A Graphical Modeling Language for Specifying Concurrency based on CSP[1]

Gerald H. HILDERINK

University of Twente
Department of Electrical Engineering – Control Laboratory
Cornelis J. Drebbel Institute for Mechatronics
Twente Embedded Systems Initiative
P.O.Box 217, 7500 AE Enschede, The Netherlands

g.h.hilderink@utwente.nl

Abstract. Introduced in this paper is a new graphical modeling language for specifying concurrency in software designs. The language notations are derived from CSP and the resulting designs form CSP diagrams. The notations reflect both data-flow and control-flow aspects, as well as along with CSP algebraic expressions that can be used for formal analysis. The designer does not have to be aware of the underlying mathematics. The techniques and rules presented provide guidance to the development of concurrent software architectures. One can detect and reason about compositional conflicts (errors in design), potential deadlocks (errors at run-time), and priority inversion problems (performance burden) at a high level of abstraction. The CSP diagram collaborates with object-oriented modeling languages and structured methods.

1 Introduction

The *Communicating Sequential Processes* language (CSP) is a process algebra for describing and analyzing concurrent systems [1,2]. CSP embraces the formal mathematics so we can specify requirements precisely and prove that our implementations satisfy them. However, a process algebra, such as CSP, in its textual form is not a pleasant modeling language for developing software. Usually, we prefer a graphical modeling language and we leave the mathematics to the tools we use. This does not mean that we should ignore CSP. On the contrary, CSP provides fundamental elements for specifying, designing and analyzing (reasoning and proofing) concurrent software that is relevant to software engineering. In other words, CSP provides a design concept and at the same time it can give guarantees about the reliability and safety of software architectures at every level in the development from design model down to its implementation. Furthermore, developing reliable and robust software for embedded real-time systems is crucial. This is not solely an implementation issue, but also a modeling issue. This is why the CSP essentials are so important for developing concurrent software.

[1] This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

In this paper, a graphical modeling language is introduced for specifying concurrency in software design. The language notations are derived from CSP and the resulting designs form CSP diagrams. The notations reflect both data-flow and control-flow aspects, as well as along with CSP algebraic expressions that can be used for formal analysis. The designer does not have to be aware of the underlying mathematics. Associated systematic techniques and rules presented in this paper provide guidance to detect and reason about compositional conflicts (i.e. errors in design), potential deadlocks (i.e. errors at run-time), and priority inversion problems (e.g. performance burden) at a high level of abstraction. The CSP diagram collaborates with object-oriented and structured methods [3,4,5]. The CSP diagram is UMLable and can be presented as a new process diagram for the UML [6,7,8] to capture concurrent, real-time, and event-flow oriented software architectures. The extension to the UML is not discussed in this paper.

In this paper, the word “process” is used many times and implies implicitly to the CSP process. In many ways, CSP represents the most fundamental properties commonly meant by a ‘process’. Therefore we are free to use “process” without referring to CSP.

2 The CSP Diagram

A CSP diagram is a graph of processes and their interrelationships. A CSP diagram enables us to specify parallel and real-time software architectures. The graphical presentation expresses the execution model of a network of processes. The validity of the graph indicates the validity of the execution model. Any valid execution model can be transformed into code or into machine readable CSP for formal analysis. The analysis can guarantee the robustness and reliability of the resulting software. The mathematical analysis in CSP is not a subject of this paper.

The theory of CSP [2] defines a few fundamental primitives that are necessary for describing the essentials of concurrent systems. Although CSP has no notion of priorities, we have added extended prioritized primitives to deal with priorities in software. These are similar primitives as found in the programming languages occam and Ada for achieving real-time requirements. We will represent these fundamental primitives as relationships between processes. Relationships are displayed as lines and processes are displayed as circles, bubble, or rectangles. Some relationships are communication-oriented and some are composition-oriented. Therefore, we distinguish the communication-oriented view and the composition-oriented view of the model. The communication-oriented view presents a *communication-graph*, which expresses the communication relationships between the processes. The composition-oriented view presents a *composition-graph*, which expresses the compositional relationships between the processes. Each graph can be viewed as an individual diagram. Therefore, the CSP diagram is a hybrid diagram consisting of two different views, namely the *communication diagram* and *compositional diagram*. A mixed view of both is also conceivable. The sets of processes in the communication diagram and the composition diagram are usually the same. An overview of these relationships is given in Figure 1.

Many data-flow languages, such as found in structured methods, do not specify whether processes run in parallel or in some sequence. The semantics of the execution framework of a data-flow diagram is decoupled from its implementation. What seemed to be a benefit is what we consider a very important disadvantage of data-flow modeling since no necessary changes can be enforced to the execution framework by the designer. For example, the underlying hardware, the distributive character of the application, and real-time requirements determine the concurrent nature of the software architecture. These properties are significant to the architecture of the model and they should be specified within the model by the designer. For this purpose the compositional relationships extend data-flow modeling (in a different layer) for specifying those aspects that are not expressed by data-flow diagrams with solely nodes and arrows. The communication diagram expresses a basic form of data-flow modeling.

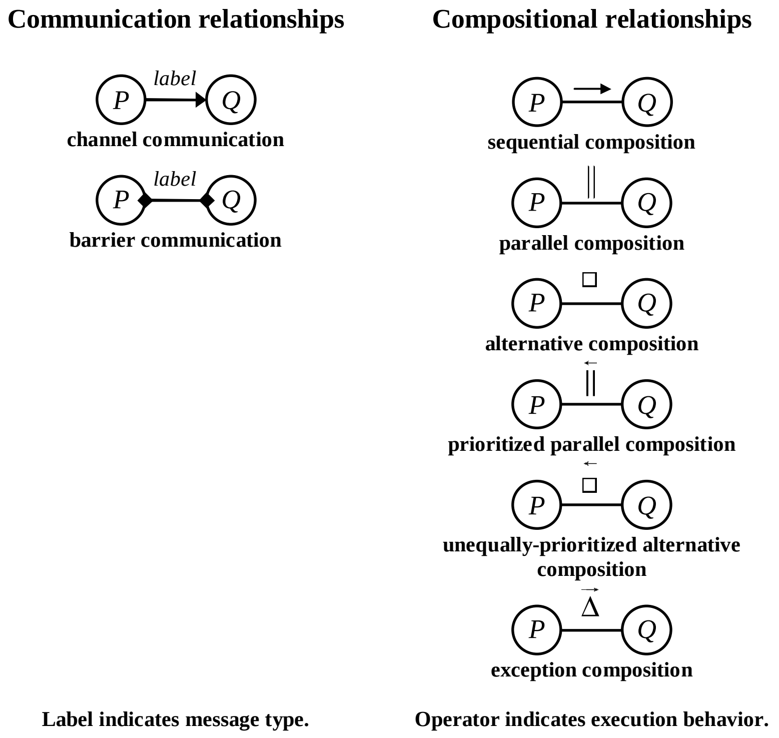

Communication relationships
Compositional relationships
label
P
Q
channel communication
label
barrier communication
sequential composition
parallel composition
alternative composition
prioritized parallel composition
unequally-prioritized alternative composition
exception composition
Label indicates message type.
Operator indicates execution behavior.

Figure 1. Overview of CSP relationships

Besides the graphical notations, a few basic techniques are described for checking the correctness of the composition diagram. With these modeling techniques, the designer is able to determine and to reason about compositional conflicts, potential deadlocks, and priority inversion problems in the design. Of course, this model checking can be automated by a design tool.

2.1 Processes

A *process* is an entity that performs a sequence of events. An *event* is an occurrence in time and space in which at least two processes participate. A process encapsulates its own workspace, its data structure, and the operations that operate on this data structure. Although these properties have strong similarities with objects, it is important to notice that a process is not an object and that a process should not be treated as an object. Objects can only operate in a workspace and that workspace is defined by a process. Usually, objects know each other, but processes do not know each other. Processes communicate with each other through intermediate communication objects, which are called *channels*. In Section 2.3, a few different kinds of channels are discussed. There are several ways whereby processes can be applied to object-oriented paradigms. For example, objects can be part of a process; a process can be implemented using objects. Another important property of processes is that processes can be composed and described by simpler processes. A process is depicted as a vertex in the form of a labeled rectangle or labeled bubble in the graph. See Figure 2.

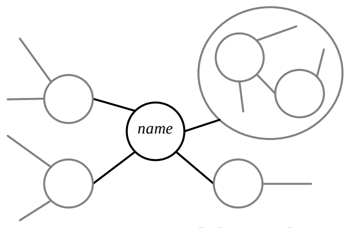


Figure 2. Processes and their relationships.

A process has a functional task description and each process is identified with a unique name. Process names should be nouns representing the role, functionality or responsibility they perform. This is similar as with naming objects [8]. Identifying "active" objects and identifying processes have lots in common because they share a similar task identification process. From a design point of view, the relationships between processes and the relationships between objects are different. This paper discusses the relationships between processes resulting in process diagrams (or CSP diagrams). Object diagrams are omitted in this paper.

2.2 Interrelationships

Even when processes do not see each other nor communicate with each other, they are always mutual related to each other since they share the same world. The simplest relationship is *concurrency* whereby processes execute in parallel and independently of each other. Concurrency may also involve synchronization between processes. Processes execute and synchronize in many ways, which can be expresses by relationships between the processes. The interrelationship is depicted as a line (or edge) between two processes. Additional symbols can be attached to the peer-ends of the line to indicate a directional relationship.

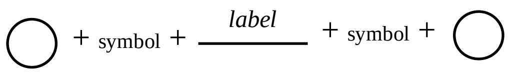


Figure 3. Interrelationship between processes.

These symbols and label specify the kind of relationship between processes. The graphical modeling language specifies a set of CSP-based relationships in the following sections. As mentioned before, the CSP-based relationships have been divided into communication relationships and compositional relationships.

Important is to note that the line should be seen distinct from the symbols that can be attached to each end of the line. The line renders an event that both processes can engage in. Depending on the kind of relationship, the associated event can be a communication event, termination event, exception event, or a timeout event. The line itself is undirected because events are symmetric and undirected [2]. The symbols are a gloss on this and they indicate the polarization of message passing or they assist in composing processes.

2.3 Communication Relationships

Two processes participate in a communication relationship when they communicate with each other. Message passing and data-exchange are two common forms of communications between processes that are supported by the communication relationships.

Definition (Communication relationship). A communication relationship is a labeled and directed relationship, which represents message flow between a sender process and receiver process.

A communication relationship is symbolized as an arrow between two processes. The arrow symbolizes channeled message passing between processes.

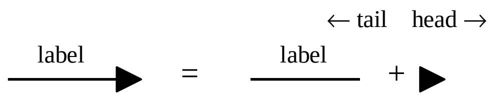


Figure 4. Communication Relationship

The arrow designates the role of the processes in the relationship. The arrow head symbol '▶' is attached to the receiver or the invoked process and the tail of the arrow is attached to the sender or invoker process. This does not necessarily imply that data is moving strictly from tail to head at communication. Data may very well be returned at the end of the invocation. The communication relationship specifies that communication can take place between the related processes, but it does not specify exactly when communication takes place. The actual channel invocations can be specified by a few primitive communication processes are described in Section 2.15.

Communication relationships can be divided into *producer-consumer relationships* or *client-server relationships*. The label expresses the type of message passing. The label can be an abstract data type (usually a class name) or a method name.

An abstract data type specifies the producer-consumer relationship. The class name specifies the type of messages (i.e. instances of that class) that can be passed from the producer process to the consumer process on rendezvous. An optional role name before ":", for example `length:Integer` specifies messages of type Integer and plays the role of length. See Figure 5a. This type of relationship is implemented using *data-channels*. Data-channels send data and do not return anything and are therefore data-unidirectional. The abstract data type determines a data-channel and a data-channel can be one out of several possibilities, namely:

1. rendezvous channel
2. buffered channel (fifo, supersampling, or subsampling)
3. unsynchronized channel or variable

The compositional relationship orthogonal to the communication relationship determines the appropriate channel.

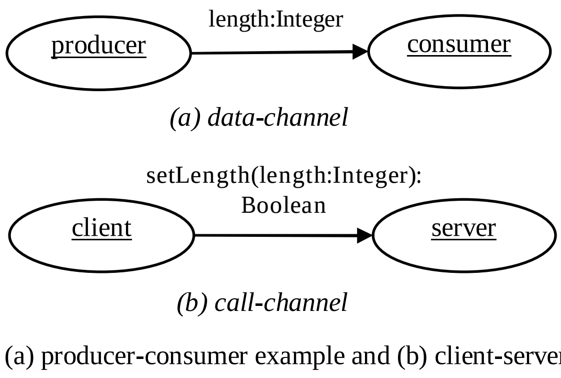


Figure 5. (a) producer-consumer example and (b) client-server example.

A method name specifies a client-server relationship. The client process invokes the method and it is executed when the server accepts the call. Both participating processes must rendezvous. For example, `setLength(length:Integer):Boolean` represents a method with an argument of type `length:Integer`. The arguments are similar as with data-channels. The method passed to the server and returns true or false to the client. See Figure 5b. This type of relationship is implemented using *call-channels*. Call-channels can be data-unidirectional or data-bidirectional. The method name determines a rendezvous call-channel independent of the compositional relationship. In this version, buffered calls and unsynchronized class between processes is not supported.

The process's inputs and outputs specify the channel type. Each pair of input and output must be of the same type otherwise they are incompatible. Incompatible inputs and outputs cannot be connected. For example, a channel-output of message type Integer cannot communicate with a channel-input of message type Float. Also, a producer process cannot communicate with a server process, because the producer process requires a data- channel and the server process requires a call-channel. This is similar for a consumer process and client process.

Commonly, data-channels are low-level communication primitives and are optimized for low-level communication. Call-channels are higher-level communication primitives and are used when the data-channel interface is too restrictive. In this case, using data-channels for higher-level communication can become less efficient than using call-channels. However, data-channels are very efficient and simple to use for building data-driven applications like control systems. Data-channels can efficiently establish communication through hardware. Their abstract and primitive interface provides hardware independency.

The previous communication relationships express unconditional communications, i.e., if both participating processes are ready for communication then they are committed in communication. The will engage in a *communication event* and withdrawing is impossible. Conditional communication is a circumstance where by the readiness of the channel is required as a condition. A process may commit in communication when the other side is ready otherwise it will avoid the commitment and carries on. Conditional communication requires a guard at the input or output side of the relationship. The relationship may select the guarded process based on the readiness of the channel and based on an additional Boolean expression. The nature of selection is specified by the alternative relationship (Section 2.4.3) as expressed in the composition-graph.

2.3.1 Shared Channels

Channels can be shared between two or more processes. The arrow may consist of branches of multiple tails and/or multiple arrow heads. This is rendered as a fish bone. We can identify four different channel configurations, as shown in Figure 6.

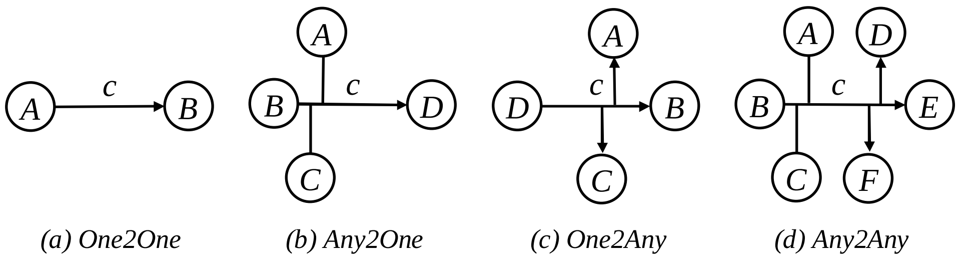


Figure 6. Channel configurations

Channels provide a peer-to-peer connection between two processes at a time. A channel communication event is two-way, i.e., only two processes (one inputting and one outputting) can engage in the event. A barrier communication event can be multi-way, i.e., more than two processes can engage in the same event (Section 2.3.2).

Configuration *a* depicts channel communication between two processes. Configuration *b* and *c* implement a non-deterministic choice of service between multiple writers. The configurations *c* and *d* implement a non-deterministic choice of delivering messages between multiple readers. The service or delivery is uncertain and is likely to be unfair. In practice and in a worst case, a reader may read all the time and other readers get no change to get a message. This is similar for multiple writers. There is a risk of starvation. A deterministic form is more common, for example, when readers or writers fairly (timely ordered) alternate on the channel. Virtually, the configurations *b*, *c* and *d* swap to configurations *a* with fairly alternating reader and/or writer processes. The access time and the priority are important parameters for fairly scheduling.

2.3.2 Barrier

Another communication primitive is the barrier synchronization primitive. A fixed number of processes are required to synchronize their execution at some point before proceeding. A barrier is depicted as a bidirectional communication relationship whereby each end is symbolized with a diamond ◆ symbol (concatenation of ◀+▶). See Figure 7.

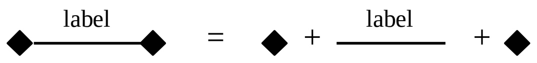


Figure 7 . Barrier synchronization relationship.

When all processes reach the barrier synchronization then the barrier construct communicates information between the processes in a unidirectional or bidirectional way. All processes continue after communication is performed.

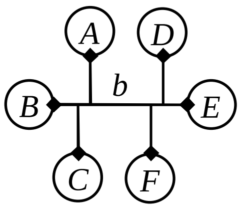


Figure 8. Barrier synchronization.

A barrier synchronization pattern could be described in terms of a protocol of channel communications [9]. The point of synchronization is not rendered by this relationship. The actual point of synchronization is rendered by a primitive communication process as described in Section 2.15.

2.3.3 Internal and External Channels

An arrow that connects each end to a different process and at the same level of abstraction is an *internal channel*. An arrow that connects one end to a process and the other end to a process at a different or distinct level of abstraction is an *external channel*. External channels are open ended and this open end is virtually connected to some other process that is out of the scope of the parent process or diagram. This happens when:

- hierarchies of processes are involved,
- software processes separately run on distributed systems,
- or when software processes communicate with hardware processes.

The tool should be able to distinguish between internal and external channels. The tool should facilitate means to link channels from different levels with each other or the tool should be able to assign device drivers (or link drivers) to the channels that communicate through intermediate devices.

2.4 Compositional Relationships

Compositional relationships are a kind of relationships between processes that are useful for describing the execution order of communicating processes. A compositional relationship is a companion to a communication relationship.

Definition (Compositional relationship). A compositional relationship is a labeled relationship between two processes whereby the label is a binary operator that expresses their compositional behavior.

Figure 9 shows two processes in relation to each other. This represents a composition of two processes which semantics is described by the operator specified by the label.

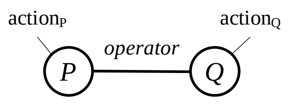


Figure 9. Compositional relationship between two processes.

Where *operator* $\in$ $\{\rightarrow, \leftarrow, \|, \overleftarrow{\|}, \overrightarrow{\|}, \Box, \overleftarrow{\Box}, \overrightarrow{\Box}, \overleftarrow{\Delta}, \overrightarrow{\Delta}\}$. The operators with an arrow are directed operators and the remainders are undirected operators. Each compositional relationship is explained in the following sections.

Optionally, an `action` attribute can be specified next to the associated process connected with a thin line to show its association. The `action` attribute does by no means belong to the associated process. Here, `action` can be used to specify a global body in which variables can be declared and/or assignment statements change the state of those variables. If an `action` attribute is specified then it will be executed right before the process will be executed, i.e. $action_P;P$ and $action_Q;Q$ The scope of `action` is determined by the parenthesizing compositional relationships (see Section 2.5). The `action` attribute is very useful for conditional communications (Section 2.4.3) and for finite looping constructs (Section 2.14). The rules that are applied to `action` are not described in this paper.

2.4.1 Sequential Relationship

A sequential relationship between processes P and Q is denoted by the label '$\rightarrow$'. This sequential composition is written as $P \rightarrow Q$. This has strong similarities with CSP's single action transition $P \overset{\surd}{\rightarrow} Q$. This process will behave as Q if P has successfully terminated ($\surd$-event) otherwise this process behaves as P. We will relax these semantics by saying P is executed before Q. This relationship (being a process) terminates when Q successfully terminates ($\surd$-event). We will use notation $(P,Q,\rightarrow)$ to represent a sequential relationship between P and Q. See Figure 10. A sequential composition in CSP is usually represented as $P;Q$ whereby Q immediately follows P when P terminates. The relationship $(P,Q,\rightarrow)$ is more relaxed as described above and represents $P \rightarrow Q$. The notation $P;Q$ is a special case of $P \rightarrow Q$ as described below. The ';' and '$\rightarrow$' operators have no symmetry laws.

The reasons why we use the arrow '$\rightarrow$' instead of semicolon ';' is because the arrow gives more design freedom and the arrow denotes the direction of execution more clearly than ';'.

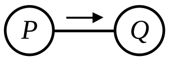


Figure 10. Sequential relationship.

Multiple compositions are represented with more than two processes in the relationship, for example, $(P,Q,R,S,\rightarrow)$. See Figure 11a. $(P,Q,R,S,\rightarrow)$ also represents other partial relationships, such as, $(P,Q,\rightarrow)$, $(P,R,\rightarrow)$, $(P,S,\rightarrow)$, $(Q,R,\rightarrow)$, $(Q,S,\rightarrow)$, $(R,S,\rightarrow)$, $(P,Q,S,\rightarrow)$, and $(P,R,S,\rightarrow)$. See Figure 11b. Partial relationships can be derived from the main multiple relationships.

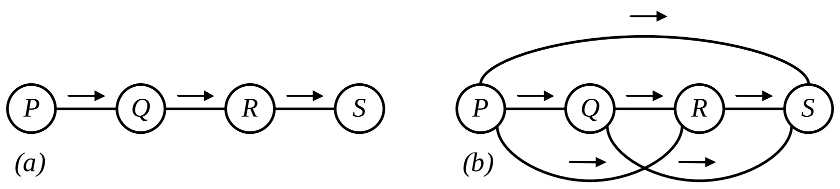


Figure 11. (a) Multiple sequential relationship and (b) multiple sequential relationship with derived relationships.

These partial relationships *cannot all* be represented with ‘;’. It is obvious that $(P,S,\rightarrow)$ does not represent $P;S$. Only process $P;Q;R;S$ is represented with ‘;’.

Formally, what we mean by $(P,S,\rightarrow)$ is:

$$(P,X,S,\rightarrow)\backslash X$$

where X is a set of processes belonging a second and longest path between P and S, thus here $X = \{Q,R\}$. This algebraic expression has resemblance with the hiding operation ‘$P\backslash X$’ in CSP. In CSP, X is a set of events and here X is a set of processes. Basically, this means that if no other processes can be found between P and S that forms the main path between P and S then this means that X is empty. If X is empty then

$$(P,\{\},S,\rightarrow)\backslash\{\} = (P,S,\rightarrow)_{\varnothing} = P;S.$$

In case of a multiple relationship we can write

$$(P,Q,R,S,\rightarrow)_{\varnothing} = (P,Q,\rightarrow)_{\varnothing}\,,\,(Q,R,\rightarrow)_{\varnothing}\,,\,(R,S,\rightarrow)_{\varnothing} = P;Q;R;S$$

In the graph this expression means *the longest paths* or *main path* between the processes P and S from P to S. The importance of this form is that we can index processes in the compositional construct so that processes are successively ordered.

We could write:

$$(P_0,\ldots,P_{n\text{-}1},\rightarrow)_{\varnothing} = \mathop{;}_{i=0..n-1} P_i$$

This form allows immediate transformation to sequential code-constructs in CSP based programming languages or using a CSP library in non-CSP based programming languages.

2.4.2 Parallel Relationship

A parallel relationship between processes P and Q is denoted by the label ‘$\|$’. This parallel composition is written as $P\|Q$. This process will behave as P and as Q in parallel. This process terminates when all participating processes, i.e. P and Q, have terminated.

We will use notation $(P,Q,\|)$ to represent a parallel relationship between P and Q. See Figure 12.

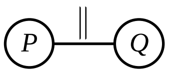


Figure 12. Parallel relationship.

A multiple composition ($P,Q,R,S,\|$) represents $P\|Q\|R\|S$. Operator $\|$ has symmetry laws and therefore all partial relationships can be represented with '$\|$'. See Figure 13.

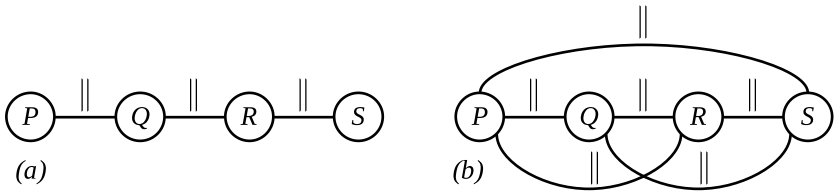


Figure 13. (a) Multiple parallel relationship and (b) multiple parallel relationship with derived relationships.

Therefore, we can write:

$$(P_0,\ldots,P_{n\text{-}1}, \|) = \mathop{\Big\|}_{i=0..n-1} P_i$$

The processes $P_0..P_{n\text{-}1}$ can be randomly ordered since operator $\|$ has symmetry laws.

When `action` attributes are specified on parallel processes then these `action` attributes will be executed in parallel. Any race hazards between shared variables in multiple `action` attributes must be prevented and therefore it is important that assignments can not update shared variables.

2.4.3 Alternative Relationship

An alternative relationship between processes P and Q is denoted by label '$\Box$'. This alternative composition is interpreted as $P\Box Q$. This process will behave as P if P can engage in a communication event or it behaves as Q if Q can engage in a communication event. If both processes can engage in a communication event then the alternative construct will choose one arbitrarily. This process terminates when the selected guarded process terminates.

The notation ($P,Q,\Box$) represents an alternative relationship between P and Q. See Figure 14.

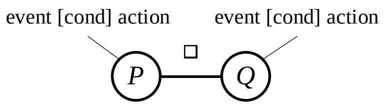


Figure 14. Alternative relationship.

A guard is depicted with a guard expression `event [cond] action` next to the guarded process with a thin line connecting the expression with the guarded process in the alternative relationship. A guarded process has only one guard expression attached to it.If the Boolean expression `cond` (or condition) is true and the guarded process can engage in `event` then the choice operator may select the guarded process. If `cond` is false then `event` will be omitted and the guarded process will not be selected. Once the guarded process is selected then `action` will be executed prior to the guarded process is executed. Here, `event` can indicate a *channel-input*-guard, *channel-accept*-guard, *channel-output*-guard, *channel-call*-guard, *skip*-guard, or an *else*-guard. A variety of guard expressions are shows in Table 1.

`channel?`	unconditional *channel-input* guarded process
`channel? [cond]`	conditional *channel-input* guarded process
`channel!`	unconditional *channel- output* guarded process
`channel! [cond]`	conditional *channel-output* guarded process
`callchannel.method!`	unconditional *channel-call* guarded process on specified method
`callchannel.method! [cond]`	conditional *channel-call* guarded process on specified method
`callchannel.method?`	unconditional *channel-accept* guarded process on specified method
`callchannel.method? [cond]`	conditional *channel-accept* guarded process on specified method
`callchannel?`	unconditional *channel-accept* guarded process on any method
`callchannel? [cond]`	conditional *channel-accept* guarded process on any method
`skip`	unconditional *skip* guarded process
`skip [cond]`	conditional *skip* guarded process
`else`	unconditional *else* guarded process
`else [cond]`	conditional *else* guarded process
`timeout(t)`	unconditional *timeout* guarded process
`timeout(t) [cond]`	conditional *timeout* guarded process

Table 1. Variety of guard expressions

The words `channel` and `callchannel` should be replaced by a channel name. The word `cond` represents a Boolean expression (or condition) and `method` should be replaced with the actual method name. The names `skip`, `else`, and `timeout` are special keywords and `t` represents the specified time. The channel-inputs, channel-outputs, and channel-calls may require additional arguments in order to express the variables that are involved in communication. In this version of the Graphical Modeling Language, these arguments must be specified in the code and not in the model. This is because the arguments have a local scope and they must be declared in the diagram. Probably, these declarations can make the diagrams less readable. More research is needed of how to deal with arguments at this level of abstractness.

A *skip*-guard does not require a channel-input or channel-output and the guard is ready all the time. An *else*-guard cannot be found in CSP but it is like a *skip*-guard with the difference that it will be selected if no other guard is ready. The *else*-guard can be modeled as a *skip*-guard in a special prioritized alternative construction. See Figure 15.

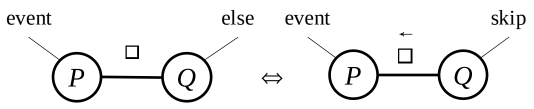


Figure 15. else-guarded construct.

The guard is said to be *unconditional* when `cond` is always true (or not specified) and the guard is said to be *conditional* when `cond` is some Boolean expression.

Guards can also be applied to shared channel but not to barrier configurations. It is important to notice that a channel-input-guard and a channel-output-guard specified at different processes in the same alternative relationship will *never* commit in communication [10].

Design rule. All guards sharing the same alternative relationship must be disjoint in such a way that no pair of channel-input-guards and channel-output-guards can become simultaneously ready.

The composition ($P,Q,R,S,\square$) represents multiple relationships. See Figure 16.

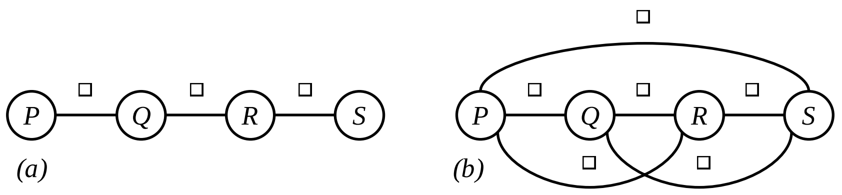


Figure 16. (a) Multiple alternative relationship and (b) multiple alternative relationship with derived relationships.

Here, $(P,Q,R,S,\square)$ is written as $P \square Q \square R \square S$ because

$$(P_0,\ldots,P_{n\text{-}1},\square) = \underset{i=0..n-1}{\square} P_i$$

Operator $\square$ has symmetry laws and so the processes $P_0..P_{n\text{-}1}$ can be randomly ordered.

2.4.4 Prioritized Parallel Relationship

A prioritized parallel relationship between processes *P* and *Q* is denoted by label '$\overleftarrow{\|}$'. This parallel composition is written as $P \overleftarrow{\|} Q$. If process *P* cannot engage in an event (i.e., communication events, termination events, and timeout events) then it will behave as *Q* otherwise it behaves as *P*. In other words, process *P* is executed with higher priority than process *Q*. This process terminates when all participating processes terminate.

We will use notation $(P,Q,\overleftarrow{\|})$ to represent a prioritized parallel relationship between *P* and *Q*. See Figure 17.

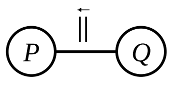


Figure 17. Prioritized parallel relationship.

The multiple relationship $(P,Q,R,S,\overleftarrow{\|})$ represents $P \overleftarrow{\|} Q \overleftarrow{\|} R \overleftarrow{\|} S$ when *P*, *Q*, *R*, and *S* belong to the longest part between *P* and *S* in the graph. See Figure 18.

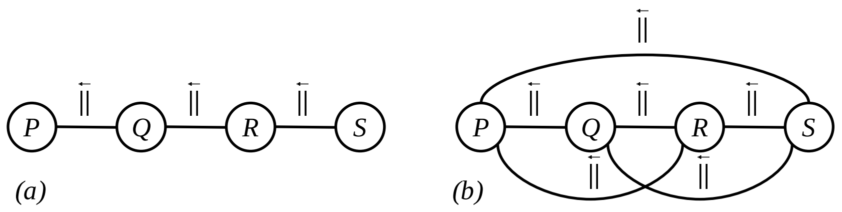


Figure 18. (a) Multiple prioritized parallel relationship and (b) multiple prioritized parallel relationship with derived relationships.

As with the sequential composition, the directed operator $\overleftarrow{\|}$ has no symmetry laws. Therefore, we mean by $(P,S,\overleftarrow{\|})$:

$$(P,X,S,\overleftarrow{\|})\backslash X$$

where *X* is a set of processes belonging to a second and the longest path between *P* and *S*, thus, $X = \{Q,R\}$. If *X* is empty then

$$(P,\{\},S,\overleftarrow{\|})\backslash\{\} = (P,S,\overleftarrow{\|})_{\varnothing} = P \overleftarrow{\|} S.$$

In case of a multiple relationship we can write

$$(P,Q,R,S,\overleftarrow{\|})_{\varnothing} = (P,Q,\overleftarrow{\|})_{\varnothing}\,,\,(Q,R,\overleftarrow{\|})_{\varnothing}\,,\,(R,S,\overleftarrow{\|})_{\varnothing} = P \overleftarrow{\|} Q \overleftarrow{\|} R \overleftarrow{\|} S$$

In the graph this expression is the longest path or main path between the processes *P* and *S* from *P* to *S*. The importance of this form is that we can index processes in the compositional construct so that processes are successively ordered and with declining priorities. We can write:

$$(P_0,\ldots,P_{n\text{-}1},\overleftarrow{\|})_\varnothing = \overleftarrow{\Big\|}_{i=0..n-1} P_i$$

This form allows immediate transformation to prioritized parallel code-constructs.

When `action` attributes are specified on parallel processes in a prioritized parallel relationship then these `action` attributes can be preempted. Any race hazards between shared variables in multiple `action` attributes must be prevented and therefore it is important that assignments can not update shared variables.

2.4.5 Prioritized Alternative Relationship

An alternative relationship between processes *P* and *Q* is denoted by the label '$\overleftarrow{\Box}$'. This prioritized alternative composition is written as $P\overleftarrow{\Box}Q$. This process is almost similar to the alternative relationship, except that when both processes can engage in a communication event then process *P* will be chosen in preference of *Q*. Process *P* is guarded with higher preference or priority.

We will use notation $(P,Q,\overleftarrow{\Box})$ to represent a prioritized alternative relationship between *P* and *Q*. See Figure 19.

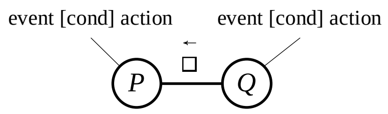


Figure 19. Prioritized alternative relationship.

A multiple relationship $(P,Q,R,S,\overleftarrow{\Box})$ represents $P\overleftarrow{\Box}Q\overleftarrow{\Box}R\overleftarrow{\Box}S$. See Figure 20.

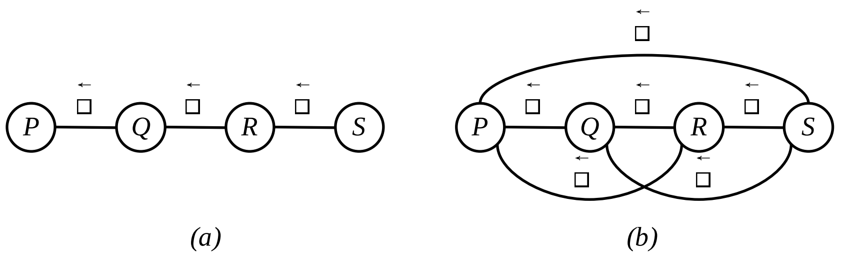


Figure 20. (a) Multiple prioritized alternative relationship and (b) multiple prioritized alternative relationship with derived relationships.

We mean by $(P,S,\overleftarrow{\Box})$:

$$(P,X,S,\overleftarrow{\Box})\backslash X$$

where *X* is a set of processes belonging to the longest path or main path between *P* and *S*, thus, $X = \{Q,R\}$. If *X* is empty then

$$(P,\{\},S,\overleftarrow{\Box})\backslash\{\} = (P,S,\overleftarrow{\Box})_\varnothing = P\overleftarrow{\Box}S.$$

In case of a multiple relationship we can write

$$(P,Q,R,S,\overleftarrow{\Box})_\varnothing = (P,Q,\overleftarrow{\Box})_\varnothing\,,\ (Q,R,\overleftarrow{\Box})_\varnothing\,,\ (R,S,\overleftarrow{\Box})_\varnothing = P\overleftarrow{\Box}Q\overleftarrow{\Box}R\overleftarrow{\Box}S$$

In the graph this expression is the longest path or main path between the processes *P* and *S* from *P* to *S*. Operator $\overleftarrow{\square}$ has no symmetry laws. The importance of this form is that we can index processes in the compositional construct so that processes are successively ordered and with declining guard priorities.
We can write:

$$(P_0,\ldots,P_{n\text{-}1},\overleftarrow{\square})_\varnothing = \overleftarrow{\square}_{i=0..n-1} P_i$$

This form allows immediate transformation to prioritized alternative code-constructs.

2.4.6 Exception (Interrupt) Relationship

An exception relationship between processes *P* and *Q* is denoted by the label '$\overrightarrow{\Delta}$'. This exception composition is written as $P\overrightarrow{\Delta}Q$. This exception relationship is not formally defined in CSP. The arrow specified the left-hand side and right-hand side of the operation.

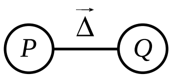


Figure 21. Exception relationship.

The exception operator originates from the interrupt operator $P\Delta_i Q$. This process behaves like *P* until *Q* can engage in event *i* at which point it behaves as *Q*. *Q* is initially awaiting for some event *i* from its environment. Event *i* is not in *P*'s alphabet.

Here, $P\overrightarrow{\Delta}Q$ and $P\Delta_i Q$ are both directed interrupt relations between *P* and *Q*, but $P\overrightarrow{\Delta}Q$ is *directional commutative*. The directional commutative property provides topographical modeling freedom. Consider the differences:

- $P\Delta_i Q \neq Q\Delta_i P$
- $P\overrightarrow{\Delta}Q = Q\overleftarrow{\Delta}P$

The operator $\overrightarrow{\Delta}$ has no symmetric laws.

The following two approaches describe ideas of how exceptions could be modeled in the CSP diagram. Approach 1 is closely related to the original semantics of the interrupt operator and approach 2 is a modified interrupt operator towards an exception operator.

Approach 1:

The triggering event *i* comes from a communication relationship with the exception handling process. The exception handling process *Q* is a guarded process that will be selected when communication event *i* is ready; process *Q* can be triggered on input or on output. Communication event *i* must be the first event in which *Q* will engage. Only one guard can be attached to the guarded process.

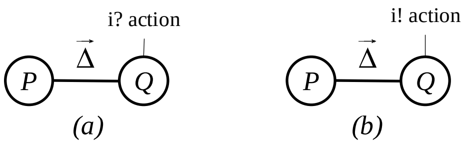


Figure 22. (a) Input-guarded interrupt process and (b) output-guarded interrupt process.

Approach 2:

In approach 1, event *i* is related to some communication event. In this approach, event *i* is related to the termination event of the process. An unsuccessful termination indicates an exception that is caused or thrown somewhere in the process. The process as rendered in Figure 23 behaves as *Q* when *P* unsuccessfully terminates; otherwise this process haves as *P*. If *P* successfully terminates then *Q* will be omitted. Note that the termination event is also not in *P*'s alphabet.

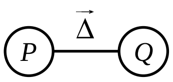


Figure 23. Exception handling based on unsuccessful termination.

If an exception is thrown in a process then this indicates an exceptional state and the process returns immediately with a message indicating the type of the exception. We can consider the exceptional state or throw action as an internal event that is invisible for the process. This internal event is then mapped onto the unsuccessful termination event. This simplifies the process description. Therefore, this approach does not require additional notations to render an exceptional state or a throw action. The actual exception message or exception handling is coded in a programming language and not in the CSP diagram.

A problem with approach 1 is that on the occurrence of an exception every channel that is used by *P* must be released. Approach 2 is much simpler than approach 1. In approach 2, if the channel is the source of the error then the channel may throw an exception at both sides of the channel. This way the erroneous process cannot lock a channel. Approach 2 can be implemented with try-and-catch clauses as found in Java and C++. Approach 1 may require a special and complex prioritized parallel implementation in order to perform pre-emption on event *i*.

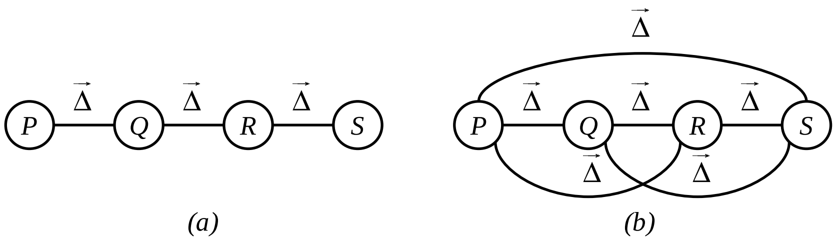


Figure 24. (a) Cascade of exception handling and (b) multiple exception relationships with derived relationships.

2.5 Parenthesized Compositional Relationships

As with algebraic compositions, the use of parentheses is inevitable to write more complex expressions. Parentheses are also required in this modeling language. Consider a model with three processes *P*, *Q* and *R* as shown in Figure 25. In this example, we specify that process *P* should be executed before *Q* and *Q* should be executed in parallel with *R*. The behavior between *P* and *R* is not specified and leaves open certain ambiguity. This means that there are more than one valid solutions and any of these solutions is accepted. Here, the valid solutions are $P;(Q\|R)$ or $(P;Q)\|R$.

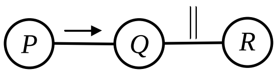


Figure 25. Example of a model with ambiguity.

Every solution should satisfy the requirements. If a solution exist that does not satisfy the requirements then further refinement steps are necessary in order to exclude this solution from the set of solutions. A unique and unambiguous solution can be achieved by specifying a relationship between *P* and *R*, as shown in Figure 26a and Figure 26b. Each cycle eliminates ambiguous interpretations.

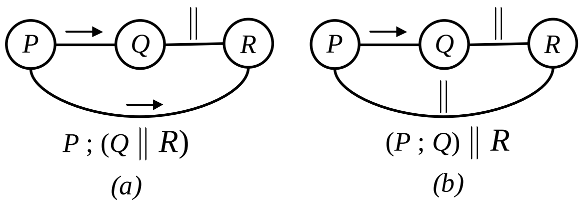


Figure 26. Unambiguous solutions using cycles of relationships (complete graph).

Imagine that for a large model any unique and unambiguous solution would require many relationships. All these lines would make the model complex and likely unreadable. In order to keep the model simplified, we introduce the parenthesis symbol on compositional relationships. This is represented by an open dot 'o' (concatenation of **(**+**)**) at the peer-end of the compositional relationship. For example, Figure 26a and Figure 26b are the equivalence of respectively Figure 27a and Figure 27b. Using parenthesizes symbols reduces the number of relationships.

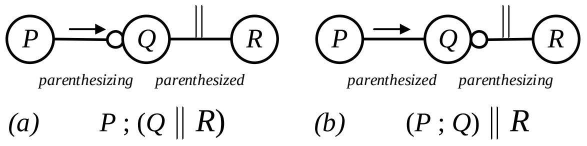


Figure 27. Unambiguous solutions using parenthesized relationships.

A compositional relationship without any dots is the strongest possible relationship. A compositional relationship with a dot at one end becomes a directed relationship. This is a *parenthesizing relationship*. A *parenthesized relationship* is a stronger relationship than its neighbor parenthesizing relationships which are directed to a process in the parenthesizing relationship. A chain of parenthesizing relationships can make one relationship stronger or weaker than the other. A stronger relationship is considered before a weaker relationship. These stronger/weaker relations are useful for describing anonymous groups or hierarchies of compositions.

2.6 Indexed Parenthesizing Relationships

A dot in the parenthesizing relationships can be indexed with an integer greater than zero. See Figure 28, were $i \in [1,\rightarrow)$. The index is an instrument useful for reallocating relationships in order to maintain the algebraic expression. This is discussed in Section 2.9.

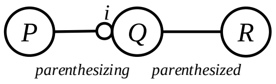


Figure 28. Indexed parenthesizing relationship with index *i*.

Indices greater than 1 should be rendered next to the dot to indicate the index. A dot with no index implicitly means that it has index 1.

2.7 Hidden Compositional Relationships

In reality, all processes are compositionally related to each other. In a CSP diagram, the compositional relationships that are specified by the designer are expressed by *visual connections* between processes. All other relations are *hidden connections*. We say that a process is unconnected or standalone when it is solely related by hidden connections. In the visual view it has no neighbors. Processes that are unconnected can be executed in any order, i.e., in parallel or in some sequence. By not specifying connections, we mean that we don't care what the execution order is and therefore we let the tool or environment decide. A hidden connection is an indexed parenthesizing compositional relationship, with parenthesis symbols at both ends of the line, as shown in Figure 29.

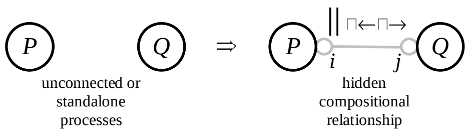

P
Q
⇒
P
Q
|| ⊓←⊓→
i
j
unconnected or standalone processes
hidden compositional relationship

Figure 29. Standalone processes and hidden compositional relationship

The choice between $P \parallel Q$, $P \,;\, Q$ or $Q \,;\, P$ is internally determined by the tool or environment. Of course, the solutions must be valid, i.e., each solution must be conflict-free. For example, Figure 30 illustrates a composition graph of four processes on the left side. Process *S* is standalone. The two graphs on the right side are showing the hidden connections of possible valid solutions that could be determined by the tool.

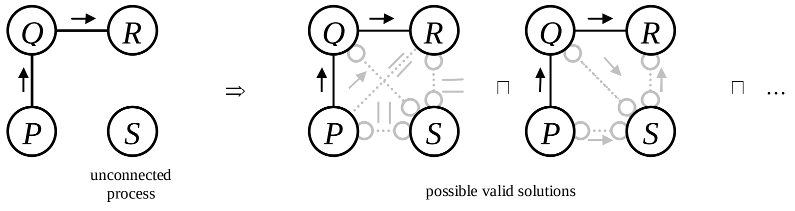


Figure 30. Example of hidden connections

The tool should choose the most optimal solution for the final implementation. The causality that is expressed by the data dependency between processes in the communication-graph can provide the necessary input in order to determine an optimal execution framework.

The hidden connections of the final solution can be visualized for understanding the generated framework at the same level of abstraction. See the gray lines in Figure 30. These hidden relationships can clarify reading the execution framework and code structures of the architecture in more detail. Normally, we are not concerned about hidden connections, but a CSP diagram allows us to express the detail of the execution framework. The ability of visualizing the hidden relationships is an ultimate solution for debugging and studying the behavior of the model at design level.

Showing every connection makes the graph easily unreadable. Fortunately, not every connection has to be shown. In Figure 30 the hidden connection $(P,R,\rightarrow)$ at the left-hand solution is derived from path $(P,Q,R,\rightarrow)$. This relationship is redundant and could be removed from the graph in order to make the graph better readable as illustrated in the second solution at the right-hand side. Also a tree-graph contains a minimized number of visible relationships without leaving information out.

2.8 Undefined Relationships

A connection between two processes can be specified without an operator or label. These undefined relationships can be useful for grouping processes together and combined with a parenthesizing relationship, i.e. being strong relationships. These undefined relationships become parenthesized relationships. As with hidden relationships, the operator of an undefined relationship can be determined by the tool or environment.

2.9 Reallocation Rules

In a process diagram, as in the CSP diagram, processes are usually located nearby the processes with the highest relationship density. The designer has the freedom to move processes around while the model grows. The connections between processes are usually kept short and crossings should be avoided as much as possible. However, reallocating processes can result in longer connections and possibly create crossings with other connections. In case a process is related to a group of processes, we present a technique that allows the process to be related to the nearest process in the group. The technique allows reallocating relationships with another (nearest) process in the group while maintaining the original relationship or original algebraic expression. Sometimes this technique can significantly shorten the connection or eliminate crossings, which make the model better readable.

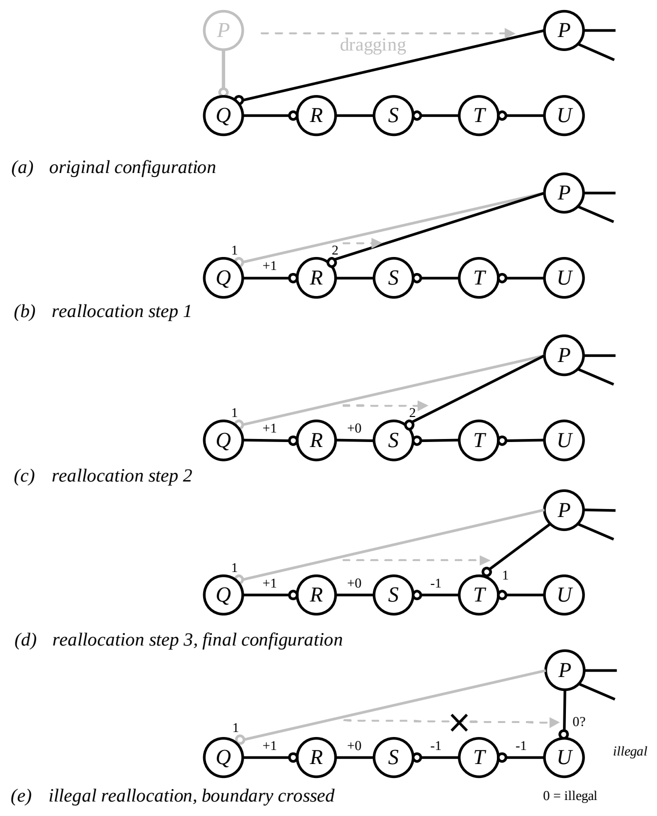

P
dragging
Q R S T U
(a) original configuration
1 2 +1
(b) reallocation step 1
+0
(c) reallocation step 2
-1
(d) reallocation step 3, final configuration
0?
illegal
(e) illegal reallocation, boundary crossed
0 = illegal

Figure 31. Example of reallocating a connection

Figure 31a shows a process *P* that is originally related to process *Q*, but it has been moved to another location in the diagram closer to other processes it is related to. These other processes are not shown in the figure. The technique presented here shows that the relationship between *P* and *Q* can be reallocated to a relationship between *P* and *T* as illustrated in Figure 31d. The steps are illustrated in Figure 31b-d.

The operators above the relationships don't really matter for this technique and therefore we will omit operators for the moment. We assume that the operators are conflict-free and that they satisfy the requirements. We use the general operator $\oplus$ and its complement $\overline{\oplus}$ to show the commutative properties.

Let operator $\oplus$ represent a binary CSP operator, where by $\oplus \in \{ \leftarrow, \|, \overleftarrow{\|}, \Box, \overleftarrow{\Box} \}$. Operator $\overline{\oplus}$ is the complement of $\oplus$, where by $\overline{\oplus} \in \{ \rightarrow, \|, \overrightarrow{\|}, \Box, \overrightarrow{\Box} \}$. These operators are directional commutative

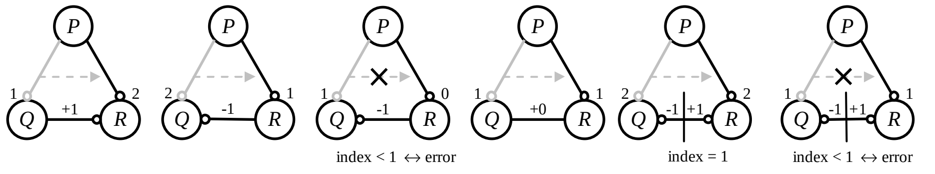
$$P \oplus Q = Q \overline{\oplus} P$$

For example, $P \| Q = Q \| P$, $P \Box Q = Q \Box P$, $P \rightarrow Q = Q \leftarrow P$, $P \overleftarrow{\|} Q = Q \overrightarrow{\|} P$, $P \overleftarrow{\Box} Q = Q \overrightarrow{\Box} P$.

The processes P and Q are relationship equivalent

$$(P, Q, \oplus) = (Q, P, \overline{\oplus}).$$

The dot represents an arrow head pointing to a direction. Each reallocation step along a compositional relationship represents an index increment, decrement, increment and decrement, or equality. Figure 32a-f shows six basic rules for reallocating relationships.



(a) incremental indexing *(b) decremental indexing* *(c) decremental illegal indexing* *(d) equal indexing* *(e) equal indexing* *(f) equal illegal indexing*

Figure 32. (a-f) Reallocation rules

Table 2 shows the equations of algebraic expressions for each basic reallocation.

Figure 32	Equation	
a	$P \oplus (Q \oplus (R)) = P \oplus ((R) \overline{\oplus} Q)$	
b	$P \oplus ((Q) \oplus R) = P \oplus (R \overline{\oplus} (Q))$	
c	$P \oplus (Q) \oplus R \neq P \oplus R \overline{\oplus} (Q)$	illegal
d	$P \oplus (Q \oplus R) = P \oplus (R \overline{\oplus} Q)$	
e	$P \oplus ((Q) \oplus (R)) = P \oplus ((R) \overline{\oplus} (Q))$	
f	$P \oplus (Q) \oplus (R) \neq P \oplus (R) \overline{\oplus} (Q)$	illegal

Table 2. Reallocation equations

In Figure 32 the rules c and f illustrate the boundaries of reallocation. Once a relationship is given a parenthesis symbol then its index is 1 or higher. Illegal indexing (index < 1) indicates an illegal reallocation in that direction and indicates a dead end. In Figure 32c and Figure 32f it is trivial to see that process *R* is not a member of the group and therefore reallocation should not be applied.

In the example of Figure 31, the connection $(P,Q,\oplus)$ has been reallocated to $(P,T,\oplus)$. The connection $(P,T,\oplus)$ is the shortest connection. Although the distance between *P* and *U* is shorter, no reallocation with *U* is possible. See Figure 31e. This reallocation is prohibited according to the rule as expressed in Figure 32c. The index may not go below 1.

The algebraic expressions of Figure 31a and Figure 31d are equal:

$$P \oplus (Q \oplus (R \oplus S) \oplus T) \oplus U = P \oplus (T \overline{\oplus} (S \overline{\oplus} R) \overline{\oplus} Q) \oplus U$$

The algebraic expression of $(P, U, \oplus)$ results in the inequality:

$$(P \oplus U) \oplus (T \overline{\oplus} (S \overline{\oplus} R) \overline{\oplus} Q) \neq P \oplus (Q \oplus (R \oplus S) \oplus T) \oplus U$$

This method can be automated. For example, while dragging processes around the CSP diagram, the tool could automatically reallocate connections to sustain the shortest connections according to these rules.

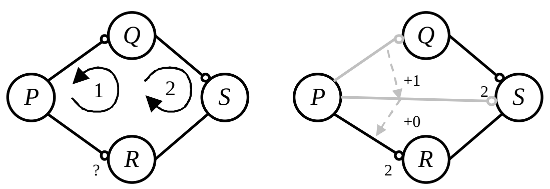
2.10 Balanced and Unbalanced Parenthesized Cycles

Cycles of parenthesizing relationships in a design should be *balanced*. This means that in a cycle the number of parenthesizing relationships pointing in one direction should compensate the number of parenthesizing relationships pointing in the other direction. The counting starts and ends in one process of the cycle. If these parenthesizing relationships do not compensate opposite parenthesizing relationships in the cycle then one cannot completely determine the algebraic expression of this co-called *unbalanced cycle*. In an unbalanced parenthesized cycle, the algebraic expression reasoned in one direction is not the same as the algebraic expression reasoned in the other direction. A balanced cycle results in a single algebraic expression reasoned from both directions.

Figure 33a illustrates an unbalanced cycle of parenthesizing relationships. We start from a process from where all parenthesizing relationships (as part of the cycle) leave. This is process *P*. The number (or weight) of parenthesizing relationships pointing clockwise is 2 and the number of parenthesizes pointing anti-clockwise is 1. After subtraction the result is 2-1=1 whereas a balanced cycle should be 0. In this example, it is not difficult to see that the index of the parenthesizing relationship between *P* and *R* should carry index 2.

There is a systematic approach that is useful for determining the correct index. Two paths in a cycle between two processes provide redundancy which is useful for this technique.

We start with the process from where all parenthesizing relationships, that are part of the cycle, leave the process. This is process *P*. We select the relationship leaving *P* that belongs to the path with the highest weight. Here, this is relationship $(P,Q,\oplus)$, because the clockwise direction has weight 2. See Figure 33a. This relationship should be reallocated to the other relationship that is leaving process *P* and is part of the cycle. This is relationship $(P,R,\oplus)$. This is illustrated by Figure 33b and according to the reallocation rules the index should be 2. Now, the weights of both directions are 2 and the total is 2-2=0 Thus, the graph has been balanced. If, during the reallocation steps, the index was lower than 1 then there is at least one more index in the cycle that is wrong or a patch with not the highest weight was selected.



(a) unbalanced cycle of parenthesizes *(b) balanced cycle of parenthesizes*

Figure 33. (a) unbalanced and (b) balanced cycle of parenthesizes

This technique should be carried out for each cycle. In case two connected cycles determine two different indices for a shared relationship then the model has a structural error. The model needs to be revised.

2.11 Hierarchies of Processes

A process can contain other processes. The CSP diagram allows constructing hierarchies of processes. Each hierarchy contains a sub-diagram, i.e., a sub-communication-graph and a sub-composition-graph. The parenthesis symbol or open dot defines a grouping of processes which in turn forms a nameless or anonymous process. The open dot can be given an index which value determines the range of the parenthesis. The index is determined by the reallocation rules (as described in Section 2.9) or by grouping processes into hierarchies.

The index of a parenthesizing relationship can be 1 or higher. Determining the index between a process and a hierarchy of processes starts with selecting one process in the group that has only outgoing parenthesizing and/or strong relationships. If we have found such a process we can connect the parenthesizing relationship with index 1 to that process. The hierarchy is made. The reallocation rules will determine the height of the index of the parenthesizing relationship with a particular process in the hierarchy. Reallocation rules do not change the hierarchies. Illegal indices (index < 1) are only possible unless a wrong starting process was selected that has an incoming parenthesizing relationship. Figure 34a depicts a relationship with process *P* and *Q*. Figure 34b shows the relationships of *P* with the sub-processes of *Q*. Process *R* has only outgoing relationships and this is where the reallocation starts.

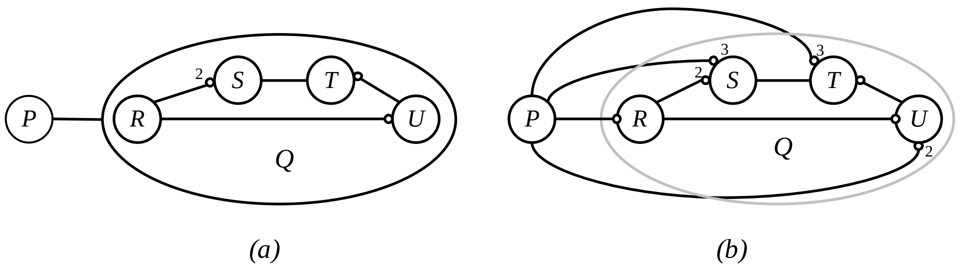


Figure 34. (a) relationship with a parent process (b) relationship with a child process and applying the reallocation rules.

The reverse procedure is also interesting and shows the existence of nameless or anonymous processes. The reverse procedure can also be used to determine design conflicts in more complex designs. These anonymous processes can be useful for structuring software into clauses. The method to determine these clauses or anonymous hierarchies is described below. All processes connected with strong relationships form a group. For example, the processes *S* and *T* in Figure 35 are therefore merged into an anonymous process, which we will identify as *ST*. See step *I*. The relationships are reconnected to the anonymous process and the indexe(s) must be decremented by 1. We can continue applying the same technique until a single process and only strong relationships remain. See Figure 35b. Follow steps *II*..*IV*.

In this example, relationships $(R,ST,\oplus)$ and $(R,U,\oplus)$ in Figure 35a can be merged only if both relationships have the same operator. See Figure 35b (step *II*). If these relationships have different operators then these operators are in conflict and the model has an error. This procedure allows to check the model for errors of conflicting operators, i.e., detecting compositional conflicts. This procedure ends until a single process is the result (step *IV*) and all parenthesized symbols have been eliminated. All figures in Figure 34 and Figure 35 represent the same solution.

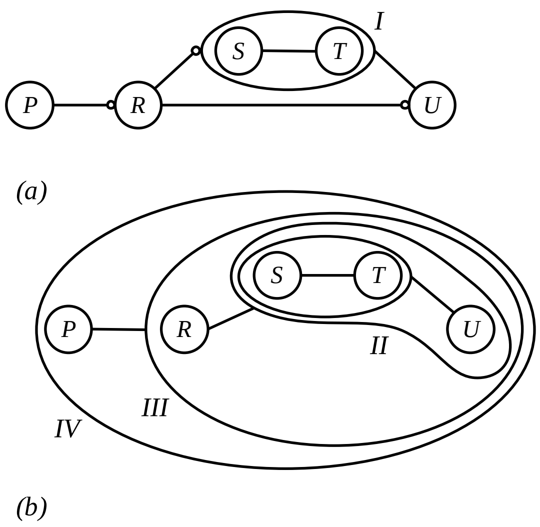


Figure 35. (a) single step merging S and T (b) all steps until a single process results.

2.12 Compositional Conflict Checking Technique

In previous sections we have seen that the CSP language provides several features to determine compositional conflicts in design and requirements. We describe a systematic approach that will combine these features to find compositional conflicts.

Definition (Compositional conflict). A compositional conflict is a failure of two compositional relationships that are in contradiction.

The systematic approach basically transforms the design to a normal form. The procedure of creating a normal form is the basis for compositional conflict checking. The systematic approach of finding compositional conflicts is based on reallocating redundant relationships temporarily on top of each other. Both relationships should have the same operator and the same index otherwise they are in conflict. If the operators match and the indices match then one relationship can be eliminated from the graph. Hierarchies of processes are transformed into parenthesizing relationships. This procedure ends when no more cycles exist and the model is totally flattened into a tree-based model. The resulting model is called the *normal form*.

This is a typical operation a tool can perform to check if the designer is applying a correct operator or index for each added relationship. During design the tool could assist the designer by showing warnings when operators or indices are in conflict. These warnings will become errors before code generation, because the model cannot be code generated.

2.13 Recursion based on Named Hierarchies

The ability of modeling process hierarchies enables creating recursive constructs. For example, an infinitely recursive process

$$X = P;X = P;P;P;P;\ldots$$

could be depicted as in Figure 36.

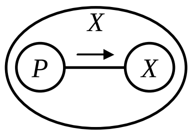


Figure 36. Infinite recursion.

A finite recursive process can be specified by a finite number *n*, as in

$$X_i = P;X_{i+1} = P;P;P;P;\ldots \qquad i \in [0..n\text{-}2]$$

$$X_{n\text{-}1} = SKIP$$

This is depicted in Figure 37.

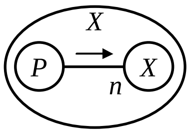


Figure 37. Finite recursion of n times.

A finite recursive process can also be conditional at run-time, like

$$X : Q \lhd (cond1) \rhd X$$

Representing: `X: if (cond1) then Q else X`, which is depicted in Figure 38.

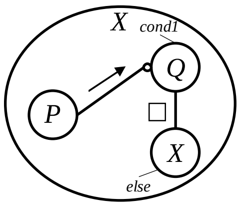


Figure 38. Finite conditional recursion

An important problem with the above mentioned recursion processes is that the communication diagram requires channels to both *X* processes. Both *X* process suppose to have the same process interface of channel-inputs and/or channel-outputs. The communication diagram will become too complex if they have any channels attached to them. Therefore *this Graphical Modeling Language does not support named recursion.* A better solution is given in Section 2.14.

2.14 Recursion based on Anonymous Hierarchies

A process can represent 'nameless' recursion involving *X*, as in

$$\mu X\bullet(P;X) = P;P;P;P;\ldots$$

This is depicted by a special μ-process as shown in

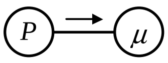


Figure 39. Infinite recursion.

The μ-process is a leaf process that can participate in only one compositional relationship with one other process. The μ-process is different from other processes. Firstly, its interface is decoupled from any other process interface and therefore the μ-process overcomes the interfacing problem as described in Section 2.13. Secondly, no matter what kind of compositional relationship is specified, it will always repeatedly execute the other process until some additional conditional expression becomes false. The conditional expression is evaluated in the order that is specified by the compositional relationship. This also gives rise to a dynamic form of recursion – or a better word is *looping*.

A few different kinds of loops are shown below that can also be found in many programming languages. These loops can be modeled with the μ-process.

DO-WHILE construct 1:

```
int i = 0;
do { i++; P; } while [i<10];
```

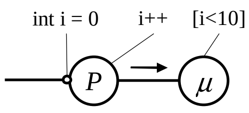


Figure 40. DO-WHILE construct 1.

DO-WHILE construct 2:

```
int i = 0;
do { P; } while [i++<10];
```

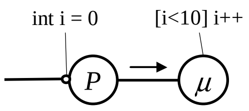


Figure 41. DO-WHILE construct 2.

WHILE or FOR construct:

```
int i = 0;
while [i<10] { i++; P; }
```

or

```
for (int i=0; i<10; i++) { P; }
```

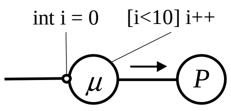


Figure 42. WHILE or FOR construct.

The μ-process can also be used with other compositional relationships other than sequential. For example, Figure 34 shows three processes in parallel:

$$P[0] \parallel P[1] \parallel P[2]$$

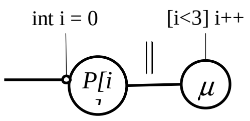


Figure 43. Parallel recursion.

Other applications of using the μ-process are omitted for now.

2.15 Primitive Communication Processes

All CSP relationships in the communication diagram and in the composition diagram are all synchronization points in the design model. This section introduces three communication primitives that express synchronization points in the design model. We model these synchronization points as primitive communication processes. These primitive processes are depicted in Figure 44.

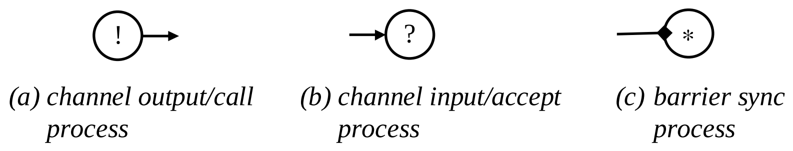

!
?
*
(a) channel output/call process
(b) channel input/accept process
(c) barrier sync process

Figure 44. Primitive processes.

The termination events of these processes correspond to their shared communication events. Figure 44a illustrates a channel-output or a channel-call. Figure 44b illustrates a channel-input or a channel-accept. Figure 44c illustrates a barrier synchronization entry on a shared barrier.

These primitive processes are useful for

- showing the points of interaction between processes,
- showing hardware access points [11],
- checking for deadlocks in design,
- and checking for priority inversion problems in design.

A shared data-channel with many reader and/or writers is basically an alternative construct and not a broadcasting construct. Broadcasting with data-channels is possible by using the special delta process as shown in Figure 45a.

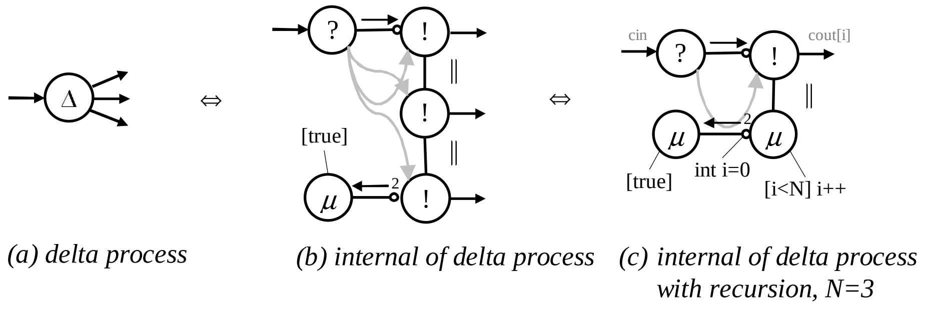

Δ
⇔
?
!
!
!
[true]
μ
2
||
||
⇔
cin
cout[i]
?
!
μ
μ
2
||
[true]
int i=0
[i<N] i++
(a) delta process
(b) internal of delta process
(c) internal of delta process with recursion, N=3

Figure 45a-c. Delta processes for broadcasting messages.

The delta process will read from the single input-channel and outputs copies of the data to each output-channel. The delta process will terminate after all data has been send. Call-channels and barrier constructs are not supported by the delta process and one has to create a dedicated broadcasting process as shown in Figure 45b-c.

2.16 *Deadlock Analysis using Compositional Relationships*

Compositional conflicts in a CSP diagram found between the primitive communication processes reflects deadlock.

Definition (Deadlock). A deadlock is a failure of two processes to cooperate with each other because of not being able to agree on a common event, although they are willing to participate in other events.

A good solution in finding deadlocks and other phenomena is using formal deadlock checkers. For example, the model could be translated into readable CSP and analyzed by a tool like FDR [12]. The tool will proof if the design is deadlock free. This is only possible if the model is conflict-free, but not necessarily deadlock-free.

During design it would be convenient to detect and to warn about the presence of potential deadlocks before finishing the model. Here, we describe a technique for finding and for reasoning about deadlocks in the design phase of the project. This is based on the conflict-free checking technique involving these primitive communication processes.

A compositional conflict is an error in the design. As a result of a compositional conflict the model cannot be code generated—no solution can be found. If the model is conflict-free then there might still arise a type of conflict which we call *deadlock*. Deadlock is a synchronization conflict in software that occurs at run-time. Potential deadlock can be traced in the model before run-time and before code generation. The primitive communication processes in Figure 44 play an important role in analyzing the model for potential deadlocks.

For example, Figure 46 shows a model that is conflict-free. The model can be code generated and executed.

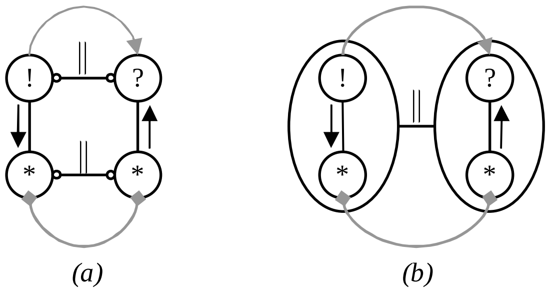


Figure 46. (a) original design, (b) compositional conflict-free.

At run-time, these processes synchronize on channel communication or on barrier synchronization and they maintain in a locked state forever—they deadlock. In the procedure of finding conflicts we define a preliminary step that allows us to detect potential deadlocks. Given the fact that a channel-input/call/sync and channel-output/accept/sync synchronize at both sides always rendezvous, we can consider the channel-input/call/sync process and the channel-output/accept/sync process as a single *rendezvous process* (i.e. in the form of an anonymous process). The relationships between the primitive processes must be parallel or prioritized parallel. See Figure 47.

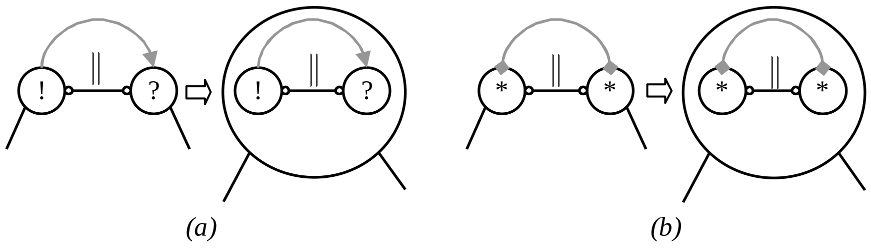


Figure 47. (a) channel comm. to rendezvous process (b) barrier comm. to rendezvous process.

It is necessary that the model is flattened, i.e., all processes are brought to the same level in hierarchy. Thus, the first step is to determine all rendezvous processes. Successively, the procedure of finding conflicts should be applied in order to find *sequence conflicts* in every path between these rendezvous processes. A sequence conflict is a compositional conflict where by sequential operators are in contradiction. For example, Figure 47a is conflict-free but not deadlock-free. First we merge the pair of primitive communication processes together into rendezvous processes and watch the compositional relationships between the rendezvous processes. One can see that the remaining compositional relationships are in contradiction. This sequence conflict is a potential deadlock. This is the same for barrier synchronization.

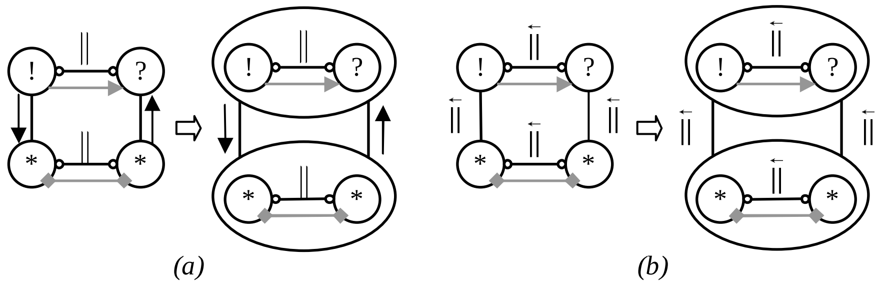


Figure 48. (a) sequence conflict between rendezvous processes = deadlock, (b) priority conflict between rendezvous processes = priority inversion problem.

2.17 *Priority Inversion Analysis using Compositional Relationships*

With the same technique as in the previous section one can find priority conflicts whereby prioritized parallel operators are in contradiction. See Figure 48b. In this case, the model suffers from *priority inversion*. It is likely that the performance of the higher priority process is pulled down by the lower priority process and as a result of that it may that the deadlines of the higher priority process cannot be met.

Usually, eliminating this design conflict will result in a better design. In case priority inversion is inevitable in the design, which is possible, one could use a special channel between processes executing at different priorities in order to delay blocking. A simple buffered channel can help solving priority inversion problems. The prioritized parallel relationship in the composition-graph that is in conflict *together* with the communication relationship in the communication-graph between the same processes specifies the location of a buffered channel. A tool could use this information to change rendezvous channels into buffered channels. This information together with the direction of communication and the frequency of the processes can be used to determine special buffers, like oversampling and subsampling buffers. Notion of time is not discussed in this paper and is subject to further research.

2.18 *Refinement and Verification*

In case the design is incomplete, the tool may likely generate a framework that does not satisfy the desired requirements. The behavior of a CSP diagram can be studied or verified by several systematic approaches:

- One can determine inconsistent relationships between processes by browsing the model and checking all the paths of relationships and adding derived relationships between non-neighbor processes on these paths. These relationships must satisfy; otherwise inconsistency is found which causes an invalid solution.
- Additional to the previous approach and to reducing the number of relationships, visual and hidden relationships with equal operators could be rendered in a layer apart from other layers. Each layer can be studied independently in order to verify the model and to study layers of requirements.
- The design can be simulated whereby sequences of events can be studied in an event trace window.
- The model can be translated to readable CSP and a special tool, like FDR, can analyze the behavior and produces traces of particular events.

The information that is provided by the CSP diagram allows verifying the design prior to executing the code and seeing whether it works or not. This makes any *trial-and-error* approach in early stages in the software development unnecessary.

In case the design does not meet the requirements, additional relationships must be specified—the model needs to be refined. The designer will undertake refinement steps until the requirements are achieved. The refinement and verification approach is a continuously interactive process.

Usually, the final model consists of one or more tree graphs. Necessarily, refinement steps may continue by adding cycles. Cycles can introduce redundancy which can help detecting conflicts between relationships and the requirements. The ultimate refinement step results in a complete composition-graph whereby all processes are visually connected to each other. The density of connections in a complete composition-graph makes the model complex and difficult or impossible to read. Although, a complete composition-graph has a unique solution, usually, a unique solution is not the goal of the designer. Any valid solution that satisfies the requirements is adequate.

Adding or removing redundant relationships is an intuitive process that depends on the complexity of the problem domain, the requirements, and the analysis approach. Complexity can be managed using parenthesizing relationships that reduce the number of cycles and bring about tree graphs. Of course, applying hierarchies of sub-graphs processes likely simplify the design, improve readability, and simplify the refinement and verification process.

3 Companionship between Communication and Composition

In Section 2.3 was mentioned that an abstract data type determines a data-channel and a data-channel can be one out of several possibilities, namely rendezvous, buffered, or variable. The compositional relationship is orthogonal to the communication relationship and both relationships between the same processes determine the appropriate channel. The possible configurations for data-channels are depicted in Figure 49a-f. Here, a communication relationship and compositional relationship are simultaneously drawn in the one figure. Each configuration optimizes communication in order to increase throughput, eliminate deadlock, or to prevent priority inversion problems. Explaining these kinds of optimization for each configuration requires additional pages and thus these configuration are not further discussed in this paper. Figure 49g-i shows a few illegal or unsupported configurations.

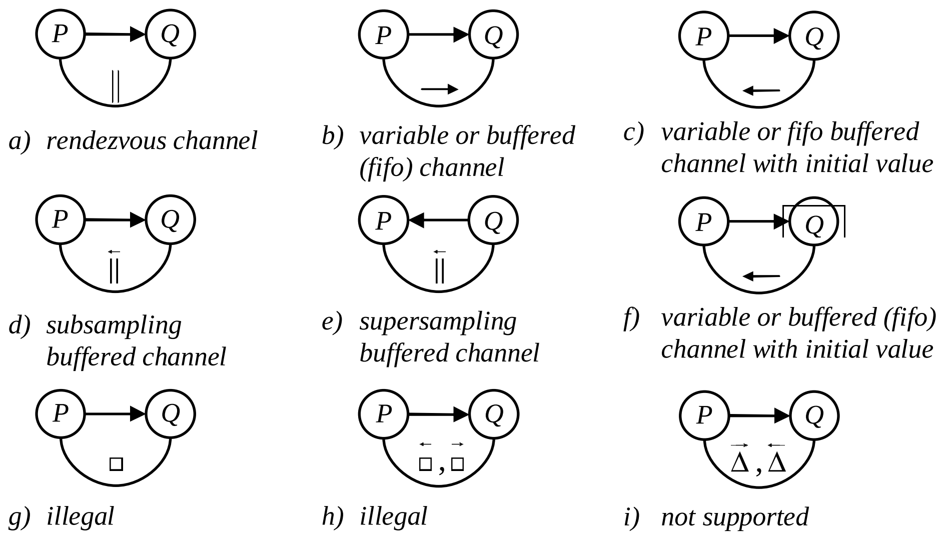

P
Q
a) rendezvous channel
b) variable or buffered (fifo) channel
c) variable or fifo buffered channel with initial value
d) subsampling buffered channel
e) supersampling buffered channel
f) variable or buffered (fifo) channel with initial value
g) illegal
h) illegal
i) not supported

Figure 49a-i. Data-channel configurations.

Call-channels and barrier only exist in a parallel relationship or in prioritized parallel relationship, as shown in Figure 50a-b. Currently, no optimizations as with data-channels are possible for call-channels and barriers.

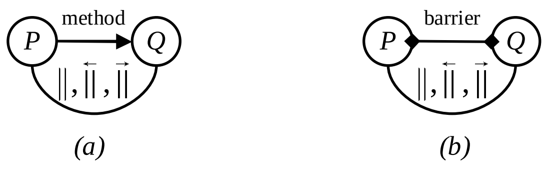


Figure 50. (a) call-channel configuration, (b) barrier configuration.

4 Design Freedom

The design methodology using CSP diagrams embraces significant freedoms during the development of software.

1. The designer can leave relationships between processes undefined when any order of execution is accepted. Usually, the execution order can be derived from the causalities as expressed by the communication graph.
2. The design model can be further refined by adding relationships where by redundant information is allowed. Of course, redundant relationships can also be removed. In other words, one can over-specify the design to be sure that the requirements are met. Over-specifying does not make the software more complex.
3. The run-time environment (kernel) takes care of the non-deterministic behaviors of the (prioritized) parallel and (prioritized) alternative relationships. The software becomes truly event-driven.
4. The designer can influence the framework of code generation by defining compositional relationships between processes. The framework will adapt to the desires of the designer. The designer is no longer restricted to a predefined framework.
5. The tool can frequently check the model for design failures, like compositional conflicts. During design, design failures are not treated as errors, but as warning. The tool can highlight the path(s) in which a particular failure occurs. This could be done similar as a word processor that underlines the incorrect words or suggest other grammar. In short, the tool can guide the designer to improve the model without restricting the design freedom. In the final model any remaining failures are considered as errors.

5 Conclusions

Designing CSP diagrams provides a new way of designing concurrent software. A CSP diagram represents the blueprint of an execution model of a concurrent-software-architecture. The presented graphical modeling language acts as a glue-logic between structured methods and object-orientation, because it provides continuation between the two paradigms. A CSP diagram is UMLable and can be presented as a new process diagram for the UML to capture concurrent, real-time, and event-flow oriented software architectures. Processes and their relationships can easily be implemented using objects.

For example, the CSP for Java packages CTJ [13,14] and JCSP [15] can be used to implement CSP diagrams.

A CSP diagram is not responsible and not detailed enough for the entire coding, but is mainly responsible for the architectural execution framework. Each process can be further detailed using other diagrams (e.g., state-charts, UML diagrams) and other tools.

This graphical modeling language can be used at every level of abstraction with the same graphical notations and semantics. The design freedom is high and the design process is guided by simple rules and semantics that can guarantee consistency and correctness. A CSP diagram can be mathematically analyzed, checked, simulated, and finally executed on a dedicated embedded real-time system. Design tools are required to support this graphical modeling language so that a software architect can really benefit from CSP diagrams. This paper introduced the basics of the graphical modeling language and further research is required to enhance the graphical modeling language and to build tools for designing CSP diagrams and for code-generation.

References

[1] C. A. R. Hoare (1985). *Communicating Sequential Processes*. London, UK, Prentice-Hall.

[2] A. W. Roscoe (1998). *The Theory and Practice of Concurrency*, Prentice-Hall.

[3] P. T. Ward and S. J. Mellor (1985). *Structured Development Techniques for Real-Time Systems*. Englewood Cliffs, NJ, Prentice-Hall.

[4] D. J. Hatley and I. A. Pribhai (1987). *Strategies for Real-Time System Specification*. New York, NY, Dorset House Publishing.

[5] E. N. Yourdon (1989). *Modern Structured Analysis*. Englewood Cliffs, NJ, Prentice Hall.

[6] P. B. Douglass (1998). *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, Massachusetts, Addison Wesley Longman, Inc.

[7] G. Booch, J. Rumbaugh, et al. (1999). *The Unified Modeling Language -- User Guide*. Reading, Massachusetts, USA, Addison-Wesley.

[8] B. P. Douglass (1999). *Doing Hard Timer: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Reading, Addison Wesley Longman, Inc.

[9] A. W. Roscoe (1987). *Routing messages through networks: an excercise in deadlock avoidance*. 7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines, Grenoble, LGI-IMAG.

[10] G. Jones (1987). *On Guards*. 7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines, Grenoble, LGI-IMAG.

[11] G. H. Hilderink, J. F. Broenink, et al. (1998). *Software design method for heterogenous embedded systems*. 17th Benelux Meeting, Mierlo, NL.

[12] FDR, Formal Systems Ltd., *FDR2*, http://www.formal.demon.co.uk/

[13] G. H. Hilderink, *Communicating Threads for Java (CTJ) home page*, http://www.ce.utwente.nl/javapp

[14] G. H. Hilderink, J. F. Broenink, et al. (1999). *Communicating Threads for Java*. Proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, Keele, UK, IOS Press.

[15] P. H. Welch and P. D. Austin, *The JCSP home page*, http://www.cs.ukc.ac.uk/projects/ofa/jcsp