process communications suggest the simplest possible kind of channels – unidirectional, point-to-point <1:1>, unbuffered, with direct naming. Such channels do not actually imply flow control but require a special kind of bilateral synchronization known as "rendezvous".*

Whiles std::thread could be regarded near close to CSP::process semantics, in C++ standard library there is not type even distant from CSP::channel. Hence the main goal set: proposal of suitable implementation of the CSP semantics of inter-process communications with C++11 standard library means in seek of higher degree of structuring of communications.

As result a compact C++ template library csp is developed which encapsulates the means of messaging and synchronization between objects of type std::thread. The csp namespace defined includes csp::chan, csp::mux, csp::sink and csp::forks classes. The csp::chan class is strong implementation of CSP channel semantics, with two methods csp::chan::send() and csp::chan::recv() corresponding to CSP communications commands <!> and <?>. The alternative command for non-deterministic selection is implemented by the method csp::mux::recv() using multiple wait on std::condition_variable and randomized choice between true guards.

Keywords: *CSP, C++11, Multithreading, Concurrency, Non-deterministic Message Passing*

INTRODUCTION

Leading methodological principle of science is to direct investigations to *the essential contradiction* of the subject domain. At the area of computer systems development, this is the contradiction between *the explicit parallelism* of current computer architectures and *the sequential thinking* (Loukantchevsky, M., 2013).

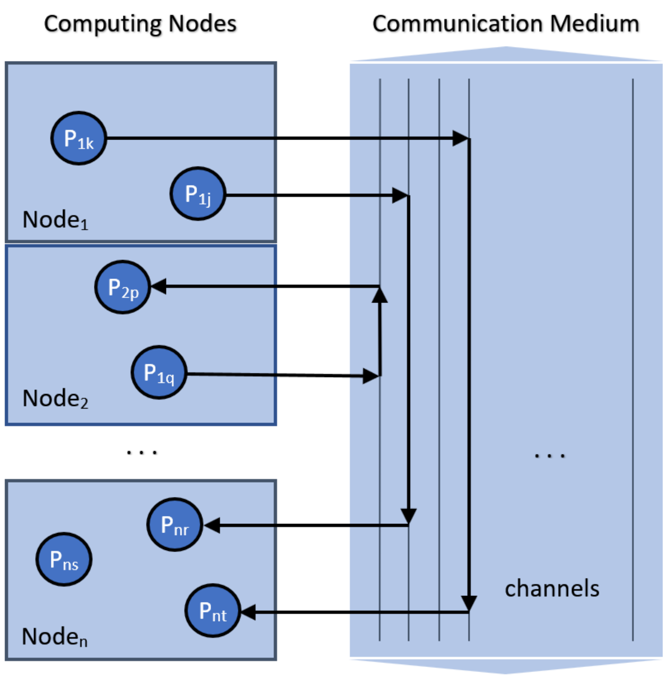


Fig. 1. *CSP* machine

The C.A.R. Hoare's theory of Communicating Sequential Processes (*CSP*) is a strong basis for analysis and synthesis of concurrent computer systems (Abdallah, A., 2005), (Hoare, C.A.R. , 1978) , (Schneider, S., 2000). Its uniqueness lies in its simultaneous manifestation as an abstract model, a parallel systems specification language and a parallel programming language. Therefore, *CSP* is one of the potential means of overcoming above mentioned contradiction.

From engineering point of view *CSP* defines an abstract parallel machine - *CSP* machine (Fig. 1). It consists of two subsystems: a set of computing nodes and a communication medium. In each of the computing nodes are executed one or more concurrent asynchronous processes. The communication medium in turn consists of set of channels. The processes grouped by pairs could communicate only by message passing over a communication channel.

Over its long history, *CSP* has found numerous of successful implementations - from *transputers* and its *OCCAM* language to modern *Go* and *Python* languages and chip multiprocessor platform *xCORE/XC* (Donovan, A., 2016), (May, D. , 2009), (Loukantchevsky, M., 2013), (Watt, D., 2009). The constructive potential of this parallel computational model despite of that is far from being expended.

Whiles *std::thread* class could be regarded near close to *CSP* process semantics, in *C++* standard library there is not type even distant from *CSP* channel (Josuttis, N. , 2012), (Williams, A., 2012). Hence the main goal set: proposal of suitable implementation of the *CSP* semantics of inter-process communications with *C++11* standard library means in seek of higher degree of structuring of communications[1].

EXPOSITION

1. *CSP* Semantics of Inter-process Communications

Let us assume we have a parallel system *S* consisted of two processes – producer *P* and consumer *Q* described by the next *CSP* equations

$$S = \{P||Q\}, P = \{Q!\,msg\}, Q = \{P?\,msg\} \tag{1}$$

Both processes *P* and *Q* form a communication pair (Fig. 2). They interact over *point-to-point* (*p2p*, <*1:1*>) communication channel through pair of communication commands: <*!*> and <*?*>, output (send) and input (receive), respectively.

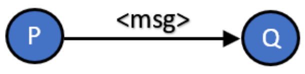


Fig. 2. Pair of communicating processes

The *CSP* semantics of inter-process communications suggest the simplest possible kind of channels: unidirectional, point-to-point <*1:1*>, unbuffered, with direct naming. Such channels do not actually imply flow control but required a special kind of bilateral synchronization known as "*rendezvous*". We could have two scenarios with rendezvous (Fig. 3). In both cases, the first process that came to the point of communication is blocked until another comes to that point.

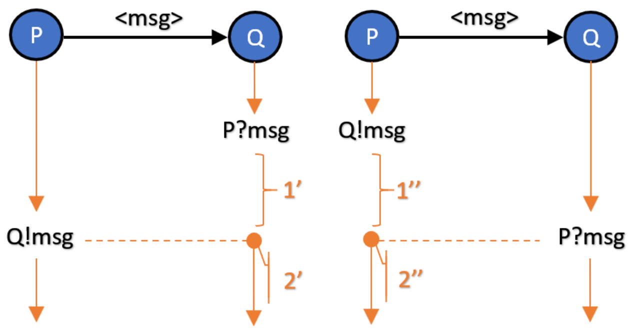


Fig. 3. Rendezvous of communicating processes

[1] For the author, this is in a sense an evolution of his previous works, such as *fiberOS/CSP*.

In parallel systems, as well as in concurrent ones, non-determinism is an intrinsic feature and is result of random choice of transitions. Dijkstra's guarded command is a mean to control and effectively use of the non-determinism in parallel systems (Dijkstra, E. , 1975). Guarded command further evolved by C.A.R. Hoare into *CSP* alternative command

$$\{\mathrm{G}_1 \rightarrow \mathrm{P}_1 \square \mathrm{G}_2 \rightarrow \mathrm{P}_2 \square \cdots \square \mathrm{G}_\mathrm{n} \rightarrow \mathrm{P}_\mathrm{n}\}, \qquad (2)$$

where the i^{th} alternative $\mathrm{G}_\mathrm{i} \rightarrow \mathrm{P}_\mathrm{i}$ consists of a guard G_i and subordinate process P_i. The alternative command is generalization of deterministic <*if-else*> operator. But only if all guards are mutually exclusive, we will have <*if-else*> deterministic behavior, otherwise, the behavior observed will be non-deterministic (Loukantchevsky, M., 2019). Special interest is the case when the guards are input (receive) commands

$$\{P_1?x \rightarrow SKIP \square P_2?x \rightarrow SKIP \square \cdots \square P_\mathrm{n}?x \rightarrow SKIP\} \qquad (3)$$

For simplicity let us consider a command with two alternatives

$$Q = \{P_1?x \rightarrow SKIP \square P_2?x \rightarrow SKIP\} \qquad (4)$$

Since both guards are input (receive) commands they are not mutually exclusive which leads to non-deterministic behavior of Q (Fig. 4).

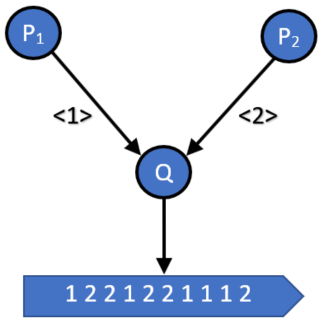


Fig. 4. Two channel non-deterministic selection

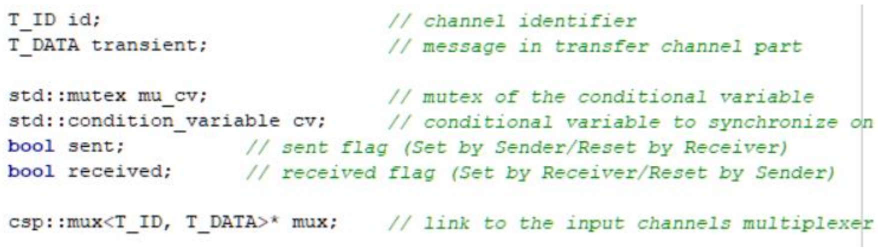
In summary semantics σ of *CSP* inter-process communications could be generalized as a tuple

$$\sigma =< p, c, o, r >, \qquad (5)$$

where $p =< P, Q >$ is the pair of processes communicating over the channel c; $o =\ll ! >, <? >>$ is the pair of commands over the channel c and r is the specific bilateral synchronization of type "*rendezvous*" between processes included in the pair.

2. *CSP* Channel Class *csp::chan* Organization

The communication channel is basic building block of the *CSP* machine's communication medium, as seen in Fig. 1. Hence the need for separate class *csp::chan* in our case. This class encapsulates channel descriptor (Fig. 5), channel constructor, member functions *csp::chan::send()* and *csp::chan::recv()* and few auxiliary functions. They all cover the semantics σ of *CSP* inter-process communications according to Eq. 5.



```
T_ID id;                          // channel identifier
T_DATA transient;                 // message in transfer channel part

std::mutex mu_cv;                 // mutex of the conditional variable
std::condition_variable cv;       // conditional variable to synchronize on
bool sent;            // sent flag (Set by Sender/Reset by Receiver)
bool received;        // received flag (Set by Receiver/Reset by Sender)

csp::mux<T_ID, T_DATA>* mux;      // link to the input channels multiplexer
```

Fig. 5. Channel class *csp::chan* descriptor

The basis of realization is the *C++11* class *std::condition_variable*. It is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread modifies a shared variable and notifies (C++ Reference, 2020), (Gottschling, P., 2016), (Gregoire, M., 2014), (Williams, A., 2012). The *std::condition_variable* class is a means for unilateral synchronization. So, to construct bilateral synchronization needed we have to use two consecutive pairs *wait()/notify()* at sender (producer) and receiver (consumer) part, respectively ordered, as shown in Fig. 6 and Fig. 7.

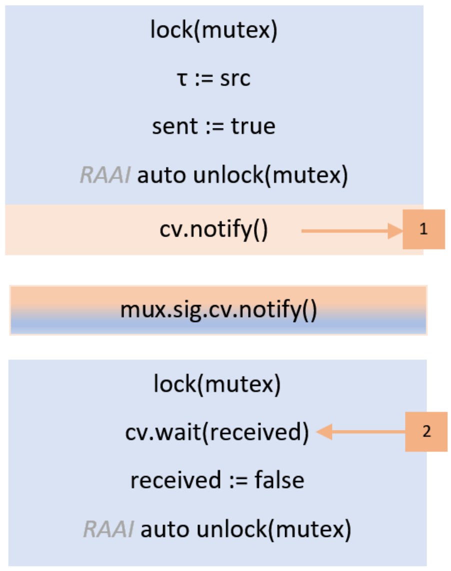


Fig. 6. Specific operating sequence of *csp::chan::send()*

As in the sender, so in the receiver is used predicative form *void wait(std::unique_lock<std::mutex>& lock, Predicate pred)* of *std::condition_variable::wait()*, where the predicate is implemented as *lambda* function (Bartlomiej, F. , 2020), (Josuttis, N., 2012). Between two stages in Fig. 6 is placed a notification to *std::condition_variable* of *csp::mux*, if the channel is part of the multiplexed inputs (*std::vector <std::shared_ptr<CHAN>> in* at Fig. 8).

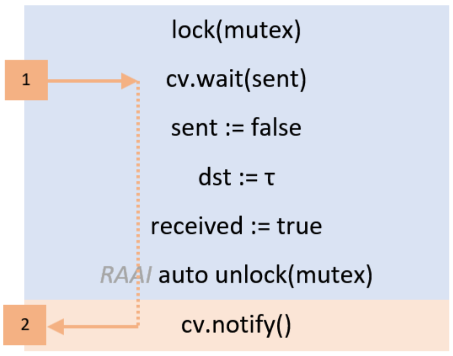
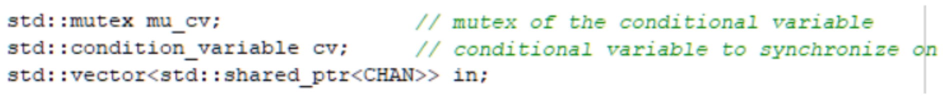


Fig. 7. Specific operating sequence of *csp::chan::recv()*

The estimated effectiveness of message transfer with the pair <*csp::chan::send(), csp::chan::rec()*> is *O(std::mutex)*. In such a way we achieved a cumulative effect – maximum proximity with the *CSP* semantics, encapsulation and hiding of low level mechanism of *std::condition_variable* and effectiveness near to that of *std::mutex*.

3. Non-deterministic *CSP* Alternative Command Realization

Except communication scheme <*1:1*> provided with the *csp::chan* class it is necessary to maintain <*n:1*> scheme, and that in more general non-deterministic form given by Eq. 3.



```
std::mutex mu_cv;               // mutex of the conditional variable
std::condition_variable cv;     // conditional variable to synchronize on
std::vector<std::shared_ptr<CHAN>> in;
```

Fig. 8. Multiplexor class *csp::mux* descriptor

In first versions of the *csp* library this was done by separate function *csp::alt()* later transformed into the method *csp::mux::recv()* being part of the multiplexor class *csp::mux*. Its descriptor includes conditional variable to synchronize on and privately owned mutex (Fig. 8). The multiplexed input channels $C_1, C_2, \ldots, C_n$ are contained in the data member of type *std::vector <std::shared_ptr<CHAN>>*.

When started *csp::mux::recv()* is blocked on std::mux conditional variable. It is marked at Fig. 9 by $G[n]$ as association with array of *n* input guards. When at least one of these guards signals, *csp::mux::recv()* unblocks and builds the list *L* with all guards evaluated to true. The index k of the selected guard contained in *L* is determined by the uniform distributed random function ĝ. Actually, we have a form of Data Level Parallelism (*DLP*) sequencer which select one of the input streams and output it to the exit of the circuit.

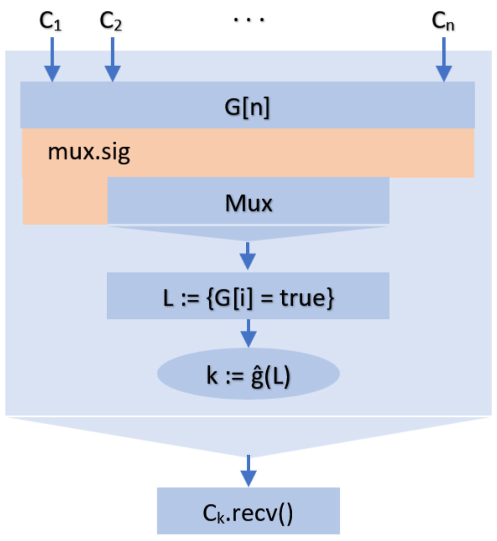


Fig. 9. Generalized operating structure of *csp::mux::recv()*

As in the *<1:1>* communication scheme, so here is used predicative form of *std::condition_variable::wait()* implemented as *lambda* (Bartlomiej, F. , 2020), (Josuttis, N., 2012). Actually here we have multiple wait over *std::condition_variable* which is resolved according to *CSP* alternative command non-deterministic semantics. Moreover, in addition to the default non-deterministic dispatching of input messages is provided "*by place*" dispatching too.

Example close by functionality to *csp::mux* is *CSP* alternative command implementation in the exokernel *fiberOS/CSP* (Loukantchevsky, M., 2019). There are provided separate *2*-channel and *n*-channel *ALT()* primitives. But these primitives are tightly bounded to exokernel *fiberOS/CSP* and cannot be used separately.

Another example of *CSP* alternative command implementation is the operator *<select>* within the platform *xCORE/XC* (May, D. , 2009), (Watt, D., 2009). Moreover, this is done at microarchitectural level. Unfortunately, such support is not common in general purpose computer architectures yet. The author's studies revealed furthermore that *xCORE/XC* implementation is not true non-deterministic.

The communication schemes *<1:1>* and *<n:1>* are in principle sufficient to perform any possible type of inter-process communications. But from practical reasons, the additional classes *csp::sink* and *csp::fork* are included in the *csp namespace* as well. The *csp::sink* class supports a communication scheme *<n:1/n>*. This is a kind of input fork, in which multiple waits are performed at the input and messages from *n* sources are received at the same time. Conversely, the *csp::fork* class performs multiple waits by output thus supporting the communication scheme <*1:n*> with multicast send. They both are subject of another discussion.

CONCLUSION

Following the practice of standard *C++* template libraries is developed a compact csp library which encapsulates the means of messaging and synchronization between objects of type *std::thread*. The *csp* library evolution begins from a point-to-point <*1:1*> communication over *CSP* proto channel and goes through several versions: *0.1*, *0.2*, …, *0.7RC* and *0.8RC* till now. The last one supports not only the basic communications schemes <*1:1*> and <*n:1*> through *std::chan* and *std::mux* classes considered here, but also the communications schemes <*n:1/n*> and <*1:n*> through additional classes *csp::sink* and *csp::fork*.

Numerous of testbed applications were developed, covering different types of interactions separately and their combinations (*p2p-mux*, *sink-mux*, *fork-mux*, etc.). Support of *co-procedures* with decentralized and centralized management as well as *active shared objects* is explored too.

The overall development was performed through the *Embarcadero C++ Builder*® development environment and its *Clang-enhanced C++* compiler (C++ Support in Clang. , 2020), (Embarcadero RAD Studio Docwiki, 2020). However, both the library itself and the testbeds include only standard constructions and hence are portable at source level for use with other *C++11* compatible environments. Finally, the code of the developed library and examples of its use is going to be posted on *GitHub*.

REFERENCES

Abdallah, A., C. Jones, J. Sanders (Eds.). (2005). Communicating Sequential Processes: The First 25 Years. - Berlin: Springer-Verlag.

Bartlomiej, F. (2020) C++ Lambda Story. URL: https://leanpub.com/cpplambda (Accessed on 26 Sept. 2020).

C++ Reference: The condition_variable class. (2020). URL: https://en.cppreference.com/w/cpp/thread/condition_variable (Accessed on 01 Nov. 2020).

C++ Support in Clang. (2020). URL: http://clang.llvm.org/cxx_status.html#cxx17 (Accessed on 01 Nov. 2020).

Dijkstra, E. (1975). Guarded Commands, nondeterminancy and formal derivation of programs. // Communications of the ACM, Vol. 18, Num. 8, pp. 453-457.

Donovan, A., B. Kernighan. (2016). The Go Programming Language. – New York: Pearson Education, Inc.

Embarcadero RAD Studio Docwiki. (2020). URL: http://docwiki.embarcadero.com/RADStudio/Sydney/en (Accessed on 01 Nov. 2020).

Gottschling, P. (2016) Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers. – Boston: Pearson Education, Inc.

Gregoire, M. (2014). Professional C++. 3rd Ed. - Indianapolis: John Wiley & Sons, Inc.

Hoare, C.A.R. (1978). Communicating Sequential Processes. // Communications of the ACM, Vol. 21, Num. 8, pp. 666-677.

Josuttis, N. (2012). The C++ Standard Library: A Tutorial and Reference. 2nd Ed. - Boston: Pearson Education, Inc.

Loukantchevsky, M. (2013). *Modelirane na kvantovi izchisleniya v paralelna izpulnitelna sreda*. Ruse: Izdatelski tsentar na RU „A. Kanchev", ISBN 978-619-7071-25-2. (***Оригинално заглавие:*** *Луканчевски, М. Моделиране на квантови изчисления в паралелна изпълнителна среда. Русе, Издателски център на РУ "А. Кънчев", 2013.*)

Loukantchevsky, M., N. Kostadinov, H. Avakyan. (2019) A Testbed of Non-determinism in Educational Context. IN: Proceedings of the 20th International Conference on Computer Systems and Technologies, New York, NY, USA, ACM, pp. 304-307, ISBN 978-1-4503-7149-0.

May, D. (2009). The XMOS XS1 Architecture. – XMOS Ltd.

Schneider, S. (2000). Concurrent and Real-time Systems: The CSP Approach. – Chichester: John Wiley & Sons, Inc.

Watt, D. (2009). Programming XC on XMOS Devices. – XMOS Ltd.

Williams, A. (2012). C++ Concurrency in Action: Practical Multithreading. – Shelter Island: Manning Publications Co.