Graphical modelling language for specifying concurrency based on CSP

G. H. Hilderink

Abstract: A graphical modelling language for specifying concurrency in software designs is presented. The language notations are derived from the communicating sequential process (CSP) language and the resulting designs form CSP diagrams. The notations reflect both data-flow and control-flow aspects of concurrent software architectures. These designs can automatically be described by 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 modelling languages and structured methods.

1 Introduction

Real-time and embedded systems require safety, reliability, robustness, and the guarantee that its processes meet their deadlines. Those systems are inherently concurrent since they have to communicate, react, and respond to a concurrent world. These requirements complicate the artefacts of software engineering; analysing, designing, implementing, and testing become more complex. It becomes even more complicated if we abandon concurrency and end up with a sequential design and implementation. Multithreading makes things even worse since it is too artificial, too low-level and too fine-grained to reason about concurrency in a natural way. This is common practice when we use sequential programming languages (e.g. C, C++, Java) and object-oriented modelling languages, like the UML. These languages are simply not properly equipped to model concurrency in an easy and elegant way. In order to create control software that fulfils the above requirements we move away from multithreading to a higher level of abstraction that is closer to the way we think about concurrent systems, which is the communicating sequential process (CSP) language. CSP provides concepts that are mathematical sound and without surprises. These concepts are very practical and useful for software and for hardware engineering, and are an excellent basis for hardware/software co-design. For instance, parallel programming languages like Ada and occam are based on CSP and they show that CSP concepts are very useful for creating real-time and embedded software. Furthermore, occam programs can be mapped directly on hardware. Thus, CSP can be applied for software and for hardware. Despite the high quality of these secure programming languages, these languages and the CSP concepts are not well-known by the majority of software engineers. There is no modelling language on top of these programming languages that supports CSP at higher levels in design and analysis that immediately shows the benefits of using CSP. In order to make these concepts useful for designing concurrent systems, we developed a graphical modelling language that supports these concepts and thus one can benefit from the use of CSP. Ada and occam can implement these designs almost directly. Also sequential programming languages (e.g. C, C++, Java) can implement these designs but in these cases a CSP library is required. Such a CSP library consists of a small set of design patterns of CSP primitives that encapsulate multithreading in a simplified and secure way. The result will be the use of threads without directly programming with threads.

CSP is a process algebra for describing and analysing concurrent systems [1, 2]. CSP embraces formal mathematics so we can specify requirements precisely and prove that they are satisfied by our implementations. However, a process algebra, such as CSP, in its textual form is not a pleasant modelling language for developing software. Usually, we prefer a graphical modelling 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 analysing (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 modelling issue. This is why the CSP essentials are so important for developing concurrent software.

A graphical modelling language is now introduced for specifying concurrency in software design. The language

IEE Proceedings online no. 20030132

DOI: 10.1049/ip-sen:20030132

Paper received 17th June 2002

The author is with Control Laboratory, Department of Electrical Engineering, University of Twente, The Netherlands

notations are derived from CSP and the resulting designs form CSP diagrams. The notations reflect both data-flow and control-flow aspects, 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 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 discussion of these topics can be found in Sections 2.14 and 2.15. The CSP diagram collaborates with object-oriented and structured methods [3–5]. The CSP diagram is UMLable and can be presented as a process diagram for the UML [6–8] to capture concurrent, real-time, and event-flow oriented software architectures. The extension to the UML will not be discussed in this paper but it is a topic of further research.

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.

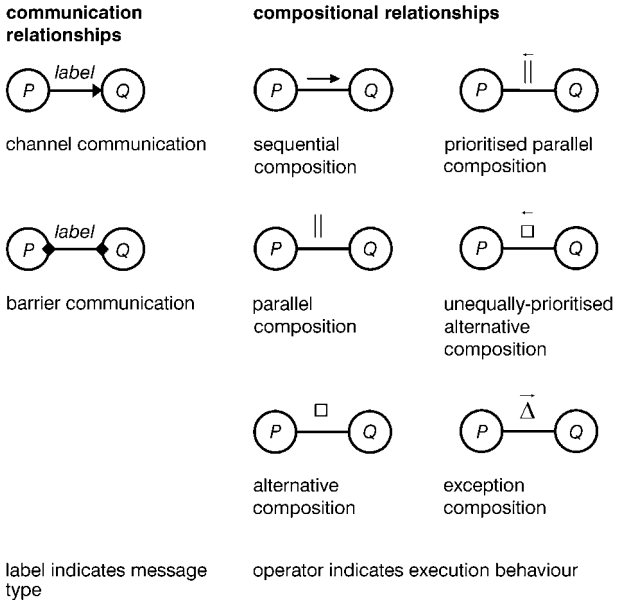


Fig. 1 *Overview of CSP relationships*

2 The CSP diagram

A CSP diagram is a graph of processes and their inter-relationships. 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. A mathematical analysis in CSP is outside the remit 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 prioritised primitives to deal with priorities in software. These are similar primitives to those 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, bubbles, 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 a *communication diagram* and a *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 Fig. 1.

Data-flow models have a great potential to specify parallelism with deployable processes and communication between processes. Unfortunately, commonly those models are not capable of expressing concurrency. *Concurrency* reflects the entire parallel, sequential, and alternative behaviours of a (sub)system which allows us to study the behaviour (e.g. reactiveness, responsiveness, synchronisation, timing, deadlock, livelock, starvation) of the (sub)system. Many data-flow languages, such as those found in structured methods, do not specify whether processes run in parallel or in some sequence. Instead, the semantics of the execution framework of a data-flow diagram is decoupled from its implementation. What originally seemed to be a benefit is what we now consider to be a very important disadvantage of data-flow modelling since no necessary changes can be enforced to the execution framework by the designer. For example, for 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 for 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 modelling (in a different layer) for specifying those aspects that are not expressed by data-flow diagrams that consist solely of nodes and arrows. The communication diagram expresses a basic form of data-flow modelling. This language illustrates that data-flow modelling is very intuitive in the capture of concurrency in systems with the use of CSP concepts.

Besides the graphical notations, the language inherently supports a few basic techniques for checking the correctness of the composition diagram. With these modelling 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 as such 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 intermediary communication objects, that 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; 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 labelled rectangle or labelled bubble in the graph.

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 to 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. Each relationship between processes itself forms a process. We discuss 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 nor communicate with each other, they are always mutually related to each other since they share the same world. The simplest relationship is *concurrency* whereby processes execute independently of each other. Concurrency involves synchronisation between processes. Processes execute and synchronise in many ways, which can be expressed by relationships between the processes. The interrelationship is depicted as a labelled line (or edge) between two processes as shown in Fig. 2. Additional symbols can be attached to the peer-ends of the line to indicate a directional relationship.

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

It is important to note that the line should be seen as distinct from the symbols that can be attached to each end of the line. The line renders an event in which both processes can engage. 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 polarisation 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*. A communication relationship is a labelled and directed relationship, which represents

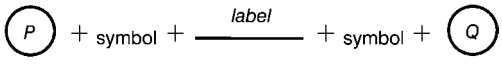


Fig. 2 *Inter-relationship between processes*

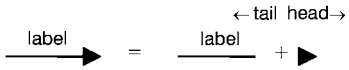


Fig. 3 *Communication relationship*

message flow between a sender process and receiver process.

A communication relationship is symbolised by an arrow between two processes as depicted in Fig 3. The arrow symbolises a channelled message passing between processes in a communication diagram.

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 are 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 as described in Section 2.13.

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 Fig. 4*a*. 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:

- rendezvous channel;
- buffered channel (fifo, supersampling, or subsampling);
- unsynchronised channel or variable.

The compositional relationship is orthogonal to the communication relationship, but there is also an invaluable relationship between the two. This relationship between the two diagrams enables formal techniques to determine appropriate channels that can improve the performance of the software architecture in a reasonable way.

A method name specifies a client-server relationship. The client process invokes the method and it is executed

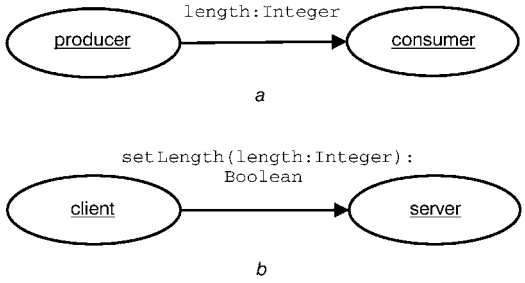


Fig. 4
a Producer–consumer
b Client–server

when the server accepts the call. Both participating processes must rendezvous whereby the processes are willing to engage at the same time, i.e. one process calls the method on the call-channel and the other process calls accept on the call-channel. 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 returns true or false to the client. See Fig. 4*b*. 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 unsynchronised calls between processes are not supported.

The process' 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 optimised for low-level communication. Call-channels are higher-level communication primitives and are used when the data-channel interface is too restrictive. Data-channels can efficiently establish communication through hardware. Their abstract and primitive interface provides hardware independency. Data-channels are very efficient and simple to use for building data-driven applications like control systems.

The previous communication relationships express unconditional communications, i.e. if both participating processes are ready for communication then they are committed in communication. They 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 carry 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 diagram.

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 Fig. 5.

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.

The configuration displayed in Fig. 5*a* depicts channel communication between two processes. The configurations in Fig. 5*b* and 5*c* implement a non-deterministic choice of service between multiple writers. The configurations in Fig. 5*c* and 5*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

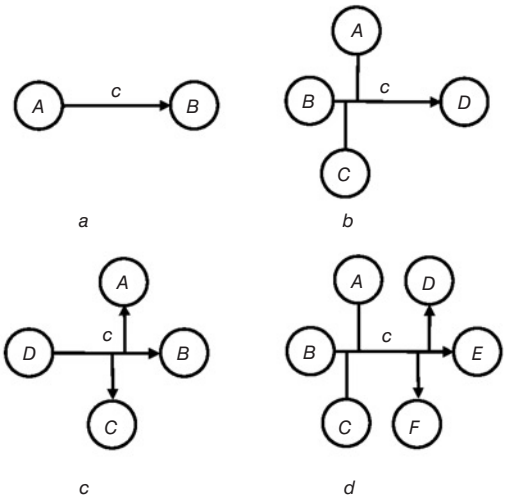


Fig. 5 *Channel configurations*
a One2One
b Any2One
c One2Any
d Any2Any

time and other readers have no chance 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 shown in 5*b*, 5*c*, and 5*d* swap to the configurations in 5*a* with alternating fairly reader and/or writer processes. The access time and the priority are important parameters for fair scheduling.

2.3.1 Barrier: Another communication primitive is the barrier synchronisation primitive. A fixed number of processes is required to synchronise their execution at some point before proceeding. A barrier is depicted as a bidirectional communication relationship whereby each end is symbolised with a diamond '◆' symbol (concatenation of ◀+▶). See Fig. 6.

When all processes reach the barrier synchronisation then the barrier construct communicates information between the processes in a unidirectional or bidirectional way. A barrier communication event can be multi-way, i.e. more than two processes can engage in the same event. All processes continue after communication is performed. An example of six processes that synchronise on a single barrier is depicted in Fig. 7.

A barrier synchronisation pattern could be described in terms of a protocol of channel communications [9]. The point of synchronisation is not rendered by this relationship. The actual point of synchronisation is rendered by a primitive communication process as described in Section 2.13.

2.3.2 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

Fig. 6 *Barrier synchronisation relationship*

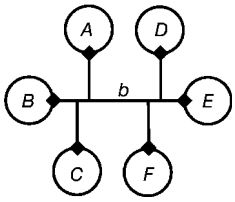


Fig. 7 *Barrier synchronisation*

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 system;
- 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 so that channels can communicate through intermediate devices. This implies support for hierarchies, reuse, and portability of processes.

2.4 Compositional relationships

A compositional relationship is a labelled relationship between two processes whereby the label is a binary operator that expresses their compositional behaviour. Thus, compositional relationships are a kind of relationship between processes that are useful for describing the execution order of communicating processes. A compositional relationship is a companion to a communication relationship. Fig. 8 shows two processes in relation to each other. This represents a composition of two processes whose semantics are described by the operator specified by the label.

Where *operator* $\in \{\rightarrow, \leftarrow, \|, \overleftarrow{\|}, \vec{\|}, \Box, \overleftarrow{\Box}, \vec{\Box}, \overleftarrow{\triangle}, \vec{\triangle}\}$. The operators with an arrow are directed operators whilst the remainder 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 parenthesising 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 2.12). 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

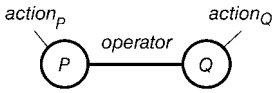


Fig. 8 *Compositional relationship between two processes*

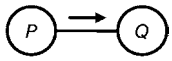


Fig. 9 *Sequential relationship*

label '$\rightarrow$'. This sequential composition is written as $P \rightarrow Q$. This has strong similarities with CSP's single action transition $P \xrightarrow{\surd} Q$. This process will behave as Q if P has successfully terminated 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 Fig. 9. 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 ';' are because the arrow gives more design freedom and the arrow denotes the direction of execution more clearly than ';'.

Multiple compositions are represented with more than two processes in the relationship, for example, $(P, Q, R, S, \rightarrow)$. See Fig. 10*a*. $(P, Q, R, S, \rightarrow)$ 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 Fig. 10*b*. Partial relationships can be derived from the main multiple relationships.

These partial relationships *cannot all* be represented with ';'. It is obvious that $(P, S, \rightarrow)$ does not represent $P;S$. Only process $(P, Q, R, S, \rightarrow)$ is represented as $P;Q;R;S$. Formally, what we mean by $(P, S, \rightarrow)$ is:

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

where X is a set of processes belonging to 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

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

In case of a multiple relationship we can write

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

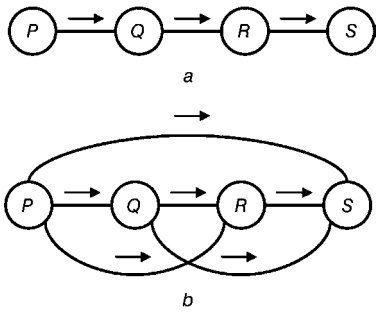


Fig. 10
a Multiple sequential relationship
b Multiple sequential relationship with derived relationships

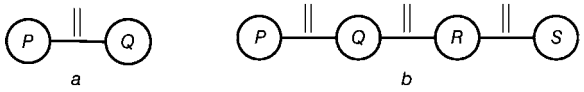


Fig. 11 *Parallel relationship*
a Parallel relationship between P and Q
b Parallel relationship between P, Q, R and S

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:

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

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 Fig. 11.

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 Fig. 12.

Therefore, we can write:

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

The processes $P_0\,..\,P_{n-1}$ 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

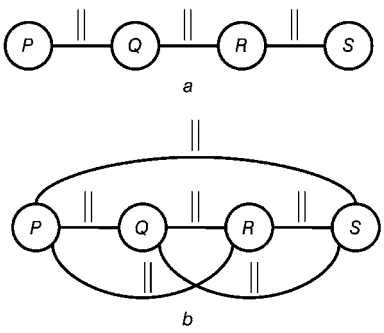


Fig. 12
a Multiple parallel relationship
b Multiple parallel relationship with derived relationships

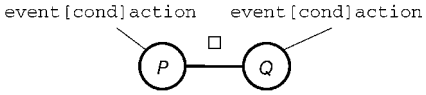


Fig. 13 *Alternative relationship*

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 Fig. 13.

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 being executed. Here, `event` can indicate a *channel-input*-guard, *channel-accept*-guard, *channel-output*-guard, *channel-call*-guard, *skip*-guard, or an *else*-guard. If `event` specifies time instead of a channel then we mean a *timeout*-guard. A *timeout*-guard becomes ready when the specified time expires starting from execution of the alternative relationship.

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 modelled as a *skip*-guard in a special prioritised alternative construction, see Fig. 14.

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 channels 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]. 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. This guideline prevents unwanted race hazards.

The composition (P, Q, R, S, $\Box$) represents multiple relationships. See Fig. 15.

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

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

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

2.4.4 Prioritised parallel relationship: A prioritised parallel relationship between processes P and Q is denoted by label '$\|$'. This parallel composition is written as $P \,\|\, Q$.

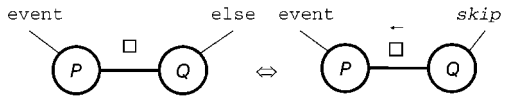


Fig. 14 *Else-guarded construct*

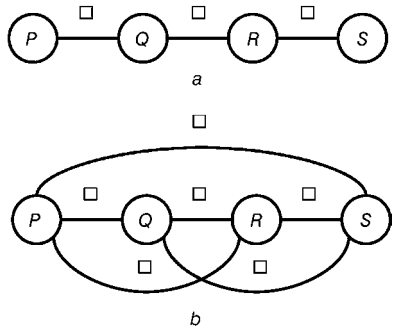


Fig. 15
a Multiple alternative relationship
b Multiple alternative relationship with derived relationships

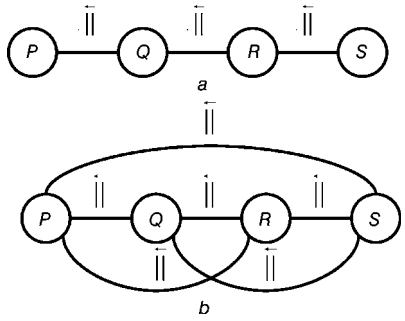


Fig. 17
a Multiple prioritised parallel relationship
b Multiple prioritised parallel relationship with derived relationships

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 prioritised parallel relationship between P and Q, see Fig. 16.

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 Fig. 17.

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

$$(P, \{\}, S, \overleftarrow{\|})\backslash\{\} = (P, S, \overleftarrow{\|})_{\text{ø}} = P \overleftarrow{\|} S$$

In the case of a multiple relationship we can write

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

$$= P \overleftarrow{\|} Q \overleftarrow{\|} R \overleftarrow{\|} S$$

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:

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

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

When `action` attributes are specified on parallel processes in a prioritised parallel relationship then these `action` attributes can be pre-empted. 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.

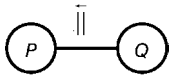


Fig. 16 *Prioritised parallel relationship*

2.4.5 *Prioritised alternative relationship:*

An alternative relationship between processes P and Q is denoted by the label '$\overleftarrow{\square}$'. This prioritised alternative composition is written as $P \overleftarrow{\square} Q$. 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 to Q. Process P is guarded with higher preference or priority.

We will use notation $(P, Q, \overleftarrow{\square})$ to represent a prioritised alternative relationship between P and Q. See Fig. 18.

A multiple relationship $(P, Q, R, S, \overleftarrow{\square})$ represents $P \overleftarrow{\square} Q \overleftarrow{\square} R \overleftarrow{\square} S$. See Fig. 19.

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

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

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

$$(P, \{\}, S, \overleftarrow{\square})\backslash\{\} = (P, S, \overleftarrow{\square})_{\text{ø}} = P \overleftarrow{\square} S$$

In case of a multiple relationship we can write:

$$(P, Q, R, S, \overleftarrow{\square})_{\text{ø}} = (P, Q, \overleftarrow{\square})_{\text{ø}}, (Q, R, \overleftarrow{\square})_{\text{ø}}, (R, S, \overleftarrow{\square})_{\text{ø}}$$

$$= P \overleftarrow{\square} Q \overleftarrow{\square} R \overleftarrow{\square} S$$

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:

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

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

2.4.6 *Exception (interrupt) relationship:*

An exception relationship between processes P and Q is denoted by the label '$\vec{\Delta}$'. This exception composition is written as $P \vec{\Delta} Q$. The process as rendered in Fig. 20 behaves as Q

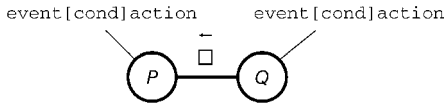


Fig. 18 *Prioritised alternative relationship*

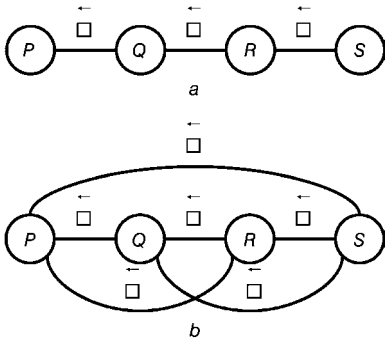


Fig. 19

a Multiple prioritised alternative relationship
b Multiple prioritised alternative relationship with derived relationships

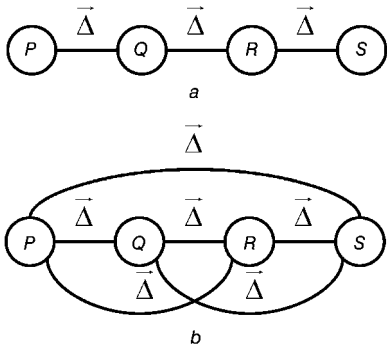


Fig. 21

a Cascade of exception handling
b Multiple exception relationships with derived relationships

when P unsuccessfully terminates; otherwise this process behaves as P. If P successfully terminates then Q will be omitted.

This exception relationship is not formally defined in CSP. The exception operator originates from the interrupt operator $P\Delta_i Q$ in CSP. 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. The exception operator is a simplified version of the interrupt operator whereby event i is represented as an internal event that is observable as a termination event of P. In this case, process P terminates unsuccessfully as a result of an error somewhere in the process. The unsuccessful termination causes the exception operator to execute the exception handling process Q.

If an exception is thrown in P then this indicates an exceptional state and P returns immediately with a message indicating the type of the exception. The exceptional state or throw action represents an invisible internal event in 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. This is implicit in the model. In this version, the actual exception message or exception handling is coded in a programming language and not in the CSP diagram. Computations or primitive communication processes (Section 2.13) render points in the model where exceptions can rise. Thus, channels and barriers can be sources of errors and they are allowed to throw exceptions at both sides of the communication. This way an erroneous process cannot lock a channel or barrier. This can be implemented with try-and-catch clauses as found in Java and C++.

In design, exception handling can be composed with multiple exception relationships (see Fig. 21*a*) and with redundant exception relationships (see Fig. 21*b*).

2.5 Parenthesised compositional relationships

As with algebraic compositions, the use of parentheses is inevitable to write more complex expressions. Parentheses are also required in this modelling language. Consider a model with three processes P, Q and R as shown in Fig. 22.

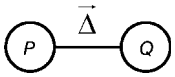


Fig. 20 *Exception relationship*

In this example, we specify that process P should be executed before Q and Q should be executed in parallel with R. The behaviour 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 $(P;Q)\|R$.

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 Fig. 23. Each cycle eliminates ambiguous interpretations.

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 '○' (concatenation of (+)) at the peer-end of the compositional relationship. For example, Fig. 23*a* and Fig. 23*b* are the equivalence of, respectively, Fig. 24*a* and Fig. 24*b*. Using parenthesises symbols reduces the number of 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 *parenthesising relationship*. A *parenthesised relationship* is a stronger relationship than its neighbour parenthesising relationships which are directed to a process in the parenthesising relationship. A chain of parenthesising 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 parenthesising relationships

A dot in the parenthesising relationships can be indexed with an integer greater than zero. See Fig. 25, where $i \in [1, \rightarrow)$. The index is an instrument useful for reallocating relationships in order to maintain the algebraic expression. This is discussed in Section 2.9.

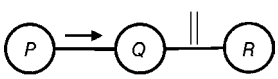


Fig. 22 *Example of a model with ambiguity*

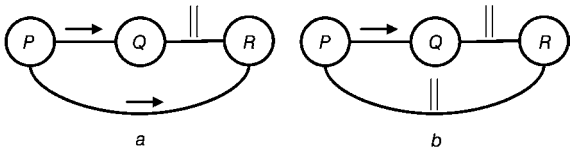


Fig. 23 *Unambiguous solutions using cycles of relationships (complete graph)*
a $P;(Q \parallel R)$
b $(P;Q) \parallel R$

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 and virtually, 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 neighbours. 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 do not care what the execution order is and therefore we let the tool or environment decide. A hidden connection is an indexed parenthesising compositional relationship, with parenthesis symbols at both ends of the line, as shown in Fig. 26.

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. 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 visualised for understanding the generated framework at the same level of abstraction. For example by using grey lines. 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 to visualise the hidden relationships is an ultimate solution for debugging and studying the behaviour of the model at the design level.

Showing every connection makes the graph easily unreadable. Fortunately, not every connection has to be shown. Redundant relationships could be removed from the graph in order to make the graph more readable as illustrated in the second solution at the right-hand side of Fig. 26. Also a tree-graph which contains a minimised

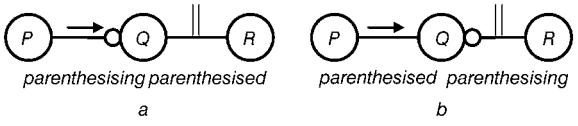


Fig. 24 *Unambiguous solutions using parenthesised relationships*
a $P;(Q \parallel R)$
b $(P;Q) \parallel R$

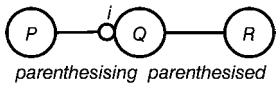


Fig. 25 *Indexed parenthesising relationship with index i*

number of visible relationships without leaving information out can be easily constructed.

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 parenthesising relationship, i.e. being strong relationships. These undefined relationships become parenthesised 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 near to 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 the case of a process that 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 makes the model better readable.

Fig. 27*a* 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 Fig. 27*d*. The steps are illustrated in Fig. 27*b*–27*d*.

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. Fig. 28*a*–27*f* shows six basic rules for reallocating relationships.

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 $\bar{\oplus}$ to show the commutative properties.

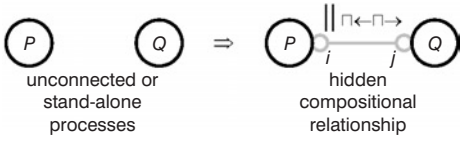


Fig. 26 *Stand-alone processes and hidden compositional relationship*

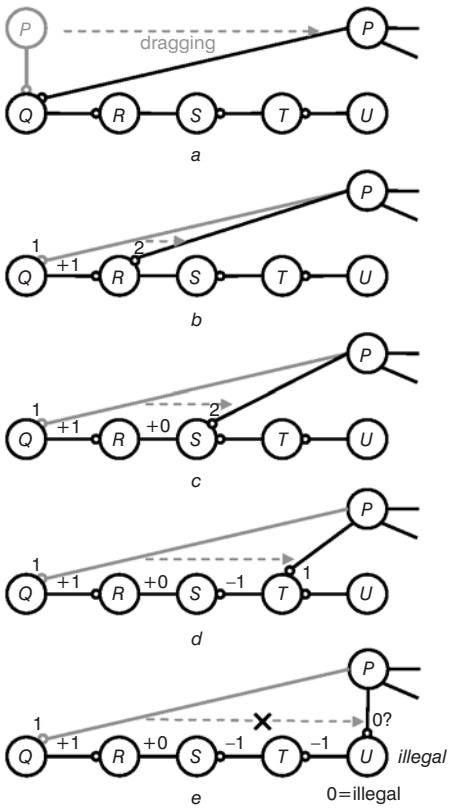


Fig. 27 *Example of reallocating a connection*

a Original configuration
b Reallocation step 1
c Reallocation step 2
d Reallocation step 3, final configuration
e Illegal reallocation, boundary crossed

Let operator $\oplus$ represent a binary CSP operator, where by $\oplus \in \{\leftarrow, \|, \underleftarrow{\|}, \square, \underleftarrow{\square}\}$. Operator $\bar{\oplus}$ is the complement of $\oplus$, where by $\bar{\oplus} \in \{\rightarrow, \|, \underrightarrow{\|}, \square, \underrightarrow{\square}\}$. These operators are directional commutative

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

The processes P and Q are relationship equivalent: $(P, Q, \oplus) = (Q, P, \bar{\oplus})$. For example, $P \underleftarrow{\|} Q = Q \underrightarrow{\|} P$, $P \underleftarrow{\square} Q = Q \underrightarrow{\square} P$, $P \rightarrow Q = Q \leftarrow P$, $P \underrightarrow{\|} Q = Q \underleftarrow{\|} P$, $P \underrightarrow{\square} Q = Q \underleftarrow{\square} P$.

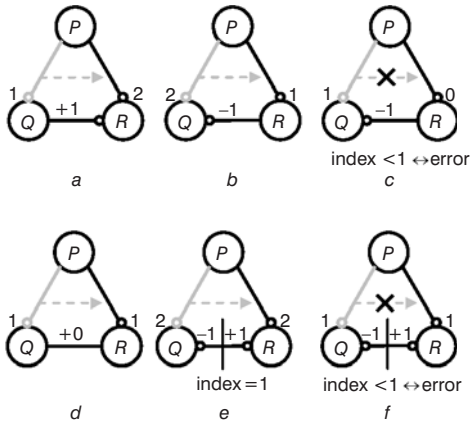


Fig. 28 *Reallocation rules*

a Incremental indexing
b Decremental indexing
c Decremental illegal indexing
d Equal indexing
e Equal indexing
f Equal illegal indexing

In Fig. 28 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 Fig. 28*c* and Fig. 28*f* 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 Fig. 27, 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 Fig. 27*e*. This reallocation is prohibited according to the rule as depicted in Fig. 28*c*. The index may not go below 1.

The algebraic expressions of Fig. 27*a* and Fig. 27*d* are equal:

$$P \oplus_1 (Q \oplus_2 (R \oplus_3 S) \oplus_4 T) \oplus_5 U = P \oplus_1 (T \bar{\oplus}_2 (S \bar{\oplus}_3 R) \bar{\oplus}_4 Q) \oplus_5 U$$

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

$$(P \oplus_1 U) \oplus_2 (T \bar{\oplus}_3 (S \bar{\oplus}_4 R) \bar{\oplus}_5 Q) \neq P \oplus_1 (Q \oplus_2 (R \oplus_3 S) \oplus T) \oplus_5 U$$

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 parenthesised cycles

Cycles of parenthesising relationships in a design should be *balanced*. This means that in a cycle the number of parenthesising relationships pointing in one direction should compensate the number of parenthesising relationships pointing in the other direction. The counting starts and ends in one process of the cycle. If these parenthesising relationships do not compensate opposite parenthesising relationships in the cycle then one cannot completely determine the algebraic expression of this so-called *unbalanced cycle*. In an unbalanced parenthesised 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.

2.11 Compositional conflict checking technique

The CSP language provides several features to determine compositional conflicts in design and requirements. We have developed a systematic approach that will combine these features to find compositional conflicts [11]. 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 temporarily reallocating redundant relationships 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 parenthesising 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.

2.12 Recursion based on anonymous hierarchies

As with many programming languages this graphical language supports recursion. A process can represent 'nameless' recursion involving X, as in

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

Similarly, recursion is depicted by a special μ-process for constructing looping/recursive constructs as shown in Fig. 29.

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 [11]. 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 looping.

A few different kinds of loops are shown in Fig. 30 that can also be found in many programming languages. These loops can be modelled with the μ-process and `action` attachments.

The μ-process can also be used with other compositional relationships other than sequential.

2.13 Primitive communication processes

All CSP relationships in the communication diagram and in the composition diagram are all synchronisation points in the design model. This section introduces three communication primitives that express synchronisation points in the design model. We model these synchronisation points as primitive communication processes. These primitive processes are depicted in Fig. 31.

The termination events of these processes correspond to their shared communication events. Fig. 31*a* illustrates a channel-output or a channel-call. Fig. 31*b* illustrates a channel-input or a channel-accept. Fig. 31*c* illustrates a barrier synchronisation entry on a shared barrier. These primitive processes are useful for

- showing the points of interaction between processes;
- showing hardware access points [12];
- checking for deadlocks in design;
- checking for priority inversion problems in design;
- throwing exceptions on lower-level errors.

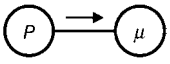
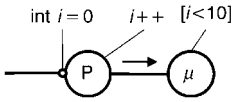


Fig. 29 *Infinite recursion*

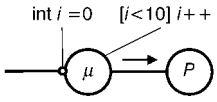
DO-WHILE construct:

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



FOR construct:

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



Recursion: P[0] || P[1] || P[2]

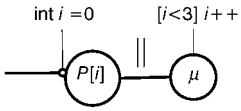


Fig. 30 *Looping and recursion*

2.14 Deadlock analysis using compositional relationships

Compositional conflicts in a CSP diagram found between the primitive communication processes reflects deadlock. 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 synchronisation 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 Fig. 31 play an important role in analysing the model for potential deadlocks. A *deadlock* is a failure of two processes to cooperate with each other because of

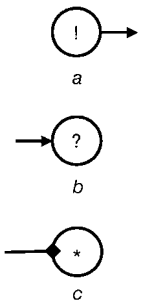


Fig. 31 *Primitive processes*

a Channel output/call process
b Channel input/accept process
c Barrier sync process

not being able to agree on a common event, although they are willing to participate in other events.

A good solution to finding deadlocks and other phenomena is the use of formal deadlock checkers. For example, the model could be translated into readable CSP and analysed by a tool such as FDR [13]. 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. The language provides enough information. 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 techniques involving these primitive communication processes.

For example, Fig. 32*a* shows a model that is conflict-free. The model can be code generated and executed. At run-time, these processes synchronise on channel communication or on barrier synchronisation 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 synchronise 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), see Fig. 32*b*. The relationships between the primitive processes must be parallel or prioritised parallel, see Fig. 33.

It is necessary that the model is flattened, i.e., all processes are brought to the same level in the hierarchy. First we merge the pair of primitive communication processes together into *rendezvous processes* and watch the compositional relationships between the rendezvous processes. 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, Fig. 33*a* is conflict-free but not deadlock-free. One can see after merging that the sequential relationships are in contradiction. This sequence conflict is a potential deadlock. This is the same for barrier synchronisation.

2.15 Priority inversion analysis using compositional relationships

With the same technique as in the previous section one can find priority conflicts whereby prioritised parallel operators are in contradiction, see Fig. 34*b*. In this case, the model suffers from *priority inversion*. The priority inversion problem can place a significant burden on the performance

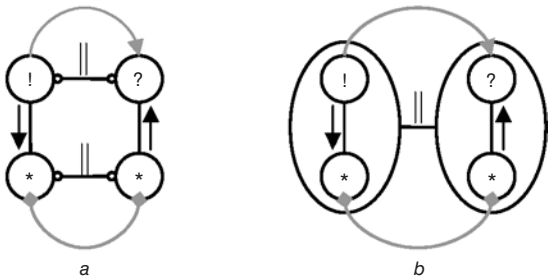


Fig. 32

a Original design
b Compositional conflict-free

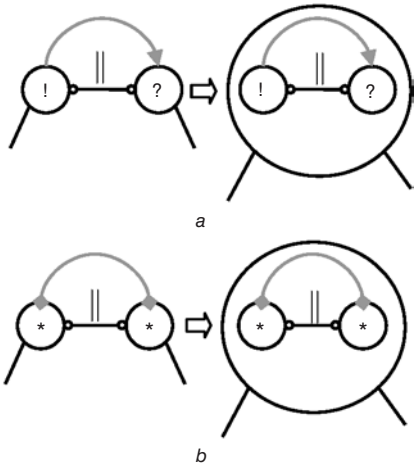


Fig. 33

a Channel comm. to rendezvous process
b Barrier comm. to rendezvous process

of the software. It this case, a higher priority process can be blocked by the lower priority process and as a result of that it may be that the deadlines of the higher priority process cannot be met.

Usually, eliminating this design conflict by correcting prioritised compositional relationships will result in a better design. In the case where priority inversion is inevitable in the design, which is possible, one could solve the problem by use a special channel (i.e. a tuned communication relationship) between processes executing at different priorities in order to delay blocking. A simple buffered channel can help to solve priority inversion problems. The communication relationships in the communication-graph between processes that are in priority conflict specify the location of a buffered property. A tool could use this information to change the 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. The notion of time

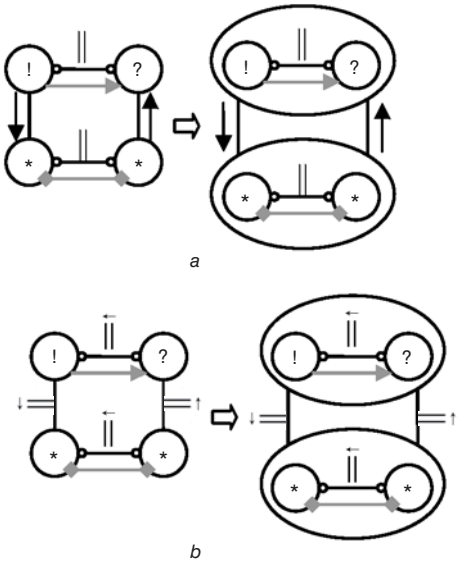


Fig. 34

a Sequence conflict = deadlock
b Priority conflict = priority inversion problem

is not discussed in this paper and is the subject of further research.

3 Related work

The graphical language as presented here is immediately useful for drawing models by hand or with some drawing tool. These CSP diagrams can be used for documenting concurrency of a real-time software architecture. However, in order to fully benefit from the language and the techniques behind it, then tool support is inevitable. Therefore, we are developing a prototype tool that allows us to design sophisticated control software at our control laboratory.

The tool will be developed in Java and it will become available for everyone who is interested in this development. Information about the tool can be found at our web site [14]. The tool will support two add-in facilities to allow extending the tool with self-made model-checking or code-generation modules. We hope to encourage researchers to be able to use and to extend this technology.

4 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 modelling language acts as a glue-logic between structured methods and object-orientation—providing continuation between the two paradigms. A CSP diagram is UMLable and can be used as a 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 [14] and JCSP [15] can be used to implement CSP diagrams.

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

This graphical modelling 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 analysed, checked, simulated, and finally executed on a dedicated embedded real-time system. Design tools are required to support this graphical modelling language so that a software architect can really benefit from CSP diagrams. In this paper we introduced the basics of the graphical modelling language and further research is required to enhance the graphical modelling language and to build tools for designing CSP diagrams and for code-generation.

5 References

1 HOARE, C.A.R.: 'Communicating sequential processes' (Prentice-Hall, London, UK, 1985)
2 ROSCOE, A.W.: 'The theory and practice of concurrency' (Prentice-Hall, Hertfordshire, 1998)
3 WARD, P.T., and MELLOR, S.J.: 'Structured development techniques for real-time systems' (Prentice-Hall, Englewood Cliffs, NJ, 1985)
4 HATLEY, D.J., and PRIBHAI, I.A.: 'Strategies for real-time system specification' (Dorset House Publishing, New York, NY, 1987)
5 YOURDON, E.N.: 'Modern structured analysis' (Prentice Hall, Englewood Cliffs, NJ, 1989)
6 DOUGLASS, P.B.: 'Real-time UML: developing efficient objects for embedded systems' (Addison Wesley Longman, Inc., Reading, Massachusetts, 1998)
7 BOOCH, G., RUMBAUGH, J., *et al.*: 'The unified modeling language—user guide' (Addison-Wesley, Reading, Massachusetts, USA, 1999)
8 DOUGLASS, B.P.: 'Doing hard timer: developing real-time systems with UML, objects, frameworks, and patterns' (Addison Wesley Longman, Inc., Reading, 1999)
9 ROSCOE, A.W.: 'Routing messages through networks: an exercise in deadlock avoidance'. 7th Occam User Group & International Workshop on Parallel programming of transputer based machines, 14–16 September, 1987, Grenoble, LGI-IMAG
10 JONES, G.: 'On Guards'. 7th Occam User Group & International Workshop on Parallel programming of transputer based machines, 14–16 September 1987, Grenoble, LGI-IMAG
11 PASCOE, J.S., WELCH, P.H., *et al.*: 'Communicating process architectures 2002'. Proceedings of Communicating Process Architectures, IOS Press, Reading, 15–18 September 2002
12 HILDERINK, G.H., BROENINK, J.F., *et al.*: 'Software design method for heterogeneous embedded systems'. Presented at 17th Benelux Meeting, 4–6 March 1998, Mierlo, NL
13 FDR, Formal Systems Ltd.: 'FDR2', http://www.formal.demon.co.uk /
14 HILDERINK, G.H.: 'Communicating Threads for Java (CTJ) home page', http://www.ce.utwente.nl/javapp, accessed 5 September 2002
15 WELCH, P.H., and AUSTIN, P.D.: 'The JCSP home page', http://www.cs.ukc.ac.uk/projects/ofa/jcsp, accessed 5 September 2002