

ConPro Programming Language

ConPro Programming Language for the Design of Concurrent Data Processing Systems

CP

SYNOPSIS

Introduction to and overview of the Programming Language ConPro implementing the Concurrent Communicating Sequential Processes Model

SYNOPSIS	1
DESCRIPTION	1
DATA TYPES	2
VALUES	2
EXPRESSIONS AND ASSIGNMENTS	3
DATA OBJECTS	5
IPC DATA OBJECTS	6
ABSTRACT OBJECTS	7
ARRAYS	8
STRUCTURES	9
ENUMERATION	10
PROCESSES	10
FUNCTIONS	11
BLOCKS	12
BRANCHES	13
LOOPS	14
EXCEPTIONS	15
IPC OBJECTS	16
MODULES	19
VERSION	20

DESCRIPTION

The ConPro programming language provides generic imperative statements like branches and loops with additional true parallel programming features. The main execution object is a process. Processes support data processing with strict sequential instruction behaviour. Concurrency is provided by (creation of) multiple processes executing in parallel on control path level and by bounded instruction blocks inside a process on data path level. Access of global shared resources is implemented with atomic guarded actions (mutual exclusion).

The programming language maps the communicating sequential processes model with atomic guarded actions to multiple finite-state

machines with data and control path specifications on Register-Transfer level, providing input for hardware synthesis.

DATA TYPES

Predefined ordinal data types *DT* are summarized in Table 2.

Tab. 1. Data types *DT*

Statement	Type <i>DT</i>	Description
<code>int[N]</code>	INT	Signed integer with data with of N bits
<code>logic</code> <code>logic [N]</code>	LOGIC	Unsigned integer and logic vector (value set {0, 1, Z, H, L}) with data width of N bits.
<code>char</code>	CHAR	Character
<code>bool</code>	BOOL	Boolean type (value set {true, false}.
<code>value</code>	VALUE	Untyped value (integer, logic, char, bool) with data type and width assigned at compile time (only used in constants)

VALUES

Values used in expressions with data content of registers, variables, and signals are associated with different ordinal data types *DT* depending on the value format, summarized in Table 2.

Tab. 2. Values and data types *DT*

Value	Type	Description
-2, -1, 0, 1, 2, 3, 4, ...	INT	Signed integer (decimal format)

Value	Type	Description
0,1,2,3,4,...	INT, LOGIC	Unsigned integer (decimal format)
0x1,0x2,...	INT, LOGIC	Unsigned integer or logic (hexadecimal format: 0-9, A-F, a-f)
0b110,0b0101,...	INT, LOGIC	Unsigned integer or unsigned logic value (binary format: 0,1)
0l111,0lZZZ	LOGIC	Logic value (logic multi-value format; 0,1,L,H,Z)
'a','A',...	CHAR	Character
"abc"	STRING	String (character array)
true, false	BOOL	Boolean value
nanosec, microsec, millisec, sec	UNIT	Time unit (used with integer values)
hz, kilohz, megahz, gigahz	UNIT	Frequency unit (used with integer values)

EXPRESSIONS AND ASSIGNMENTS

Expressions are used in assignments, branches, function applications, and loops. There are arithmetic, relational, and boolean/logical operations.

Tab. 3. Arithmetic, relational, and boolean/logical (bitwise) operators with applicable data types

Operator	Type	Description
+, -, *, /	INT, LOGIC, CHAR	Addition, Subtraction (Negation), Multiplication, Division. (CHAR: ASCII code)

Operator	Type	Description
< <= > >= = <>	INT, LOGIC, CHAR	Lower than, lower equal than, greater than, greater equal than, equal, not equal.
and, or, xor, not	BOOL	Boolean operators
land, lor, lxor, lnot	INT, LOGIC, CHAR	Bitwise logical operators
@ a @ b @ c ..	LOGIC	Bitvector concatenation operator
~ V~B	INT	$\log_B(V)$ (only in constant definitions allowed)
lsl, lsr objlshx n	INT, LOGIC	Static and dynamic (non-constant shift parameter n) shift operations. Left hand operator must be an object (register, variable, signal).
to_logic to_int to_char to_bool	INT, BOOL, LOGIC, CHAR	Type conversion (only applicable to single objects)

All operands of an expression must be of the same type. Explicit type conversion can be used only to convert the data type of native objects (register, variable, signal).

Assignments of expression values to data objects are shown in Def. 1.

Def. 1. Assignment of a value (calculated from an expression) to a data object (register, variable, signal, queue, channel).

```
LHS <- RHS;
x <- expr;
```

Function application (only not recursive) is provided by the function name and an argument list, which can be empty (Def. 2). Arguments containing expressions are evaluated before function application. Function applications can be embedded in expressions.

Def. 2. Function application (f) and procedure execution (p) with arguments, w/o arguments

.....

```
dst <- f(arg1,arg2,...);
dst <- f();
    in expression f(...)
p(arg1,arg2,...);
p();
```



DATA OBJECTS

Sequential data processing requires storage objects (memory). There are registers and variables for data storage. A register is a single memory object mappable to CREW access behaviour, whereas a variable is bounded to a memory block (RAM) with EREW access behaviour. Additional there is a signal object without any storage required for interconnect of hardware components. Data objects are assigned to a specific data type.

There are data objects with local and global visibility (scope). Local objects can only be accessed by one process, whereas global objects can be accessed by multiple processes concurrently. Concurrent access of those global objects are resolved and serialized with a atomic guarded actions and a scheduler for each object.

Storage objects of structure type (concerning only registers and signals) are splitted in an independent set of storage objects with each object associated with a structure element.

Tab. 4. Data object definitions (TYPE: structure type)

.....

Statement	Data Type DT		Description
reg R: DT;	INT,	LOGIC,	Creates a register storage object R of data type DT and optional data width N (bits).
reg R: DT[N] ;	CHAR,	BOOL,	
reg R,S,...:DT;	TYPE		

Statement	Data Type DT	Description
<pre>block B; var V: DT in B; var V: DT[N] ...</pre>	INT, LOGIC, CHAR, BOOL, TYPE	Creates a variable storage object V of data type DT and optional data width N (bits). Variables are bound in a RAM block, which must be specified explicitly. RAM data cell width and number of cells are determined at compile time.
<pre>sig S: DT; sig S: DT[N]; sig S,T,...:DT;</pre>	INT, LOGIC, CHAR, BOOL, TYPE	Creates a inter-connect signal object S of data type DT and optional data width N (bits).
<pre>const C: DT := V; const C: DT[N] := V; const NC: value := V;</pre>	INT, LOGIC, CHAR, BOOL, VALUE	Creates a constant object C with value V of data type DT (or generic VALUE) and optional data width N (bits).

IPC DATA OBJECTS

There are predefined data objects implementing synchronized inter-process communication: queue and channels. Though queues and channels are abstract objects they can be used directly in expressions (RHS) and assignment statements (LHS). Queues have a buffer storage depth (size) which can be specified with the parameter `depth`. Channels have always a depth of one and can be buffered or unbuffered only implementing a handshake.

Queues or channels of structure type are splitted in a set of coupled queues or channels with each object associated with one structure element.

Tab. 5. IPC data object definitions (TYPE: structure type)

Statement	Data Type DT	Description
queue Q: DT; queue Q: DT[N] ; queue Q,R,...: DT with P=V and ..;	int, logic, char, bool, TYPE	Creates a queue object Q of data type DT with optional parameter settings.
channel Q: DT; channel Q: DT[N] ; channel Q,R,...: DT with P=V and ..;	int, logic, char, bool, TYPE	Creates a channel object Q of data type DT with option- al parameter settings.

CP

Tab. 6. IPC data object parameters

Parameter	Value	Description
depth	1..256 (∞)	Size of queue buffer. Chan- nel depth is always 1.

ABSTRACT OBJECTS

Abstract objects are modified only by a set of defined methods. There are builtin objects (queue, channel) and user abstract objects, with behavioural implementation and method interfaces defined by the EMI ADTO programming language.

Tab. 7. Abstract object definitions

Statement	Type	Description
object O: AT; object O: AT with P=V and P=V ...;	AT: mutex, semaphore, event, timer, ...	Creates an abstract object O of type AO with optional pa- rameter list (parameter P and value V).

ARRAYS

Arrays can be created with registers, variables, and signals with a specific array element data type. A register array has still CREW access behaviour for each cell, though dynamic element selectors require adressable array selectors (one for each accessing process).

Additionally arrays can be created with structure and abstract object types.

CP

Tab. 8. Array definitions (TYPE: structure type)

Statement	Data Type DT	Description
<pre>array A: reg[I] of DT; array A: reg[I,J,K] of DT;</pre>	<pre>int, logic, char, bool, TYPE</pre>	Creates a register storage array A with I elements of data type DT and optional data width N (bits). Multi-dimensional arrays can be created by extending the size parameter list I,J,K.
<pre>array A: var[I] of DT; block B; array A: var[I] of DT in B;</pre>	<pre>int, logic, char, bool, TYPE</pre>	Creates a variable storage array A with I elements of data type DT and optional data width N (bits) bounded to a RAM block. The assignment of a specific block is optional.
<pre>array A: sig[I] of DT;</pre>	<pre>int, logic, char, bool, TYPE</pre>	Creates a signal array A with I elements of data type DT and optional data width N (bits).
<pre>array A: object AT[I]; array A: object AT[I] with P=V and P=V ...;</pre>	<pre>AT: queue, channel, ...</pre>	Creates an abstract object array A with I elements of object type AT and optional parameter list (parameter P with value V).

Statement	Data Type DT	Description
<pre>A. [i] <- A. [j] ; A. [i, j, k] <- A. [x, y, z] ;</pre>	AT	Access of array elements on left-hand (write) and right-hand (read) side of an assignment and in expressions by using the dot bracket selector.

STRUCTURES

There are data and component structure types (records). Data structures are used to bind single data elements semantically coupled to a named type. Component structures define a hardware component interface (port signals). Data structure types can be applied to all data object definitions including queues and channels.

Tab. 9. Stucture type definitions and definition of objects of structure type.

Statement	Data Type DT	Description
<pre>type T: { e1 : DT; e2 : DT; .. };</pre>	int, logic, char, bool	Defines a data structure type T with elements e1, e2, .. of specified data types. Registers, variables, and signal objects of this type can be instantiated.
<pre>type T: { e1 : N1; e2 : N2; e3 : N3 to N4; e4 : N5 downto N6; .. };</pre>	logic	Defines a bit-field data structure type T with elements e1, e2, .. of specified data width (data type logic) N1, N2 ,.... Registers, variables, and signal objects of this type can be instantiated.
<pre>type Tc: { e1 : DIR DT; e2 : DIR DT; .. }; DIR = {input, output, inout}</pre>	int, logic, char, bool	Defines a component structure interface type Tc with elements e1, e2, .. of specified data types and data/signal flow directions.

CP

Statement	Data Type DT	Description
reg R: T; var V: T; ...	TYPE	Defines a data object (register R, variable V, signal S) of the user defined structure type T.
component C: Tc;	TYPEc	Instantiates a component object with an interface structure type Tc .
reg R: T; R.e1 <- .. X <- R.e2 ..	TYPE	Access of structure elements and bit fields by using the dot selector.

ENUMERATION

Tab. 10. Enumeration type definitions

Statement	Object Type	Description
type e : { S1; S2; .. }; type e : {.. } with code=C; C={one,bin,gray ,NAME}	REGISTER, SIG- NAL, VARIABLE	Definition of symbolic enumeration list defining named constants. The value of an enumeration element is calculated at compile time. The first enumeration element has index 1, the seconde 2, and so forth. The final coding style can be set with parameter code.

PROCESSES

A process is the main execution unit. Entire processes are operating independently and concurrently, but process statements are executed sequentially. A process has a local process space consisting of data and some abstract objects. Inter-process communication and synchronization is performed by using global objects with guarded atomic access. Concurrent access is serialized by a scheduler.

Tab. 11. Process definition and process control.

CP

Statement	Description
<pre>process P: begin definitions statements end; process P: ... end with P=V and ..;</pre>	Definition of a process with object definitions (optional) and a sequence of statements. Processes can be parameterized by applying a parameter list (parameter P and value V). Except the main process each process must be started explicitly.
<pre>array P: process [N] of begin definitions statements end; # ≡ process number</pre>	Definition of a process array with object definitions (optional) and a sequence of statements. Processes can be parameterized by applying a parameter list (parameter P and value V). N is the size of array (number of processes to be created).
<pre>P.call(); P.start(); P.stop();</pre>	Process control statements (process methods). Process calling is a synchronous operation. If a process P1 calls a process P2, the process P1 is blocked until process P2 has finished his work (by reaching the end state). Process starting and stopping does not block the executing process. It is an asynchronous operation.

FUNCTIONS

Functions are implemented by using processes with additional parameters (global registers) initialized with a value at function application time. The application of a function within an expression or the procedure execution passes (optional) arguments to the parameters and starts the process associated with the function. The calling process is blocked until the function process finishes (by reaching the end state). A return value is passed back to the calling process (in case of a function).

Tab. 12. Function definition and function application

CP

Statement	Description
<pre>function f(p1:DT,..) return (pn:DT): begin definitions statements end; .. end with P=V and .. ; .. end with inline;</pre>	<p>Definition of a function with (optional) formal parameters p_1, p_2, \dots and a return parameter p_n. There is no return statement. The value of the return parameter must be modified within the function body.</p> <p>Function and procedures can be parameterized. The inline parameter replaces each function application or procedure execution with the respective statement sequence. Local data objects are shared.</p>
<pre>function p(p1:DT,..): begin definitions statements end;</pre>	<p>Definition of a procedure with (optional) formal parameters p_1, p_2, \dots but without a return parameter. There is no return statement.</p>
<pre>P(v1,v2,...); P(); X <- f(v1,v2,...); X <- f();</pre>	<p>Function and procedure application with and w/o arguments.</p>

BLOCKS

Process or function statements can be grouped with a block statement. A block can be parameterized, for example with different scheduling, allocation, or optimization behaviour.

Tab. 13. Statement blocks

Statement	Description
<code>begin stmt1; stmt2; .. end;</code>	Sequetntial statement composition.
<code>begin stmt1; stmt2; .. end with P=V and ..;</code>	Sequetntial statement composition with additional behaviour or synthesis parameterization (paremeter P with value V).
<code>begin stmt1; stmt2; .. end with bind[=true]; ⇔ stmt1, stmt2,...;</code>	Parallel data statement composition. Equal to comma separated list of (data assignment only) statements. Only one data statement may perform a guarded access of a global object (read of global registers is not guarded).

CP

BRANCHES

There are different branch statements available. They pass the program flow to an alternative statement or a block of statements depending on values. Branches can appear on module top-level and within block statements (processes, functions...).

Tab. 14. Branch statements

CP

Statement	Kind	Description
<pre>if <i>expr</i> then <i>statement1</i>; if <i>expr</i> then <i>statement1</i> else <i>statement0</i>;</pre>	Boolean Branch	Depending on the result of the boolean expression <i>expr</i> a branch occurs either to <i>statment1</i> (<i>expr</i> =true) or optional to the <i>statement0</i> (<i>expr</i> =false). If there is more then one statement, a block is required.
<pre>match <i>expr</i> with begin when <i>v1</i>: <i>stmt1</i>; when <i>v1</i>,<i>v2</i>,...: <i>stmt123</i>; when <i>v1</i> to <i>v2</i>: <i>stmtv1tov2</i>; when <i>v1</i> downto <i>v2</i>: <i>stmtv2tov1</i>; others: <i>stmte</i>; end;</pre>	Multi-value Matching Branch	Different constant values are matched with the result of the expression <i>expr</i> and the respective statements are selected on successfull matching. The default others case (matching all other values) is optional. A list of values <i>v1</i> , <i>v2</i> ,... specifies different alternatives matching the same case.
<pre>exception <i>e1</i>,...; try <i>statement</i>; with begin when <i>e1</i>: <i>stmt1</i>; when <i>e1</i>,<i>e2</i>,...: <i>stmt123</i>; others: <i>stmte</i>; end;</pre>	Exception Handler Branch	Execptions raised in <i>statements</i> are matched and the respective statements are selected on successfull matching. The default handler <i>else</i> (matching all other exceptions) is optional.

LOOPS

There are different loop statements available. Each loop repeats the execution of the loop body as long as a boolean condition is satisfied. A counting loop iterates a list of values, specified by a range.

Tab. 15. Loop statements

Statement	Kind	Description
<pre> for i = a to downto b do begin <i>statements</i> end; end with unroll; end with P=V ..; </pre>	Counting Loop	<p>The for-loop executes the loop body <i>statements</i> for each element in the iterator list, a range of values including boundaries. The loop iterator variable <i>i</i> holds the current iteration value.</p> <p>Loops can be parameterized. The unroll parameters replicates the loop body $(b-a)+1$ times and replaces the loop iterator with the current iteration value.</p>
<pre> while <i>expr</i> do begin <i>statements</i> end; </pre>	Conditional Loop	<p>The while-loop executes the loop body as long as the boolean expression <i>expr</i> is true. The test of the boolean expression is performed before each loop iteration.</p>
<pre> always do begin <i>statements</i> end; </pre>	Unconditional Loop	<p>This loop never terminates (except by raising an exception).</p>
<pre> wait for <i>cycles</i>; wait for <i>time</i>; wait for <i>cond</i>; wait for <i>cond</i> with <i>statement</i>; wait for <i>cond</i> with <i>stmt1</i>; else <i>stmt0</i>; </pre>	Delay/Blocking Loop	<p>The wait for statement blocks the execution until a time interval is passed or a condition is true. Optionally a signal assignment can be applied (<i>stmt1</i>) as long as the condition is false. An optional default statement (<i>stmt0</i>) is applied at the remaining time.</p>

CP

EXCEPTIONS

Exceptions are used to leave a control environment, for example a function, loop, or branch. Exceptions are propagated beyond control environments untill an exception handler catches the exception. Otherwise an uncaught exception fault appears.

An exception raised within a nested control environment (nested branches/loops or function calls) is passed to the next higher environment level untill a handler environment is reached. Exception handler environments can be nested, too. Exception not caught by a particular handler (without default-branch) are re-raised.

CP

Tab. 16. Exception handler statements

Statement	Kind	Description
<code>exception ex1, ...;</code>	Type	Definition of a named exception signal.
<code>try statement; with begin when e1: stmt1; when e1,e2,...: stmt123; others: stmte; end;</code>	Exception Handler Branch	Exceptions raised in <i>statements</i> are matched and the respective statements are selected on successfull matching. The default handler <code>else</code> (matching all other exceptions) is optional.
<code>raise ex;</code>	Raising	Raises an exception signal.

IPC OBJECTS

Inter-process communication takes place by using global objects. Though global data storage objects are guarded by a mutex scheduler, they are not suitable for process synchronization. There are abstract IPC objects available providing synchronization of different kind, summarized in Table 17.

Tab. 17. Synchronization objects and their methods

Object	Methods	Description
queue open Core; queue q: DT ?with depth=N and scheduler ="fifo";	<i>q.write ~</i> q <- expr; <i>q.read ~</i> x <- q; q.unlock()	A queue is a buffer holding up to N elements with synchronized access in FIFO order. The read operation blocks until at least one data element is available. The write operation blocks if the queue is full. The unlock operation unblocks all blocked processes. The data type can either be a core type or record structure type.
channel open Core; channel c: DT ?with model=M; M={buffered, unbuffered}	<i>c.write ~</i> c <- expr; <i>c.read ~</i> x <- c; c.unlock()	A channel is a buffer holding one (buffered) or no element with synchronized access. The read operation blocks until at least one data element is available or one write operation is pending. The write operation is blocked until the buffer is empty or a read operation occurs. The unlock operation unblocks all blocked processes.

CP

Object	Methods	Description
mutex open Mutex; object o: mutex ?with schedule=S; S={fifo}	o.lock() o.unlock() o.init()	A mutex implements mutual exclusion access to shared resources. A mutex is either locked or unlocked. Only one process can own the lock by using the lock operation. The unlock operation releases the lock. Locking an already locked mutex blocks the process. Initialization is required by using the init operation.
semaphore open Semaphore; object o: semaphore ?with scheduler=S and depth=N and init=V; S={"fifo"}	o.down() o.up() o.unlock() o.init(V)	A semaphore implements an asynchronous counter with a value range [0..depth-1] used in producer-consumer applications. The counter value never becomes negative. The down operation decrements the counter. If the counter is already zero, the process is blocked. The up operation increments the counter. Initialization is required by using the init operation.

Object	Methods	Description
event open Event; object o: event ?with latch=B;	<code>o.await()</code> <code>o.wakeup()</code> <code>o.init()</code>	An event implements simple signal-based process synchronization. Multiple processes can wait for an event by using the <code>await</code> operation. Another process can wakeup those blocked processes at the same time by using the <code>wakeup</code> operation. A latched event prevent race conditions if the waiting request arrives after the wakeup operation. Initialization is required by using the <code>init</code> operation.
barrier open barrier; object o: barrier;	<code>o.await()</code> <code>o.init()</code>	A barrier implements simple signal-based process synchronization of group of N processes. A group of processes can wait for a barrier event (enter the barrier) by using the <code>await</code> operation. If the N-th process enters the barrier all processes are unblocked immediately at the same time. The group size N is determined at compile time. Initialization is required by using the <code>init</code> operation.

Object	Methods	Description
timer open Timer; object o: timer ?with mode=M; M={0,1}	o.await() o.time(T) o.start() o.stop() o.init() o.sig_action(S,L1,L0)	A timer is a self synchronizing event object. Processes can wait for the timer event by using the await operation. After the time interval has elapsed, which must be set by the time operation, the waiting processes are woken up. The timer must be started and can be stopped by the start and stop operations. In mode=0 the timer operates continuously, in mode=1 only one time. The sig_action operation attaches a signal S to the timer returning the actual state (L0: inactive, L1: active).

SYSTEM CONFIGURATION

The System module and an instantiated system object can be used to specify overall system design constraints, i.e., clock frequency or reset level, and target device(s).

```
open System;
object sys: system;
sys.clock  (18500 kilohz);
sys.target ("xc3s1000-ft256-5");
sys.target ("xcf04s");
sys.target ("xc18v04");
sys.reset_level (0);
sys.reset_internal (1);
```

MODULES

Commonly there is one main module defined by the main entry source file. A module consists explicitly of a module behavioural description including processes, object definitions, and top-level state-

ments and implicitly of a module interface defining the component port interface, at least the clock and reset port signals. Additional port components can be added by exporting objects (register, signals) and hardware component interfaces. Additional compound modules can be defined on structural component level. Each compound module consists of an implementation definition (a main module which must be imported), behavioural components instantiated from this main module, and an inter-connect component connecting all instantiated module components.

Tab. 18. User defined modules

Statement	Description
<i>definitions</i> <i>declarations</i> <i>processes</i> <i>top-level stmts</i>	Declares and creates a top-level behavioural module <i>M</i> (from source file <i>m.cp</i>).
module <i>MC</i> begin <i>import</i> <i>component</i> <i>connect</i> <i>mapping</i> end;	Declares and creates a new compound module <i>MC</i> from a behavioural module with import, component instantiation, inter-connect and mapping parts.
import <i>M</i> ; component <i>C1, C2, ...</i> <i>M</i> ;	Import of a behavioural module <i>M</i> (top-level main module) and module component instantiations (replicated module components).
type <i>ic</i> : { port <i>S</i> : <i>dir typ</i> ; ... }; component <i>IC</i> : <i>ic</i> := { <i>C1.S1, ...</i> }; <i>IC.S1</i> << <i>IC.S2</i> ;	Definition of an inter-connect component type and instantiation of an inter-connect component with default signal mapping of module component port signals. Finally inter-connect component signals can be connected with additional mapping statements.

VERSION

Last modified: November 14, 2013

CP