Chapter 5

Concurrent Communicating Sequential Processes

The extended Parallel Data Processing Model

epubli, ISBN 9783746752228 (2018)

This chapter introduces a parallel multi-process model with global shared resources and the *ConPro* programming language used for the design of parallel data processing systems, e.g., the agent processing platforms presented in this book, specifically suitable for application-specific System-on-Chip (SoC) architectures and designs.

5.1 Parallel Data Processing

Embedded systems used for the control, for example, in Cyber-Physical-Systems (CPS), perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. SoC designs are preferred to achieve high miniaturization and low-power applications. Traditionally, program-controlled multiprocessor architectures are used for the execution platform with very limited parallelization capabilities, but application-specific digital logic gains more importance, which can enable parallel data processing.

Concurrency has great impact on the system and data processing behaviour concerning latency, data throughput, and power consumption. Streaming and functional data processing requires fine-grained concurrency on the data path level, however, reactive control systems (for example serving communication tasks) require coarse-grained concurrency on control path level.

The behavioural level usually describes the functional behaviour of the full design interacting with the environment. Most applications and data processing are modelled on algorithmic behavioural level using some kind of imperative programming language.

The following introduction of a process-based data processing model is required and used

1. For the implementation of the agent-behaviour (activities and transition);
2. For the implementation of the agent processing platform.

5.2 The original CSP Model

The Communicating Sequential Processes model (CSP, introduced by C. Hoare [HOA85]) is a common computational model for parallel and distributed computing used extensively in the last decades. The CSP model is basically state-based, composing the state of a computational system in processes with a control and data state. Each process performs the processing with a sequence of instructions modifying the state of the process.

The CSP model is very attractive in parallel software development (multi-threaded programming models) and for hardware design due to the proximity to finite state machines and Register-Transfer (RTL) architectures. The

epubli, ISBN 9783746752228 (2018)

state-based ATG/*AAPL* agent behaviour model can be easily mapped on the CSP model.

One major aspect of the CSP model is the capability to compose complex parallel and distributed systems with a set of processes by using process constructors providing the design of arbitrary nested sequential and parallel processes.

Furthermore, CSP is a language based on process algebra for describing and modelling interaction between processes, e.g., communication. Therefore, it can be used to analyse the system behaviour of parallel and distributed systems.

There are basically three different process classes distinguished by their composition operation and execution behaviour [PARSYS]:

Process

A process p is an active execution unit associated with a data and control state. The behaviour of the control state of a process can be modelled with State-Transition Graphs (STG). The data state consists of the current values of all storage variables assigned to a process. The control state is associated by a particular action performed by the process, i.e., the execution of an instruction. Each process has an abstract process execution state PS with a state from the set $PS \in \{\texttt{START}, \texttt{RUN}, \texttt{BLOCKED}, \texttt{STOP}\}$, which can be affected by other processes. Possible processing state transitions are given by Definition *5.1*, which depends on the external process environment and the behaviour of other processes.

Def. 5.1 *Allowed process execution state transitions*

$$
\begin{array}{ll}
\texttt{START} & \rightarrow \texttt{RUN} \\
\texttt{RUN} & \rightarrow \texttt{BLOCKED} \\
\texttt{BLOCKED} & \rightarrow \texttt{RUN} \\
\texttt{BLOCKED} & \rightarrow \texttt{STOP} \\
\texttt{RUN} & \rightarrow \texttt{STOP}
\end{array}
$$

This means a process must be activated by an event $\Diamond x$, performing the execution state transition $\texttt{START} \rightarrow \texttt{RUN}$, for example, based on a communication with another process. Process blocking is a result of an event participation, and a process is blocked (halted) until an event occurs. The termination of a process results in the execution state transition $(\texttt{RUN}|\texttt{BLOCKED}) \rightarrow \texttt{STOP}$, and can have different reasons: Normal termination, exceptional termination (a fault occurs), or external termination controlled by another process.

Multiprocess Sets

A set of processes P consists of multiple distinct processes p_1, p_2, ..., which is written $P=\{p_1, p_2, ..\}$. The processes interact with each other following specific rules and orders, discussed below. The set can be considered as being a meta process encapsulating the contained processes.

Events

An event is commonly related to input and output operations and the happening of an event requires the satisfaction of a condition *cond* (e.g., the availability of data), which is indicated with an identifier prefixed with $\Diamond x(cond)$, or in short form $\Diamond x$. There is an empty event $\Diamond$(true) or in short form $\blacklozenge$, which can happen at any time.

Process Blocking

It was assumed that a process is activated (started), performs data processing, and terminates. Without any external communication and inter-process synchronization a process will always pass through the process state sequence START→RUN→STOP. But inter-process communication will introduce wait conditions for a process that will block (suspend) the process execution until an event occurs and the condition is satisfied (e.g., change of data).

In this case, a process will pass through an extended process state sequence in the form of START→RUN→BLOCK→RUN→BLOCK→..→STOP.

Process blocking is a fundamental concept enabling the synchronization of multiple process systems and requires engaging of process in events, written formal:

$p_1 \rightarrow \Diamond x_1(cond_1) \rightarrow p_2 \rightarrow \Diamond x_2(cond_2)$ ..

Elementary Processes

An elementary process is an atomic process executing one atomic instruction without interruption. An elementary process must be started (activated) and terminates after the execution of the instruction. An elementary process is equal to a process set $p=\{i\}$.

The instruction of an elementary process can modify the data or the control state of a program.

Sequential Processes

A sequential process is composed of a sequence of elementary processes (instructions) $I=\{i_1, i_2,..\}$ of elementary statements $i \in ST=\{st_1, st_2,..\}$ executed in the given order. There are pure computational statements (changing the data state of a process, e.g., assignments) and control statements

changing the control state (the currently active process instruction) of the process (i.e., branches and loops), which is illustrated in Figure 5.1, and formally given in Definition 5.2.

Therefore, the activation of an elementary process (i.e., an instruction) can be conditional and depends on the evaluation of computed or external data. Process alternation and process flow choices are discussed later.

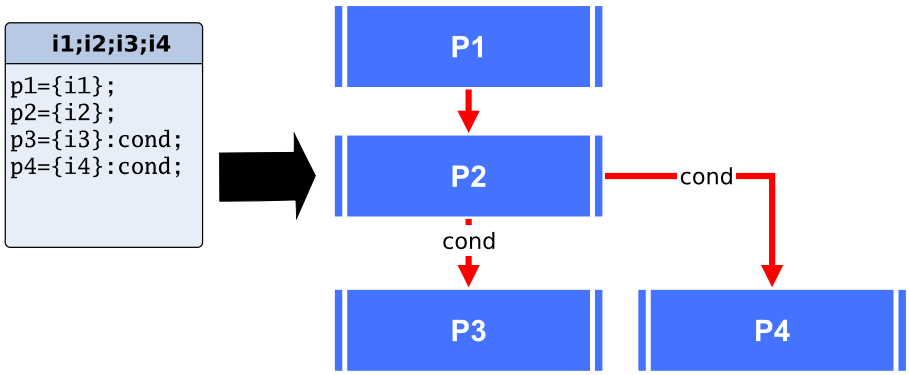

Fig. 5.1 *Process flow: Instructions can be considered as being elementary and atomic processes; a sequential composition activates a process chain one after one.*

Def. 5.2 *Sequential Process Composition Constructor p_{seq} (; operator) and the execution timing model enclosed in ⊢⊣ and denoted with $x\angle\tau$ giving the execution time point or interval of a sub-process. [➡: process transition, ➠: temporal flow]*

Composition

$p_{seq} : \{p_1, p_2, ..\} \quad \rightarrow \quad p_1 ; p_2 ; p_3 ; p_4 ; ..$

$\Leftrightarrow$

$p_{seq} : \{i_1, i_2 , ..\} \quad \rightarrow \quad p_1=\{i_1\}; p_2=\{i_2\} ; p_2=\{i_3\}; p_4=\{i_4\} ..$

Process Flow Behaviour

$p_{seq} : \{p_1, p_2, ..\} \quad \equiv \quad \blacktriangleright p_1 \blacktriangleright \blacklozenge \blacktriangleright p_2 \blacktriangleright \blacklozenge \blacktriangleright ..$

Timing Model

$\vdash p_1=\{i_1\} \angle\tau_1 \Rightarrow p_2=\{i_2\} \angle\tau_2 \Rightarrow p_3=\{i_3\} ... \dashv$

with $\tau_1 < \tau_2 < \tau_3 < ..$

The execution of a sequential process $p_{seq}=\{p_1, p_2, .., p_n\}$ starts the first process p_1 of the sequence. After the termination of this sub-process, the next process p_2 is activated by an empty event. The sequential process p_{seq} terminates if the last sub-process p_n has terminated.

Control State

Each elementary statement $i \in ST$ of a sequential process p_{seq} is related to a state s_i of a set of control states $s \in S=\{s_1, s_2,..\}$. The values of all data variables of a process create the data state D of a process, a multidimensional vector $D=[v_1, v_2,..]$.

There is a set of transitions T with $t_{1,2} : s_1, \rightarrow s_2 \mid cond$ between states of a sequential process activating instruction processes, which can depend on the satisfaction of preconditions *cond* related to process data. All states and transitions create a state-transitions graph $STG = \langle S , T \rangle$, where the transitions are the edges of the graph connecting transitions. An example is shown in Figure 5.2.

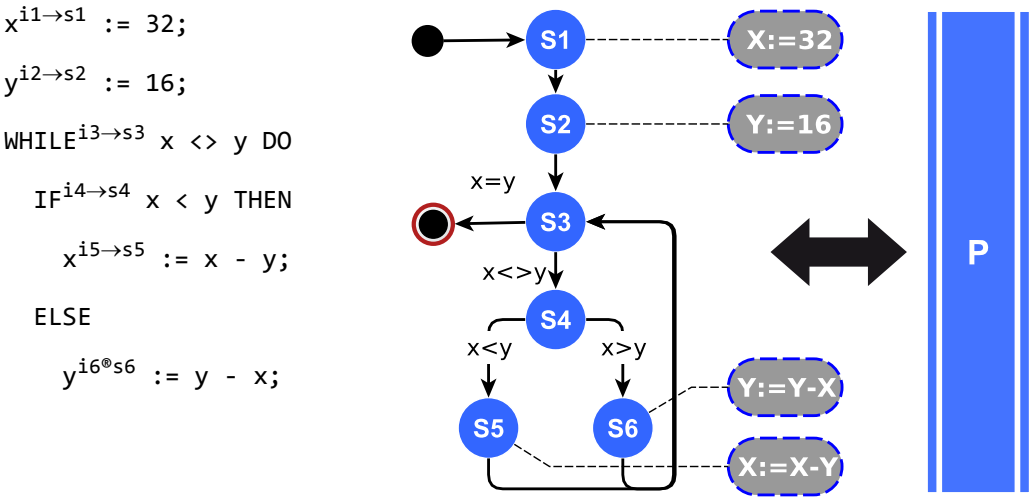

Fig. 5.2 *An imperative program fragment (GCD computation algorithm) and the relation to a state-transition graph and the sequential process. Each elementary instruction is assigned to a control state. Pure decision states (i.e., S4) can be further compacted with the previous control state (i.e., S3).*

Interruption and Restart

A sequential process p can be interrupted by another process q, written as $p \hat{} q$, which does not depend on a regular termination of the process p. Furthermore, a sequential process can be restarted, either performed explicitly by another process, introducing the concept of looping or iteration, written as $\hookleftarrow p$, or implicitly due to a catastrophic failure. Restarting of processes enables the implementation of procedures that can be executed repeatedly.

Parallel Processes

A parallel process consists of a composition of processes executed concurrently that can be elementary, sequential, and other parallel processes. Initially, these processes execute without any coordination except the start and termination of each individual process, shown in Figure 5.3, and formally defined in Definition 5.3.

It is expected that multiple processes execute concurrently that they interact with each other. The interaction introduces events that can be considered as being checkpoints. These checkpoints require some simultaneous participation of the processes. There is an alphabet α of events for each process. Thus, the union of event sets of processes implement the co-ordination and synchronization of a set of processes, for example:

$$\alpha(p \parallel q) = \alpha p \cup \alpha q$$

Def. 5.3 *Parallel Process Composition (|| operator) and the corresponding timing model*

Composition

$$p_{par} : \{p_1, p_2, ..\} \rightarrow p_1 \parallel p_2 \parallel p_3 \parallel p_4 \parallel ..$$

Process Flow Behaviour

$$p_{par} : \{p_1, p_2, ..\} \equiv \blacktriangleright (p_1 \parallel p_2 \parallel p_3 \parallel p_4 \parallel ..) \blacktriangleright ..$$

Timing Model

$$\vdash p_1 \dashv = (i_{1,1} \angle \tau_{1,1} \Longrightarrow i_{1,2} \angle \tau_{1,2} \Longrightarrow i_{1,3} \angle \tau_{1,3} \Longrightarrow i_{1,4} \angle \tau_{1,4}, ..) \angle [\tau_{1,1}, \tau_{1,n}]$$

$$\vdash p_2 \dashv = (i_{2,1} \angle \tau_{2,1} \Longrightarrow i_{2,2} \angle \tau_{2,2} \Longrightarrow i_{2,3} \angle \tau_{2,3} \Longrightarrow i_{2,4} \angle \tau_{2,4}, ..) \angle [\tau_{2,1}, \tau_{2,n}]$$

$$\vdash p_3 \dashv = (i_{3,1} \angle \tau_{3,1} \Longrightarrow i_{3,2} \angle \tau_{3,2} \Longrightarrow i_{3,3} \angle \tau_{3,3} \Longrightarrow i_{3,4} \angle \tau_{3,4}, ..) \angle [\tau_{3,1}, \tau_{3,n}]$$

with $\vdash p_1 \dashv \angle [\tau_1, \tau_n] =$

$[\tau_{1,1}, \tau_{1,n}] \cup$

$[\tau_{2,1}, \tau_{2,n}] \cup$

$[\tau_{3,1}, \tau_{3,n}] ...$

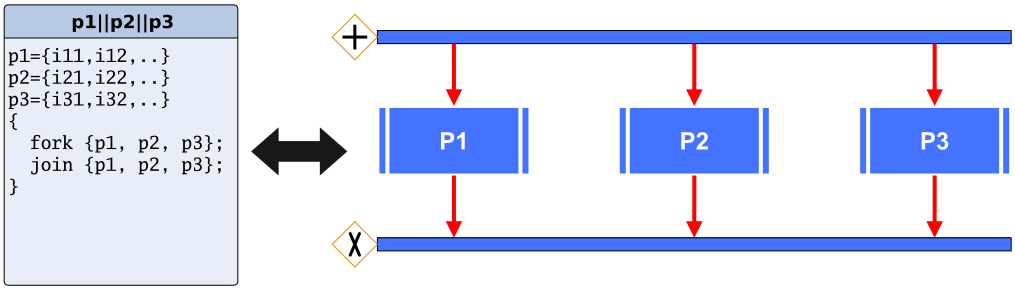

Fig. 5.3 *Control flow of the parallel process execution: the process flow is forked and after all processes are terminated joined again (all processes are encapsulated in a meta process).*

Dynamic Process Creation and Process Forking

Processes can be contained in a meta process (explained below) or can be created by a process performing forking of a child process, either a copy of the parent or a new (totally different) process. After a child process was forked, the parent and the child processes are executed in parallel. The termination of the new compound process may depend on the termination of both processes, but the parent process is not waiting for the termination of the child process. The fork operation can create a new process dynamically at run-time, or statically only starting the child process that was already instantiated (important for resource-limited designs), and it acts as a dynamic process flow forking with a final joining.

Def. 5.4 *Parallel Process Composition using Forking (// operator). The parent or main (master) process p_A creates a subordinated process p_B.*

Composition

$$p_{fork} : \{p_A, p_B\} \quad \rightarrow \quad p_B \,//\, p_A \neq p_A \parallel p_B$$

Process Flow Behaviour

$$p_{fork} : \{p_A, p_B\} \quad \rightarrow \quad \blacktriangleright p_A \blacktriangleright \Diamond fork \blacktriangleright (p'_A \parallel p_B) \blacktriangleright$$

Process Alternation

A process alternation selector allows the observation of multiple events, e.g., blocking IO statements, considered as being elementary processes. If one of the statements, for example, containing a blocking access of a channel, is ready, the statement is executed and an associated process is started handling this event, shown in Figure 5.4, and defined formally in Definition 5.5.

Def. 5.5 *Process Choice Alternation (Handler p_{Hi} waits for the execution of a blocked IO process p_{IOi}):*

Composition

$p_{choice,\,general} : \{p_1, p_2, p_3, ..\} \rightarrow p_1 \mid p_2 \mid p_3 .. \Rightarrow$

$p_{choice,event} : \{p_{IO1}, p_{IO2}, .. , p_{H1}, p_{H2}, .\} \rightarrow p_{H1} : p_{IO1} \mid p_{H2} : p_{IO2} ..$

Process Flow Behaviour

$p_{choice} : \{p_{IO1}, p_{IO2}, .. , p_{H1}, p_{H2}, ..\} \rightarrow$

$(\Diamond io_1 \blacktriangleright p_{IO1}) \blacktriangleright p_{H1} \mid (\Diamond io_2 \blacktriangleright p_{IO2}) \blacktriangleright p_{H2} \mid ..$

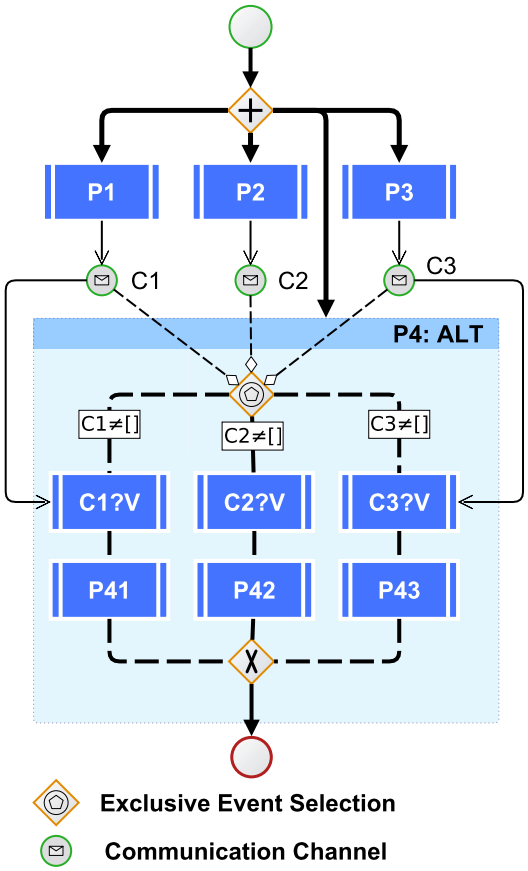

Fig. 5.4 *Process alternation flow. A process observes multiple blocking IO processes, e.g., the reading of channels, by using an alternation, finally executing a handler.*

The process alternation constructor ensures the mutual exclusion in the case more than one IO event occurs (i.e., ready IO statements). Only one IO statement process is activated at any time, queueing other pending IO processes that are ready. The process alternation acts as a conditional fork in the process flow.

Process Flow and Meta Processes

Processes can be encapsulated and grouped in a meta process. In the above definitions p_{par}, p_{seq}, p_{fork} are meta processes, for example. The starting of the meta process starts the process control flow inside the meta process. The meta process terminates iff all processes have terminated (see Figure 5.5), and implicates fork and join operations of the process control flow (i.e., the flow of process activation).

The process flow is the control flow on process level, which is different from the control flow on instruction level, though it can depend on it. The process flow has its origin in the process control state and the process control operation set, which modifies the process state.

In principle, there are three four basic situations in the process flow:

1. A terminated process activates implicitly one next process in a sequence;
2. The process flow forks and starts multiple processes of a group;
3. The process flow joins the termination of the group of processes;
4. A process starts (fork) or stop explicitly a process.

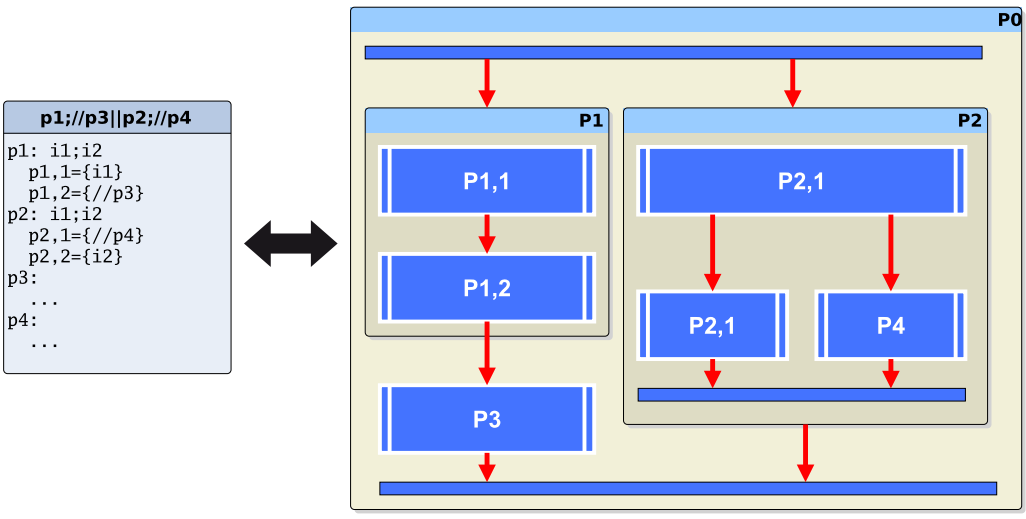

Fig. 5.5 *Examples of meta processes and process forking at run-time. The process flow contains fork and join planes (multi-threading).*

Process Interleaving

Process interleaving, written as $p_1 ||| p_2 ||| p_3 ..$, is a fundamental method to enable resource sharing and resolving of the resource competition arising with the concurrent access of shared global resources by multiple processes. Interleaved processes execute concurrently as long as they do not access shared resources, otherwise they are executing as sequential processes.

5.3 Inter-Process Communication and Synchronization

Parallel execution of processes require synchronization regarding their control flow and the possibility to exchange data. Process synchronization occurs if the progress of computation of one process depends on the state of other processes. The simplest synchronized inter-process communication is the activation (starting) and joining (waiting for termination) of processes. Synchronization requires the concept of *process blocking*, that means the interruption of the control flow of a process until an event occurred.

There are two kinds of process interaction requiring synchronization:

1. Competition $\Rightarrow$ Concurrent access of shared resources;
2. Cooperation $\Rightarrow$ Distribution, exchange, and merging of data.

Channel-based Communication. The original CSP model uses channels with a handshake behaviour for the data exchange and the synchronization (co-ordination) of processes. A queue is the extension of a channel with a buffer consisting of cells, that is some kind of shared data memory resource. Since the original CSP do not allow concurrent access of shared resources, or more precisely there may no be competition, there may be only one writer and one reader process accessing a channel.

The principle usage of queues by processes is shown in Figure 5.6. Without concurrent access of a queue, there is one reader and one writer process.

A channel or a queue can be considered as an abstract data type object that is accessed by applying a read or write operation, explained in Definition 5.6, which is parametrizable by the data type (α) of the queue elements and the maximal number of stored data elements (*size*). A special case is a queue with *size*=1, providing a fully handshake data transfer between two processes (four-phase acknowledge).

Channel: The invariant to be always satisfied is composed of the conditions: 1. If there is no pending write operation, a read operation is blocked, 2. If there is no pending read operation, a write operation is blocked. A channel is fully synchronous.

epubli, ISBN 9783746752228 (2018)

Queue: The invariant to be always satisfied is composed of the conditions: 1. If the queue is empty, a read operation is blocked, 2. If the queue is full, a write operation is blocked. A queue is semi-synchronous.

The algorithmic description of the queue behaviour is given in Algorithm 5.1.

Def. 5.6 *Channel-based communication. A channel offers a read and a write operation for the coordinated exchange of data between processes.A read or write statements is treated as a sequential process.*

Short Notation

V:=C.read() $\equiv C?V$
C.write(ε) $\equiv C!\varepsilon$

V:=Q.read() $\equiv Q?V$
Q.write(ε) $\equiv Q!\varepsilon$

with a channel C (or queue Q with N buffer cells) and a variable V.

Process Flow Behaviour

$(C?V; p) \parallel (C!\varepsilon; q) \Rightarrow p \parallel q$

Alg. 5.1 *Queue object operations and synchronization (with data type α) in pseudo-notation. It is assumed that two processes p_1 and p_2 uses the queue (one reading, one writing)*

```
object specification QUEUE =
  PARAMETER = {α=<datatype>,size:integer}
  STATES = {EMPTY,FULL,FILLED}

  operation write(x): (process,α) → unit
  operation read: process → α

object behaviour QUEUE =
  VAR = state:STATES, QD: array[size] of α

  operation write(p,x) is
    If state=FULL Then Block(p) until state≠FULL
    eval x,QD=[y,z,..] → QD=[x,y,z,..]
    If |QD| = size Then state := FULL Else state := FILLED


  operation read(p) return(x) is
    If state=EMPTY Then Block(p) until state≠Empty
    eval QD=[..,y,x] → QD=[..,y],x
   If |QD|=0 Then state := Empty else state := FILLED
```

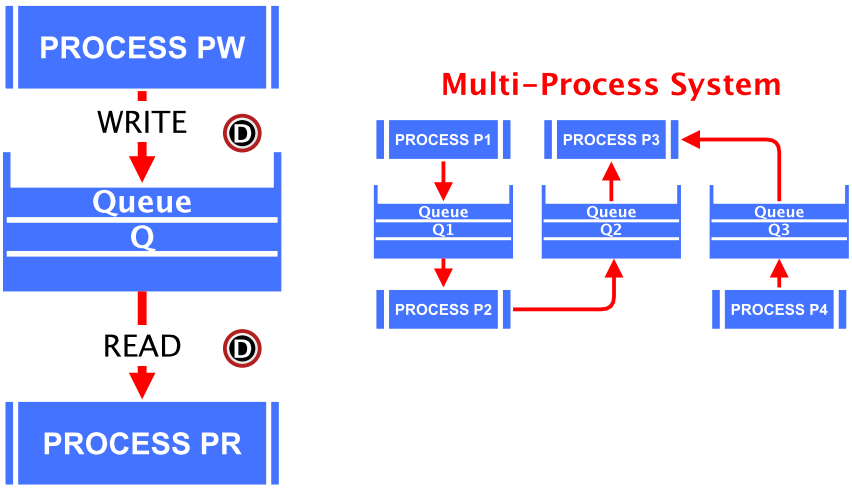

Fig. 5.6 *Inter-Process Communication with Queues (or channels)*

5.4 The extended CCSP Model

The original CSP model did not consider concurrency of multiple processes accessing a shared object with arising competition conflicts. The extended CSP model includes shared objects with access competition, which is resolved by atomic guarded actions applied to these objects. Global objects, for example, memory or synchronization objects, can be accessed by multiple processes concurrently without a violation of the object and process consistency. Invariants of objects and processes are preserved by using a Mutex scheduler resolving concurrency and competition by serialization (basically equal to process interleaving).

The principle multiprocess system architecture is shown in Figure 5.7. The system consist of a set of processes that access global shared objects by applying guarded operations.

It can be shown that the extended Concurrent CSP (CCSP) model can be efficiently mapped on hardware and software level implementations, as already proven in [BOS11A], [BOS10A], and [BOS13A] by different case studies including protocol stacks and agent processing platforms.

Global objects are accessed by a set of guarded operations (methods). A global object can be treated as a monitor locking the access to the resource by calling an operation applied to the object, for example, the read and write operations of memory objects or the lock and unlock operations of Mutex synchronization objects.

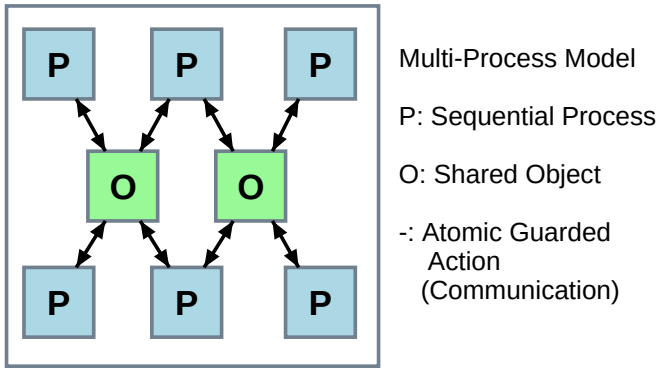

Fig. 5.7 *Multi-Process model with global shared objects and concurrent access*

The following subsections introduce some important synchronization objects and their operational semantic used in the CCSP model.

5.4.1 Atomic Registers

Concurrency and competition of multiple processes accessing a resource requires conflict resolution functions and objects. One common basic object used in the context of concurrent competition resolution enabling composition of higher-level synchronization objects is the atomic register, which insures consistent behaviour with multiple readers and writers, illustrated in Figure 5.8, based on [RAY13].

Constraints and behaviour:

1. A read or write operation op accessing an atomic register appears as executed at one specific time point $\tau(op)$.
2. The time point of the execution of a read or write operation lies guaranteed inside the interval $\tau(op_S) \leq \tau(op) \leq \tau(op_E)$, where op_S is the starting time and op_E the finishing time of the operation request (from the point of view of the process).
3. For two different operations it is always satisfied:
   $\tau(op_1) \neq \tau(op_2)$
4. Each read operation returns the value from the temporal nearest write operation execution in the past.

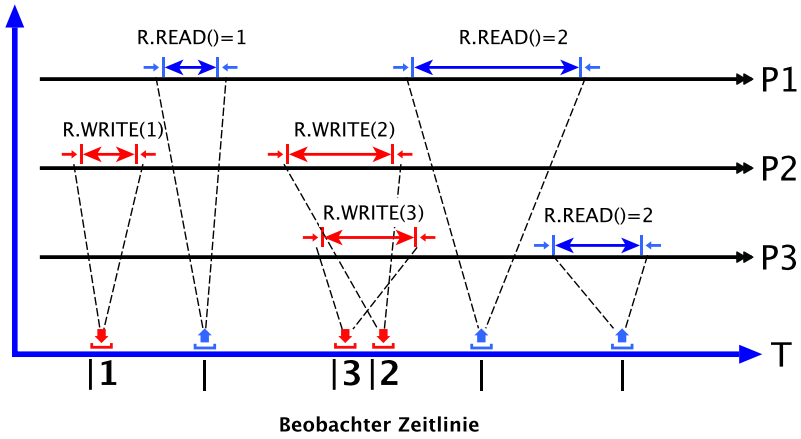

Fig. 5.8 *Atomic register: time-line of multiple partially or fully temporal overlapping read and write operations and the real execution (three concurrent processes P1,P2,P3).*

In the case of a concurrent access an atomic register behaviour is equal to the behaviour of a non-atomic register with strict sequential operational access (process interleaving)!

☞ The serialization of concurrent access of shared resources is the preferred approach used throughout this work to resolve the competition conflict in multiprocess systems based on mutual exclusion, discussed in the next sections.

5.4.2 Atomic Statements

For the implementation and satisfaction of mutual exclusion not only atomic registers are required, but also atomic statements. Though a mutual exclusion object (the Mutex) is used to make a sequence of statements atomic, e.g., non interruptible, special atomic statements are required on the processing architecture level. Some examples are test and set operations supported by microprocessors.

In the following section non-ordered atomic statement sequences are denoted by

$$| \, i_1, i_2, .. \, |,$$

those executions may not be interrupted, that means, the statements inside the bounded block must be executed within one time step.

5.4.3 Mutex

A mutual exclusion lock is used to protect and synchronize concurrent access of shared objects or program blocks accessing non-atomic objects like data structures. A Mutex can be in one of two states $s \in$ {LOCKED, FREE}, see Figure 5.9. A Mutex is an object accessed by the two operations ***acquire*** and ***release***. If the Mutex object is already locked, a process trying to acquire the lock will be blocked using the Block operation and the process control will be stored in a waiter list. If the Mutex is released by the owner, the next waiting process is scheduled. The algorithm is summarized in Definition 5.7. It assumes the control of the execution of processes by using the BLOCK and WAKEUP operation.

Mutex Invariant to be always satisfied: Maximal one process may be the owner of the lock and can pass the acquire operation without blocking.

Def. 5.7 *Mutex object signature and behaviour with a FIFO scheduler (pseudo-notation)*

```
object specification MUTEX =
  STATES = {LOCKED, FREE}

  operation acquire: process → unit
  operation release: process → unit

object behaviour MUTEX =
  VAR = atomic state:STATES
        atomic WAITERS: process list

  operation acquire(p) is
    If state=LOCKED Then
      eval p,WAITERS=[..] → WAITERS=[..,p]
      Block(p)
    state := LOCKED

  operation release(p) is
     If WAITERS ≠ [] Then
       eval WAITERS=[q,..] → WAITERS=[..],q
       WAKEUP(q)
     Else
      state := FREE
```

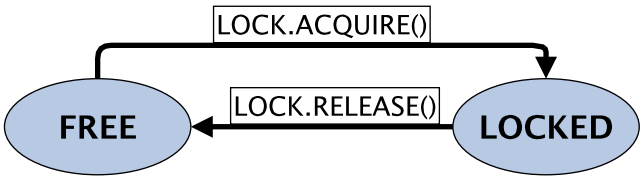

Fig. 5.9 *State change of a Mutex object by applying the acquire and release operation.*

The scheduling policy used to release blocked processes has significant effect on the overall multiprocess system behaviour, and eventually on the computational progress. Two different scheduling policies are considered: a dynamic FIFO and a strict static process priority driven approach:

Process Priority Scheduling

This scheduler serves and releases blocked processes waiting for a resource request execution in a strict order based on unique static process priorities assigned at compile or run-time. This scheduler is not starvation free, but can be implemented in hardware with the lowest resource requirements by using a mutual exclusive conditional branch based on combinatorial logic (state-free) [BOS10A] [BOS11A].

FIFO Scheduling

This scheduler serves and releases blocked processes waiting for a resource request execution in the first-in first-out order. This scheduler is starvation free, but has much higher resource requirements and is state-based. Software implementations should always use FIFO schedulers to guarantee fair scheduling.

5.4.4 Semaphore

A Semaphore object is a guarded counter with the following features:

1. Invariant $S.counter \geq 0$ is always satisfied
2. The Semaphore counter is initialized with a non-negative value s_0
3. There are two operations $\{up, down\}$ which modify the counter.
4. Extended invariant is always satisfied: $S.counter = s_0 + \#S.up - \#S.down$ with $\#S.up$ and $\#S.down$ being the number of up and down operation calls.

Only the down operation can block (lock) a requesting process iff the counter is zero. A process executing the up operation will release the waiting

blocked process without changing the counter value effectively in this case. A Semaphore is mainly used for cooperation management in multiprocess systems, for example, deployed in producer-consumer systems. A Semaphore can be implemented using a Mutex object, shown in Definition 5.8, basically implementing a Monitor object discussed in Section 5.4.8.

Each synchronization object is a shared global resource itself and always requires the resolution of the competition conflict, here realized with the Mutex and serialization scheduler.

A typical application of Semaphores used for the synchronization and coordination in producer-consumer systems is shown in Figure 5.10.

Def. 5.8 *Semaphore object signature and behaviour with a FIFO scheduler (pseudo-notation, | .. | denotes a guarded atomic statement execution)*

```
object specification SEMA =
  operation down: process → unit
  operation up: process → unit
  operation init: integer → unit

object behaviour SEMA =
  VAR = WAITERS: process list
         L: object MUTEX
        counter: natural

  operation down(p) is
     L.acquire(p);
    If counter = 0 Then
      eval p,WAITERS=[..] → WAITERS=[..,p]
      | L.release(p) , Block(p) |
     Else
      counter := counter - 1
       L.release(p)

  operation up(p) is
     L.acquire(p);
     If counter = 0 ∧ WAITERS ≠ [] Then
       eval WAITERS=[q,..] → WAITERS=[..],q
       WAKEUP(q)
     Else
       counter := counter + 1
     L.release(p)
```

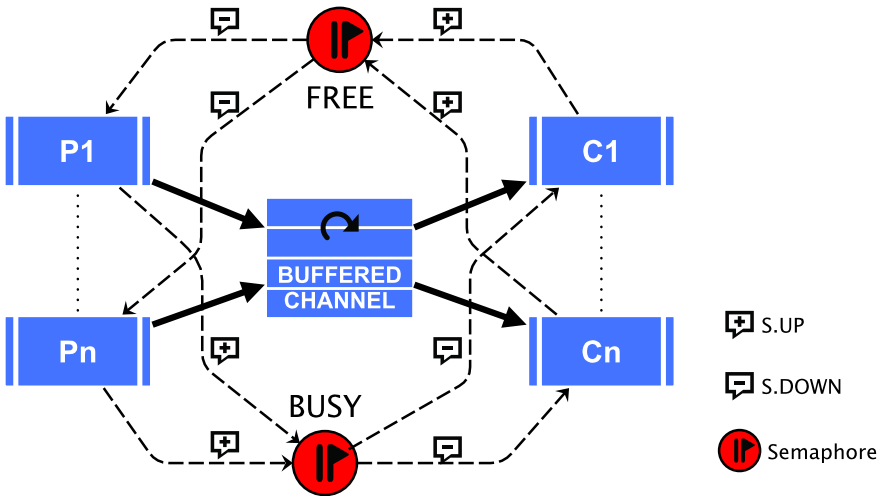

Fig. 5.10 *Producer-Consumer example using Semaphores FREE and BUSY implementing a synchronized buffer.*

5.4.5 Event

An event object is used to temporal synchronize a group of processes. There is a group of processes P_a waiting for an event. Until the event is signalled by a process $p_s \notin P_a$, all processes of P_a are blocked. If the event is signalled all processes of P_a are released at once. There are two operations {*await*, *wakeup*}, explained in Definition 5.9. The event is not starvation free. The event get lost if the event is raised before the processes perform the waiting for the event. To avoid this starvation risk, an event memory (latch) can be added, which is set if a wake-up operation meets an empty waiters list that is reset by the first process executing the *await* operation.

Def. 5.9 *Event object signature and behaviour (pseudo-notation, |..| denotes a guarded atomic statement execution)*

```
object specification EVENT =
  operation await: process → unit
  operation wakeup: process → unit

object behaviour EVENT =
  VAR = WAITERS: process list
         L: object MUTEX
       event: boolean

  operation await(p) is
     L.acquire(p);
     If not event Then
```

```
      eval p,WAITERS=[..] → WAITERS=[..,p]
      | L.release(p) , Block(p) |
    Else
      event ← false;
      L.release(p)

operation wakeup(p) is
    L.acquire(p);
    If WAITERS ≠ [] Then
      ∀{q ∈ WAITERS} do WAKEUP(q)
    Else
      event ← true;
    WAITERS ← []
    L.release(p)
```

5.4.6 Barrier

A barrier is similar to the event object, and it is a rendezvous object used by a group of processes. In contrast to the event is the barrier self-synchronizing. The size N of the process group P must be known in advance. The first N-1 processes will be blocked until the last process joins the group and releases all waiting processes. There is only one operation *await*.

Def. 5.10 *Barrier object signature and behaviour (pseudo-notation, |..| denotes a guarded atomic statement execution)*

```
object specification BARRIER =
  PARAMETER = {N:integer}
  operation await: process → unit

object behaviour BARRIER =
  VAR = WAITERS: process list
        L: object MUTEX
        count: natural

  operation await(p) is
    L.acquire(p);
    count++;
    If count < N Then
      eval p,WAITERS=[..] → WAITERS=[..,p]
      | L.release(p) , Block(p) |
    Else
      ∀{q ∈ WAITERS} do WAKEUP(q)
      count ← 0;
      L.release(p)
```

5.4.7 Timer

A Timer is similar to the Event object, too. The event is raised by an internal timer, which can be activated periodically or only one time.

Def. 5.11 *Timer object signature and behaviour (pseudo-notation, |..| denotes a guarded atomic statement execution)*

```
object specification TIMER =
  MODE = {once,interval}
  PARAMETER = {mode:MODE, timeout:integer}
  operation await: process → unit
  operation start: unit
  operation stop: unit

object behaviour TIMER =
  VAR = WAITERS: process list
        L: object MUTEX
       time: integer

  operation await(p) is
     L.acquire(p);
     eval p,WAITERS=[..] → WAITERS=[..,p]
     L.release(p);
    Block(p)

  operation start is
     L.acquire(p);
     time ← GlobalTime()+timeout;
     WAITERS ← [];
     ptimer.start();
    L.release(p)

  operation stop is
     L.acquire(p);
    ∀{q ∈ WAITERS} do WAKEUP(q);
     WAITERS ← [];
     ptimer.stop();
    L.release(p)

   process ptimer is
    wait for GlobalTime ≥ time;
     L.acquire(p);
    ∀{q ∈ WAITERS} do WAKEUP(q);
     WAITERS ← [];
    If mode=interval Then
       time ← GlobalTime()+timeout;
       L.release(p);
      ptimer.start();
     Else
       L.release(p);
```

5.4.8 Monitor

In contrast to the previously introduced objects, the monitor object is basically a programming paradigm and high-level construct, rather a synchronization object that is implemented directly on hardware or software level.

Mutex and Semaphores are low-level synchronization objects and the acquire/release or up/down operations must be used accordingly and balanced, otherwise deadlock or starvation effects can occur.

Monitors implement synchronization on a higher abstraction level of a programming language and provide automatic locking without the risk of deadlocks or starvation, shown in Figure 5.11.

A Monitor encapsulates data and operations on programming level and ensure the mutual exclusion. It satisfies that almost one process is active within an operation of a Monitor and satisfies the invariance (consistency) of the Monitor data. A Monitor uses event variables to enable process interleaving under preservation of a predicate, the invariant of a monitor [PARSYS].

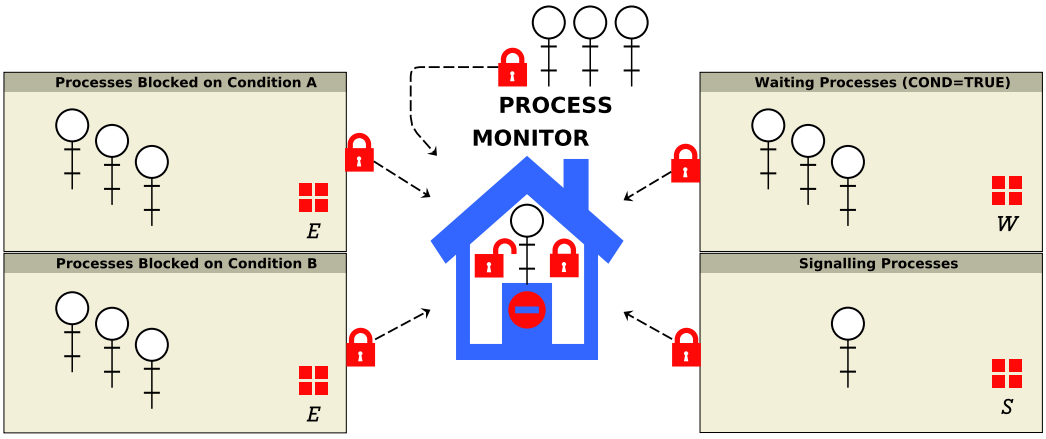

Fig. 5.11 *A Monitor is used to auto-synchronize the mutual exclusive access of an object by applying operations.*

5.5 Signal Flow Diagrams, CSP, and Petri-Nets

Signal flow, a common data processing model in digital signal processing applications, can be transformed immediately to a pipelined CSP process-channel architecture. Petri-nets are used for an intermediate representation in the final synthesis of CSP systems implementing the behaviour of the signal flow diagram.

Figure 5.12 shows an example for the composition of a signal flow diagram of a PID controller. An explanation can be found in Section 13.1

The signal flow diagram is first transformed into an S/T Petri Net representation, which is shown in Figure 5.13. Functional blocks are mapped to transitions, and states represent data that is exchanged between those functional blocks. The partitioning of functional blocks to transitions of the net can be performed at different composition and complexity levels. The signal flow diagram from Figure 5.12 was partitioned using complex blocks (merging low-level blocks like multipliers and adders) to reduce communication complexity (and data processing latency).

The Petri Net is then used 1. To derive the communication architecture; and 2. To determine an initial configuration for the communication network. Functional blocks with a feedback path require the injection of initial tokens in the appropriate states (not required in the example).

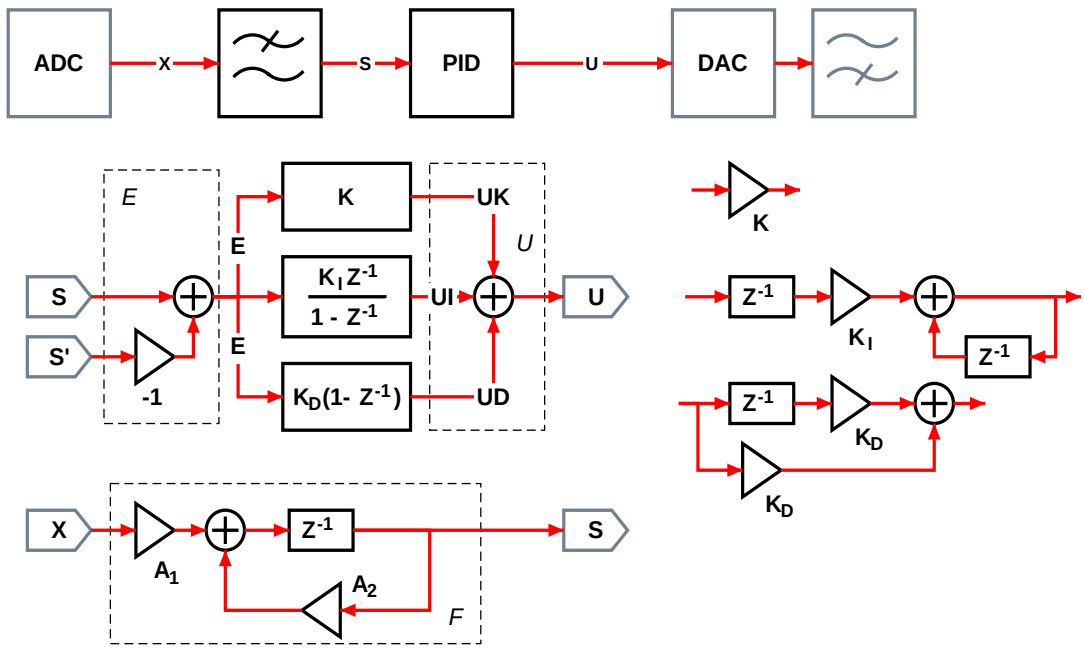

Fig. 5.12 *Composition and modelling of a digital control system with signal flow diagram* [BOS11B]

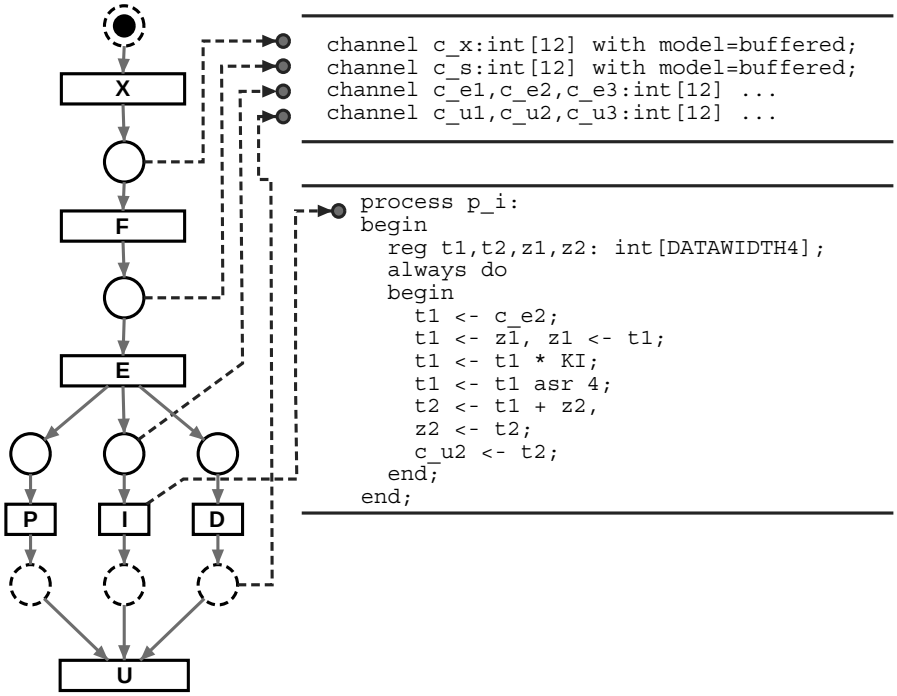

Fig. 5.13 *Mapping of the signal flow diagram to a Petri Net and mapping of Petri Net to communication channels and sequential processes using the ConPro programming language* [BOS11B].

States of the net are mapped to buffered communication channels and transitions are mapped to parallel and concurrently executing processes - each with sequential instruction processing - using the *ConPro* programming language, introduced in the next section, shown in Figure *5.13*, too.

Data sets are represented by tokens that are passed between the processes by the communication channels. A process can consume and process only one token each time. Queues can be used instead of channels (that cannot store more than one data set token), releasing processes immediately after they transferred a token to a queue, and prevent the waiting for the consumer of a token.

Forked states indicate concurrency in the Petri Net flow. Exploring concurrency in signal flow diagrams using Petri Nets reduces latency for the complete computation of one data set, starting with the reading of a new data set by the first block, and ending by the writing of the computed output by the last block of the computational system. Also, pipe-lining can improve the throughput of data set streaming significantly, offered inherently by the Petri Net representation.

5.6 CSP Programming Languages

There are several programming languages related to the CSP model. The original CSP model is available in the *OCCAM* programming language [OCC95]. The *OCCAM* programming languages supports sequential, parallel, and alternate process constructors, which can be nested arbitrarily. An *OCCAM* program can be executed on one or multiple processors, originally related to the Transputer architecture [IVI99]. The *OCCAM* programming model does not support true concurrency and shared object access with competition. Due to the lack of suitable programming languages for high-level synthesis related to the extended *CCSP* model, the *ConPro* programming language was designed, close to the *CCSP* model [BOS11A][BOS10A].

5.7 The μRTL Programming Language

The parallel programming language *Conpro*, which is introduced in the next section, maps sequential processes on RTL process blocks consisting of a Finite-State Machine (FSM) and a register-based data path. The behaviour of an RT process block is specified with an intermediate representation, given by the *μRTL* language introduced in this section (based on [BAR73]). This language can be seen as a core language to model and understand the behaviour of the extended concurrent CSP programming model.

Operation	*Description*
$r \leftarrow expr$	Assignment of the value to a register.
i_1 ; i_2 ; i_3 ; ...	Sequential execution of statements.
i_1 , i_2 , i_3 , ... ;	Parallel execution of statements (limited to non-blocking statements, i.e., assignments)
branch *s*	An unconditional branch to a different instruction related to the state *s* before or after this statement.
branch *cond*:*s*	Conditional branch if the condition is satisfied, otherwise the next instruction is executed.

Tab. 5.1 *Minimal symbolic parallel programming language that can be immediately synthesized to hardware RT architectures (ε: computational expression).*

Operation	Description
s:	An instruction label assigning a state to an instruction.
$o \rightarrow op(arg_1, arg_2, \ldots)$	Procedural application of an operation to a global shared object with optional arguments.
$r \leftarrow \varepsilon(o \rightarrow op(arg_1, arg_2, \ldots))$	Functional application of an operation to a global shared object with optional arguments returning a value.

process $p \rightarrow$ {
i_1 ; i_2 ; i_3 ; ...
}
	Definition of a named process composition. A process p is treated as an object, too, supporting start and stop operations.

Tab. 5.1 *Minimal symbolic parallel programming language that can be immediately synthesized to hardware RT architectures (ε: computational expression).*

An example deriving the μ*RTL* specification from the *OCCAM* programming language is shown in Example 5.1.

Ex. 5.1 *A μRTL specification derived from an OCCAM program*

OCCAM

```
[100] INT Data:
INT Mean:
SEQ
  Mean := 0
  SEQ Index = 0 FOR 100
    Mean := Mean + Data[Index]
  Mean := Mean / 100
```

μRTL

```
process main → {
  s1: Index ← 0, Mean ← 0;
  s2: branch (Index=100):s6;
  s3: Mean ← Mean + Data[Index];
  s4: Index ← Index + 1;
  s5: branch s2;
  s6: Mean ← Mean / 100;
}
```

The relationship of the *μRTL* instruction to the previously introduced process algebra and the process flow behaviour of the μ*RTL* instructions is given below (♦: Synchronisation and scheduling point).

Process Flow Behaviour

$r \leftarrow \varepsilon$	$\Rightarrow \{r \leftarrow \varepsilon\} \Rightarrow \blacklozenge$
i_1 ; i_2 ; ..	$\Rightarrow \{i_1\} \Rightarrow \blacklozenge \Rightarrow \{i_2\} \Rightarrow ..$
i_1 , i_2 , ..	$\Rightarrow \{i_1\} \parallel \{i_2\}.. \Rightarrow \blacklozenge$
$o \rightarrow op()$	$\Rightarrow \{o \rightarrow op()\} \Rightarrow \Diamond \neg guard(op) \Rightarrow$
branch *cond*:s	$\Rightarrow (\neg cond \Rightarrow \{i_{n+1}\} \mid cond \Rightarrow \{i_s\}) \Rightarrow \blacklozenge$

5.8 The ConPro Programming Language

Concurrency is modelled explicitly on the control path level with processes executing a set of instructions sequentially, initially independent of any other process. Parallel composition is available on programming level but is static at run-time to enable the synthesis of SoC designs with static resources. That means the number of processes is fixed at design and compile time.

Inter-Process Communication (IPC) provides process synchronization with different objects (Mutex, Semaphore, event, timer) and data exchange between processes using queues or channels, based on the CSP model. More fine-grained concurrency is provided on data path level using bounded blocks executing several instructions (only data path, e.g., data assignments) in one time unit. Block level parallelism can be enabled explicitly or implicitly explored by a basic block scheduler.

Hardware blocks, modelled on hardware level (*VHDL*), can be accessed from the programming level by two methods:

1. Using an object-orientated programming approach with methods
2. Using a signal component structure interface and signals

The first approach treats all hardware blocks, including IPC, as abstract data type objects (ADTO) with a defined set of methods accessible on process level (at run-time) and on top-level (only applicable with configuration methods, for example, setting the time interval of a timer). The bridge between the hardware and *ConPro* programming model is provided by the External Module Interface (*EMI*).

A signal component structure can be used to instantiate and access external hardware blocks and to create the top-level hardware port interface of the *ConPro* design. Signals are interconnection elements without a storage model.

They provide only an interface to external hardware blocks. Signals are used in component structures, too.

The *ConPro* programming model is a common source for hardware and alternatively software synthesis, reusing the same program source for different implementations, shown in Figure 5.14 on the left side. The *EMI* programming paradigm is a central part of this multi-target implementation synthesis by encapsulating and abstracting hardware blocks, for example, IPC and communication objects. Most *EMI* objects can be specified by a hardware (*VHDL*) behaviour and an operational equivalent and complementary software model, without compromising the programming level, shown in Figure 5.14 on the right side.

In Figure 5.16 (see page 181) the *ConPro* programming language block structure and the composition of top-level module designs using these blocks are summarized.

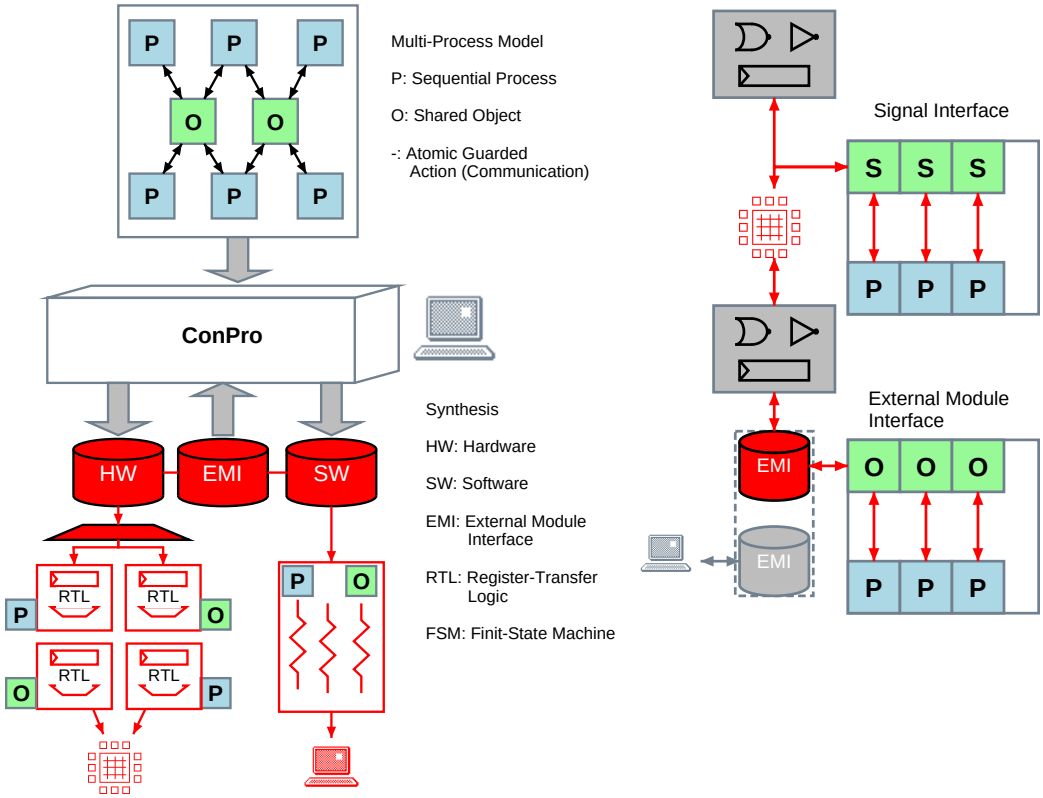

Fig. 5.14 *Left: The ConPro programming model as a common source for hardware and alternatively software synthesis. Right: Hardware Interfaces from programming level, (a) Signal Interface, (b) EMI*

5.8.1 Process Composition

Processes are the main execution units of a *ConPro* design module. A process executes a set of instructions in a sequential order. Processes communicate with each other and the environment by using shared objects (Inter-process communication IPC). There is sequential and parallel process composition on control and data path level using bounded blocks, shown in Figure 5.15.

Parallel process composition on control path level is flat and provides only coarse-grained concurrency. Parallel process composition on data path level inside a process is fine-grained, but is limited to non-blocking statements not affecting the control path of a process.

The *ConPro* programming language supports named processes defined on the top-level only, in contrast, for example, to the *OCCAM* programming model, which allows embedded and nested parallel and sequential process constructors anywhere.

A process definition consists of a header defining the unique process name and the process body defining local object definitions (types, data and some abstract objects) and sequence of statements.

Processes are abstract objects, too. Thus, there is a set of methods {`start`, `stop`, `call`} which can be applied to process objects (identifiers). They are controlling the state of a process.

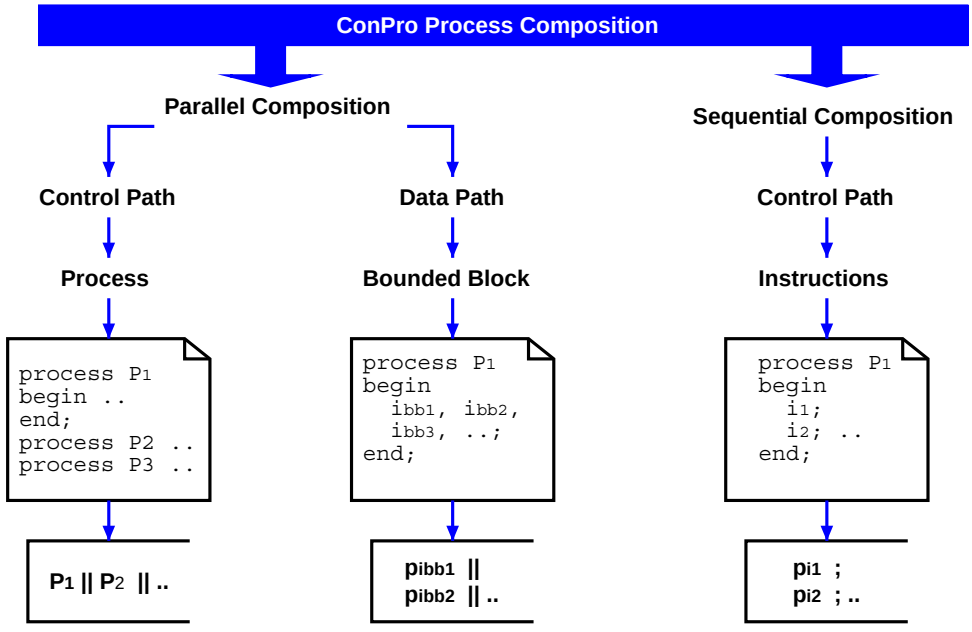

Fig. 5.15 *Different levels of concurrency and process composition in ConPro*

A process must be started by another process explicitly by using the `start` method, which starts the process concurrently (asynchronous call). A process can be started synchronously by using the `call` method, which suspends the calling process until the called process terminates. Each process can be stopped asynchronously by using the `stop` method. There is only one main process, which is started automatically at start-up. To simplify the definition of a set of processes, parametrizable process arrays can be defined, sharing the same process body definition, which are replicated to different and independent physical processes.

Shared program functions and procedures are implemented with sequential processes, too. Function processes are started by a calling process by using the call method applied to the function process object, which suspends the calling process until the function process has terminated. An additional function call wrapper is attached to a function process to serialize concurrent function calling (Mutex scheduler).

Def. 5.12 *ConPro process definition and control using object method applications. Processes are objects that can be parametrized (i.e., synthesis parameters).*

```
process pname:
begin
  definitions
  statements
end ⟦ with parameter=value ⟧;

array paname : process[N]
begin
  definitions
  statements
end ⟦ with parameter=value ⟧;

...
     pname.start();
pname.stop();
pname.call();
     paname.[index].start();
...
```

5.8.2 Data Storage Objects

True bit-scalable data types (TYPE β) and storage objects (subset $\Re$ of TYPE α) are supported. The data width can be chosen in the range δ=1-64 Bit. A data storage object $\Re$ is specified and defined by a cross product of types ($\alpha \times \beta$). Storage objects can be read in expressions and can be modified in assignments.

Registers are single storage elements either used as a shared global object or as a local object inside a process. In the case of a global object, the register provides concurrent read access (not requiring a Mutex guarded scheduler) and exclusive Mutex guarded write access. If there is more than one process trying to write to the register, a Mutex guarded scheduler serializes the write accesses. There are two different schedulers available with static priority and dynamic FIFO scheduling policies.

Variables are storage elements inside a memory block either used as a shared global object (the memory block itself) or as a local object inside a process. A variable always provides exclusive Mutex guarded read and write access by the memory block to which it belongs.

Different variables concerning both data type and data width can be stored in one or different memory blocks, which are mapped to generic RAM blocks. Address management is done automatically during synthesis and is transparent to the programmer. Direct address references or manipulation (aka pointers) are not supported.

The memory data width, always having a physical type logic/bit-vector, is scaled to the largest variable stored in memory. To reduce memory data width, variables can be fragmented, that means a variable is scattered over several memory cells.

Different memory blocks can be created explicitly, and variables can be assigned to different blocks.

Queues are storage elements with FIFO data transfer order and synchronized inter-process communication objects, too. They are always used as shared global objects. Queues and channels can be read directly in expressions and can be used on the left-hand side of assignments like any other storage object.

Channels are similar to queues. But they can be buffered (behaviour like a queue with one cell, depth is 1) or unbuffered (providing only a handshake data transfer).

Def. 5.13 *ConPro data storage and signal object definitions and data types with optional parametrization (DT: Data type, width: bit width including sign bit, ⟦..⟧: optional)*

```
reg name : DT ⟦ with parameter=value ⟧;
var name : DT ⟦ in blockname ⟧;
block name;
sig name : DT;
queue name : DT ⟦ with depth=value ⟧;
channel name : DT ⟦ with depth=value ⟧;
TYPE DT = { logic, logic[width], int[width], bool, char }
TYPE OT = { reg, var, sig, queue, channel }
```

Register, variables, queues, and channels can be defined for product types (arrays and structures), and sum types (enumeration types).

5.8.3 Signals and Components

Signals are interconnection elements without a storage model. They provide an interface to external hardware blocks. Signals are used in component structures, too.

Signals can be used directly in expressions like any other storage object. Signals can be read in expressions, and a value can be assigned using the assignment statement, Reading a signal returns the actual value of a signal, but writing to a signal assigns a new value only for the time the assignment statement is executed (active), otherwise a default value is assigned to the signal. Therefore, there may be only one assignment for a signal.

Signals are non-shared objects, and have no access scheduler. Only one process may assign values to a signal (usually using the `wait for` statement), but many processes may read a signal concurrently. Additionally, signals can be mapped to register outputs using the `map` statement. The definition of a signal object is shown in Definition 5.13.

5.8.4 Arrays and Structure Types

Arrays can be defined for storage and abstract object types, shown in Definition 5.14. Arrays can be selected with static (index known at compile time) and dynamic selectors (variable expressions).

Def. 5.14 *ConPro array definition and array element access (size: number of array elements)*

```
array AS: OT[size] of DT ⟦ with parameter=value ⟧;
array AO: object obj[size] ⟦ with parameter=value ⟧;
array AO: process[size] of begin .. end
          ⟦ with parameter=value ⟧;
... AS.[index] ...
... AO.[index].method ...
```

A structure type definition contains only data types (DT), and no object types (OT). There are three different subclasses of structures for different purposes, formally described in Definition 5.15:

Multi-Type Structure

The generic structure type binds different named structure elements with different data types to a new user defined data type, the native product type.

Bit Structure

This structure sub-class provides a bit-index-name mapping for storage objects. All structure elements have the same data type. The bit-index is either a one bit value or a range of bits. This structure type provides the symbolic/named selection of parts of vector data types (for example, logic vector or integer types) and specified the bit access of object data.

Component Structure

This structure defines hardware component ports, either of a ConPro module top-level port, or of an embedded hardware component (modelled on hardware level). This structure type can only be used with component object definitions. The component type has equal behaviour like the signal type.

Def. 5.15 *ConPro structure type definitions and element selection*

```
Define a new structure type with a data type DT specification for
each element:
     type ST: {
       E1: DT;
       E2: DT;
       ...
     };
     reg instance:ST;

Define a new component structure type with data type DT
and signal direction DIR specification for each element.
     type CT: {
       port E1: DIR DT;
       port E2: DIR DT;
       ...
     };

Define a new bit structure type with bit-width specification
for each element.
     type BT: {
       E1: width;
       E2: width;
       ...
     };
Access of type object elements
     ... obj.elem ...
```

In the case the object type of a structure instantiation is a register, just N independent registers are created. In the case of a variable object type, N objects are stored in a RAM block. Arrays from structure types can be created. For each structure element a different array is created. Hardware component port types are defined with structures, too, with the difference

that for each structure element the direction of the signal must be specified. Some care must be taken for the direction: if the component is in lower hierarchical order (an embedded external hardware component), the direction is seen from the external view of the hardware component.

If the component is part of the top-level port interface of a *ConPro* module, it must be seen from the internal view.

5.8.5 Exceptions and Handling

The *ConPro* programming model supports the concepts of exception signals and exception handling with statements. Exceptions provide the only way to leave a control structure, for example loops, conditional branches or functions itself. Exception are abstract signals, which can be raised anywhere and caught within a `try-with` exception handler environment, either within the process/function where the exception was raised, or outside. Thus, exceptions are automatically propagated along a call path of processes and functions using exception state registers if they are not caught within the raising process/function.

Def. 5.16 *ConPro definition of exceptions and exception handlers*

Definition of an exception

```
exception E;
```

Raising of an exception (inside processes)

```
... raise E; ...
```

Catching of raised exceptions

```
try I with
begin
  when E1: I1;
  when E2: I2;
  ...
end;
```

5.8.6 Functions and Procedures

A function definition consists of a unique function name identifier, the function parameter interface, and the function body with statements. The function body consists of local object definitions (types, data and some abstract objects) and an instruction sequence.

epubli, ISBN 9783746752228 (2018)

Each function parameter and the set of return value parameters are handled like registers. Only call-by-value semantic is supported. Values of function arguments are copied to the respective parameters before the function call, and return values are copied after function call has finished. Within the function body, all parameters and the (named) return parameter can be used in assignments and expression like any other register. There is no return statement. The last value assigned to the return parameter is automatically returned.

There are two different types of functions: in-lined and shared. The in-lined function type is handled like a macro definition. Each time a function is applied (called), the function call is replaced by the function body, and all function parameters are replaced by the function arguments (including return value parameters). Shared functions are implemented using a sequential process and the call method with an additional function call wrapper. Each time a shared function is called the argument values are passed to the function parameters (global registers), and the return value(s) are passed back, if any.

Def. 5.17 *ConPro definition and application of functions and procedures*

```
function fname (p1:DT,p2:DT,..) [[ return (r1:DT,..) ]]:
begin
  definitions
  instructions
end;

pname();                 procedure call
pname(i,1);
y ← fname1(x);           function call returning a value
{y1,y2} ← fname2(x);     function call returning two values
```

5.8.7 Modules

A *ConPro* design hierarchy consists of a behavioural module level (Module-B) containing global (shared) objects and processes. A module is mapped to a circuit component with a top-level hardware port interface. Structural modules (Module-S) can be composed of behavioural modules with optional internal interconnect components. Each process (and process level) consists of local (non-shared) objects and a process instruction sequence, specifying the control and data flow. Abstract object types are implemented with abstract object modules (Module-O).

Behavioural modules implement objects and processes. A behavioural module is defined by the source code file itself. Actually there is only one module hierarchy level, the main module. More source code files can be

included using an include statement. There are two kinds of behavioural modules: a module embedding objects and processes, and modules providing access and implementation of abstract data type objects (ADTO). These are mainly inter-process communication and synchronization objects, for example Mutex, semaphore, timer and some communication links. Each ADTO module to be used must be opened using the open statement.

Structural modules are used to build System-On-Chip (SoC) circuits from behavioural modules.

Def. 5.18 *ConPro definition of structural modules and structural composition*

```
Design toplevel IO port definition:

     type top_io_type:{
       port S1: DIR DT;
     };
     component TOP: top_io_type;
     export TOP;

Define a new structural module with specified name:
module MS

     begin
       import
       component
       structtype
       mapping
     end;

Import of behavioural modules and instantiation of components
(ciruits):

     import MB;
     component C1,C2,..:MB;

Defines and instantiate a port interface of interconnect
component with initial signal mapping:

     type ICT:{ port...};
     component IC:ICT :=
     {
       C1.TOP.S1,
       C1.TOP.S2,...
       C2.TOP.S1,...
     };

Internal interconnect using mapping statements:
          IC.S1 >> IC.S2;
```

The definition of the structural composition using IO port structures, component instantiations, and module block is shown in Definition 5.18. The structural composition aids the designing of complex SoC circuits by reusing multi-process blocks.

5.8.8 The ConPro Building Blocks

The overall *ConPro* design hierarchy and the compositional blocks are shown in Figure 5.16. Top-level modules are composed of process definitions, global objects (storage, synchronization, communication), and user-defined types, and finally top-level interface ports connecting processes and objects with outside world. The structural composition can embed hardware blocks on top-level by using components. Processes are composed of instructions and local storage objects.

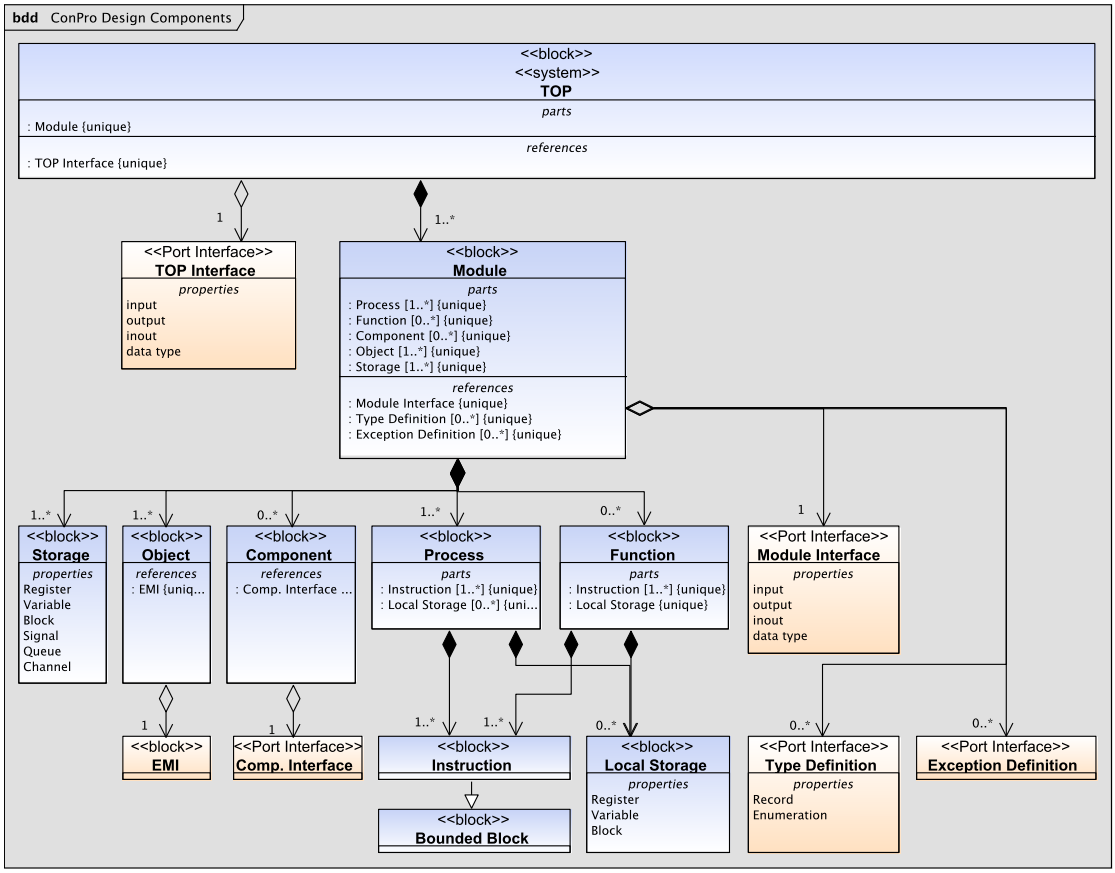

Fig. 5.16 *ConPro Design Hierarchy and Compositional Blocks Diagram*

5.8.9 Control and Data Processing Statements

Data Processing

Data processing is performed by evaluating expressions and storing computation results in registers or variables using assignments. An assignment statement has a target data object (register, variable, or a signal), the left-hand side (LHS), and an expression, the right-hand side (RHS), which can be arbitrarily nested and consists of arithmetic, relational, and boolean operations with operands accessing registers, variables, or signals. As like in traditional imperative programming models, there is a strict sequential order: first the RHS is evaluated, and then the result is stored, which is commonly ensured by an appropriate hardware model in the data path and can be processed in one control state. Assignments can only be used in process body blocks.

Def. 5.19 *ConPro computational statements (assignment and expressions)*

Assignment
var *x*: *DT*; ‖ reg *x*: *DT*; ‖ sig *x*: *DT*;
x <- *expression*;

Micro-Sequential timing model: *result*=Eval(*expression*) ; Write(*x*, *result*)

Expressions
expression ::= *value* | *expression operator expression* .
operator ::= + - * / % land lor lnot band bor bnot lsl lsr .

Bounded Blocks

Instruction blocks can be used to bind instructions, either part of a control statement like a branch, or used to define a sub-process with a sequential or parallel execution model. Instruction can be parametrized, attaching specific processing or synthesis settings, shown in Definition 5.20.

Def. 5.20 *ConPro Sequential and Parallel Process Constructor using blocks.*

```
begin  seq. sub-process [[ begin ]] parallel sub-pro.
  i1;                          i1, i2, i3, ..
  i2;[[ end ]] [[ with bind ]];
  ..
end with [[ parameter=value and .. ]];
```

Parameters: { bind, unroll, schedule, inline, ..}

Conditional Branches

Two different conditional branches exist: a mutual branch (if-then-else) based on a boolean condition, and multi-case branch matching values with the outcome of an expression evaluation (match-with-when). Mutual branches can be chained (if-then-else-if) offering a prioritized and ordered conditional branching. In contrast, the multi-case branch can be parallelized in the control and data path regarding the evaluation of the matching condition, offering a branch with constant processing time for each case matching.

Def. 5.21 *ConPro Branches*

```
if cond then i_cond=true [ else i_cond=false ];

match expr with
begin
  when v1: i_expr=V1;
  when v2,v3, ..: i_expr=v2∨expr=v3..;
  ..
[ when others: i_others; ]
end;
```

Loops

The following loops are offered by the programming model, with the syntax defined below. In addition to common loop statements there are special loops addressing the requirements in parallel process systems with event-based client-server/request-service operational semantic and the handling of hardware logic signals.

Counting Loop [for-do]

A counting loop repeats the instruction execution of the loop body with an incrementing or decrementing loop variable. The number of loop iterations can be given by static or dynamic limit values (registers, variables). Loops can be completely unrolled. On top-level, the for-loop can be used to generate an unrolled sequence of top-level instructions, for example, mapping instructions using arrays. Top-level for-loops will always be unrolled, therefore the unroll parameter is unnecessary

Conditional Loop [while-do]

The loop body is executed as long as a boolean expression evaluates to the value true.

epubli, ISBN 9783746752228 (2018)

Unconditional Loop [always-do]

The loop body is executed repeatedly without any condition. Used in event-based service processes. An unconditional loop can only be terminated by stopping the process or by raising an exception.

Constant Delay [wait-for-time]

The process can be suspended (delayed) exactly for a specified amount of clock cycles or time units.

Conditional await [wait-for-condition]

This statement blocks the process until a boolean expression is true. In the case the expression is false and the wait statement is blocked, signal assignments can be applied optionally for this time. The expression generally operates on global signals or registers that may not be guarded.

Constant clock cycles application [wait-for-time-with]

For a specified number of clock cycles simple expressions can be evaluated and assigned to signals. A signal on the LHS in an assignment holds a value only for one clock cycle. A signal in this apply statement holds the value for N clock cycles. Outside the application statement the signal must be assigned with a default value. This can be specified optionally in the apply statement

Def. 5.22 *ConPro Loops*

```
for x = a to ⟦ downto b ⟦ step v_s ⟧ do i_loop ;

while cond do i_loop ;

always do i_loop ;

wait for time;
wait for cond;
wait for time with i_0;
wait for cond with i_0 ⟦ else i_1 ⟧;
```

5.9 Hardware Architecture

The previously introduced extended CCSP model can be directly implemented and mapped on microchip level [BOS11A][BOS10A].

5.9.1 Processes

Each *ConPro* process is implemented on hardware level with a Finite State Machine (FSM) and a Register-Transfer Level (RTL) architecture of the data path, finally creating an interconnected multi-FSM architecture.

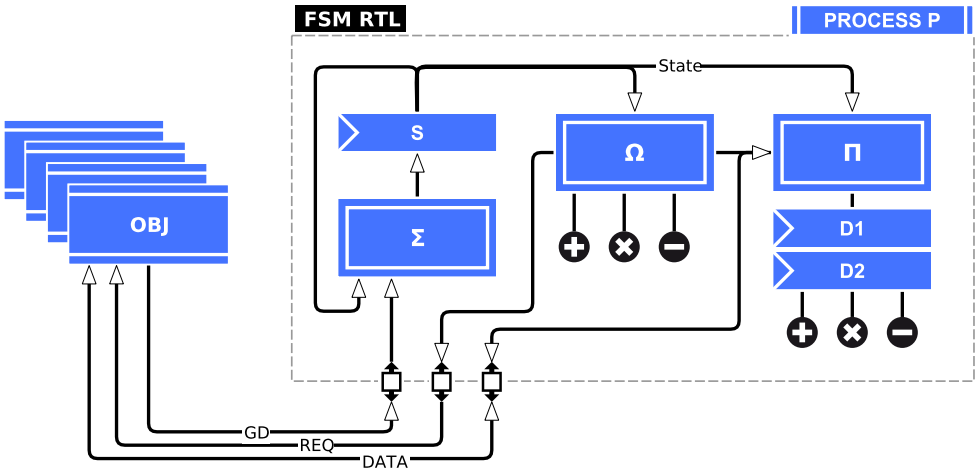

Fig. 5.17 *Hardware architecture of a sequential ConPro process and interconnect to global objects (Σ: state transition network, S: control state register, Ω: combinatorial data path, Π: transitional data path)*

The program flow of the instructions of a *ConPro* process are mapped on a control flow and states of the FSM, shown in Figure 5.17, with an additional start and end state for each process. Complex instructions are split into multiple states. The data path is divided into a pure combinational and a transitional part with registers. The first one accesses local (read only) and global objects (read, write, and control access). Access of global resources is always guarded by the access scheduler (mutual exclusion lock) of the shared object. A request signal (`RE/WE` or `CALL`) is activated in the data path. A guard signal `GD` is read and evaluated by the finite state machine transition network. The actual state, accessing the object, is hold until the guard signal changes to low level (happening only for one clock period to avoid race conditions).

Shared program functions and procedures are implemented with sequential processes, too.

5.9.2 Modules

All hierarchical modules, processes, objects, and instantiated hardware components are merged into one (flat) SoC design with one top-level port interface, shown in Figure 5.18.

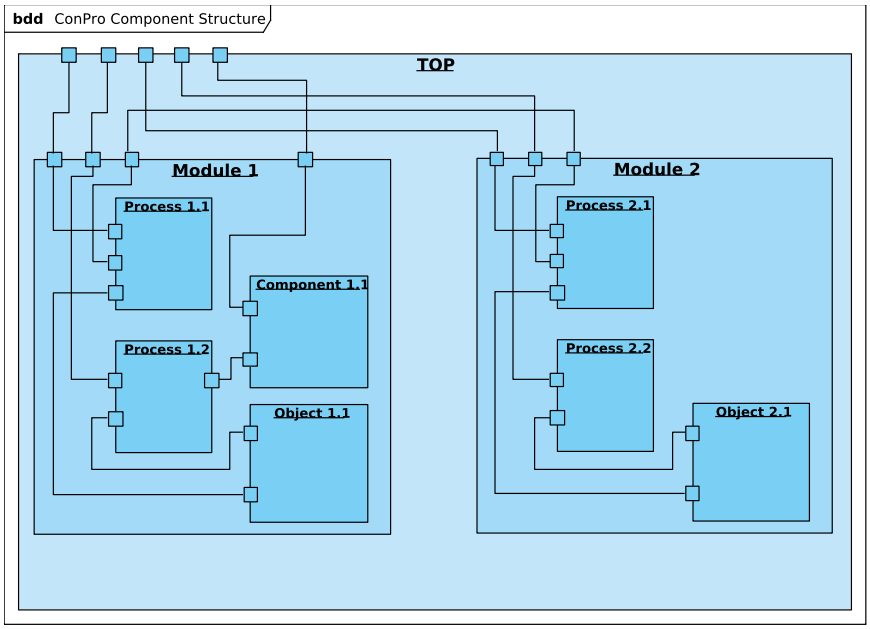

Fig. 5.18 *ConPro Hierarchical Block Architecture with Interconnection.*

5.9.3 Mutex Scheduler

Access of shared objects must be guarded inherently by a Mutex using a mutual exclusion scheduler. This scheduler is responsible to serialize concurrent access. The scheduler is connected with all processes accessing the resource. A process activates a request (REQ), and waits for the release of the guard (GD), which unblocks the process signalling the grant of the resource, shown in Figure *5.19*.

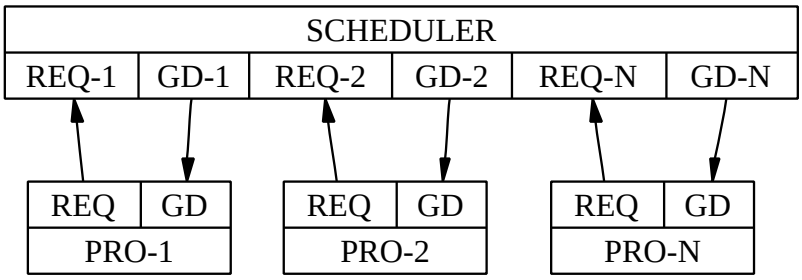

Fig. 5.19 *Scheduler Block Architecture and process interconnect*

There are two different schedulers available:

Static Priority Scheduler [default]

This is the simplest scheduler and requires the lowest amount of hardware resources. Each process ever accessing the resource gets a unique ordered priority. If there are different processes accessing the resource concurrently, the scheduler always grants access to the process with the highest priority. There is a risk of race conditions using this scheduling strategy. Commonly, the order of processes appearing in the source code determines their priority: the first process gets the highest priority, the last the lowest. A scheduled access requires at least two clock cycles.

Dynamic FIFO Scheduler

The dynamic scheduler provides fair scheduling using a process queue. Each process requesting the resource and looses the competition is stored in a FIFO ordered queue. The oldest one in the queue is chosen by the scheduler if the resource is released by the previous owner. The dynamic scheduler avoids race conditions, but requires much more hardware resources.

The algorithms for both schedulers are defined in Definitions 5.23 and 5.24.

Def. 5.23 *Static Priority Scheduler: From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}. A process-i request activates REQ-i, and if the resource is not locked, the request is granted to the next process in the if-then-else cascade. If the request is finished, then ACK is activated and releases the locked object and releases GD-i for this respective process indicating that the request is finished.*

```
Loop Do
  ACT ← False;
  ∀ gd ∈ {GD-1,GD-2,..} Do gd ← True;
  If REQ-1 ∧ ¬LOCKED Then
    LOCKED ←True;
    ACT ← True;  Start Service for Process 1
  Else If REQ-2 ∧ ¬LOCKED then
    LOCKED ←True;
    ACT ← True;  Start Service for Process 2
  ...
  Else If ACK ∧ REQ-1 ∧ LOCKED Then
    GD-1 ← False;
    LOCKED ← False;
  Else If ACK ∧ REQ-2 ∧ LOCKED Then
    GD-2 ← False;
    LOCKED ← False;
  ...
```

Def. 5.24 *Dynamic Queue Scheduler: From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}. A process-i request activates REQ-i, and if the resource queue is empty or this process is at head of the queue, the request is granted to the process in the if-then-else cascade. If the request is finished, then ACK is activated and removes the process from the resource queue and releases GD-i for this respective process indicating that the request is finished.*

```
Loop Do
  ACT ← False;
  ∀ gd ∈ {GD-1,GD-2,..} Do gd ← True;

  If REQ-1 ∧ LOCKED=[] ∧ ¬PRO-1-LOCKED Then
    LOCKED ← [PRO-1];
    PRO-1-LOCKED ← True;
    OWNER ← PRO-1;
    ACT ← True;           Start Service for Process 1
  Else If REQ-2 ∧ LOCKED=[] ∧ ¬PRO-2-LOCKED Then
    LOCKED ← [PRO-2];
    PRO-2-LOCKED ← True;
    OWNER ← PRO-2;
    ACT ← True; Start Service for Process 2
  ...
  Else If REQ-1 ∧ LOCKED≠[] ∧ ¬PRO-1-LOCKED Then
    LOCKED ← LOCKED @ [PRO-1]; Append Process 1 to Queue
    PRO-1-LOCKED ← True;
  Else If REQ-2 ∧ LOCKED≠[] ∧ ¬PRO-2-LOCKED Then
    LOCKED ← LOCKED @ [PRO-2]; Append Process 2 to Queue
    PRO-2-LOCKED ← True;
  ...
  Else If REQ-1 ∧ Head(LOCKED)=PRO-1 ∧ OWNER≠PRO-1 Then
    ACT ← True;           Start Service for Process 1
    OWNER ← PRO-1;
  Else If REQ-2 ∧ Head(LOCKED)=PRO-2 ∧ OWNER≠PRO-2 Then
    ACT ← True;           Start Service for Process 2
    OWNER ← PRO-2;
  ...
  Else If ACK ∧ Head(LOCKED)=PRO-1 Then
    GD-1 ← False;
    PRO-1-LOCKED ← False;
    OWNER ← NONE;
    LOCKED ← Tail(LOCKED);
  Else If ACK ∧ Head(LOCKED)=PRO-2 Then
    GD-2 ← False;
    PRO-2-LOCKED ← False;
    OWNER ← NONE;
    LOCKED ← Tail(LOCKED);
  ...
```

5.9.4 Functions

Shared functions are implemented with processes and an additional function scheduler guarding the concurrent access of functions and function parameter. Input and output parameters of functions are implemented with global registers.

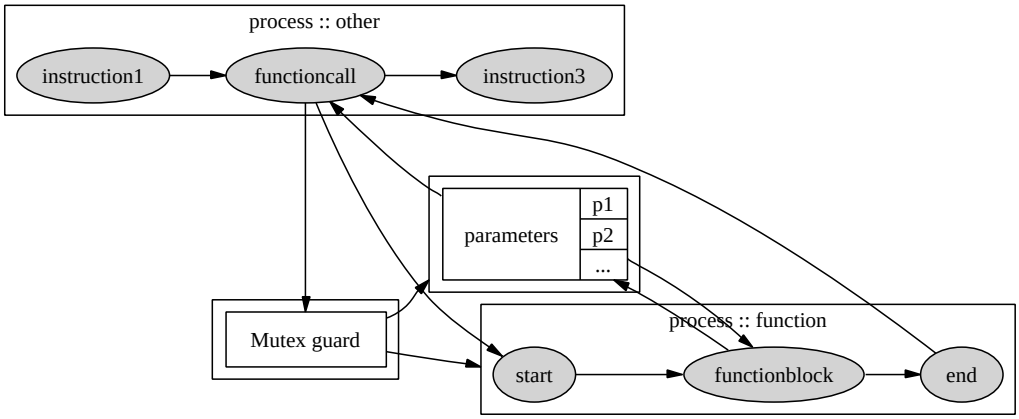

Fig. 5.20 *Shared function call using a Mutex scheduler to serialize concurrent function access by different processes.*

The control flow (part of the state diagram of each process) for a function call is shown in Figure *5.20*.

5.10 Software Architecture

In addition to the derivation of the proposed hardware architecture model there is the possibility to derive a software model with equal operational and functional behaviour.

Each *ConPro* process is implemented on software level with a light-weighted process (thread), finally resulting in a multi-threaded program with a shared memory model. Global objects are implemented with thread-safe Mutex-guarded global functions. *ConPro* functions can be implemented with thread-safe re-entrant functions without the necessity of a Mutex scheduler.

Different back-ends exists for the *C* or *OCaML* programming language. Most *ConPro* objects and process statements can be directly implemented in software, except hardware blocks, signals, and statements using signals.

epubli, ISBN 9783746752228 (2018)

5.11 Further Reading

1. C. A. R. Hoare, *Communicating Sequential Processes*. 2004.
2. R. Milner, *The space and motion of communicating agents*. Cambridge University Press, 2009, ISBN 9780521738330
3. R. Milner, *A Calculus of Communicating Systems*, Vol. 92, L. Springer, 1986.
4. R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999, ISBN 0521643201
5. D. Sangiorgi and D. Walker, *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001, ISBN 0521781779
6. G. Jones, *Programming in OCCAM*. 1985.