ConPro Programming Language

ConPro Programming Language for the Design of Concurrent Data Processing Systems

SYNOPSYS

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

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 DT	Description
`int[N]`	`INT`	Signed integer with data with of N bits
`logic` `logic[N]`	`LOGIC`	Unsigned integer and logic vector (value set {`0, 1, Z, H, L`}) with data width of N bits.
`char`	`CHAR`	Character
`bool`	`BOOL`	Boolean type (value set {`true, false`}.
`value`	`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, greate 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` *obj* `lsx` *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();
expr(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;` `reg R: DT[N];` `reg R,S,...:DT;`	`INT, LOGIC,` `CHAR, BOOL,` `TYPE`	Creates a register storage object R of data type DT and optional data width N (bits).

Statement	Data Type DT	Description
`block B;` `var V: DT in B;` `var V: DT[N] ...`	`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.
`sig S: DT;` `sig S: DT[N];` `sig S,T,...:DT;`	`INT, LOGIC,` `CHAR, BOOL,` `TYPE`	Creates a inter-connect signal object S of data type DT and optional data width N (bits).
`const C:` `  DT := V;` `const C:` `  DT[N] := V;` `const NC:` `  value := V;`	`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 interprocess 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 optional parameter settings.

Tab. 6. IPC data object parameters

Parameter	Value	Description
`depth`	`1..256 (∞)`	Size of queue buffer. Channel 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 parameter 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.

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

Statement	Data Type DT	Description
`array A: reg[I] of DT;` `array A: reg[I,J,K] of DT;`	`int, logic, char, bool, TYPE`	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.
`array A: var[I] of DT;` `block B;` `array A: var[I] of DT in B;`	`int, logic, char, bool, TYPE`	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.
`array A: sig[I] of DT;`	`int, logic, char, bool, TYPE`	Creates a signal array A with I elements of data type DT and optional data width N (bits).
`array A: object AT[I];` `array A: object AT[I] with P=V and P=V ...;`	`AT: queue, channel, ...`	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
`A.[i] <- A.[j];` `A.[i,j,k] <-` `  A.[x,y,z];`	`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
`type T: {` `  e1 : DT;` `  e2 : DT;` `  .. };`	`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.
`type T: {` `  e1 : N1;` `  e2 : N2;` `  e3 : N3 to N4;` `  e4 : N5 downto N6;` `  .. };`	`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.
`type Tc: {` `  e1 : DIR DT;` `  e2 : DIR DT;` `  .. };` `DIR = {input, output, inout}`	`int, logic, char, bool`	Defines a component structure interface type `Tc` with elements `e1`, `e2`, .. of specified data types and data/signal flow directions.

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, SIGNAL, 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.

Statement	Description
`process P:` `begin` `  definitions` `  statements` `end;` `process P: ...` `end with P=V and ..;`	Definition of a process with object definitions (optional) and a sequence of statements. Processes can be parameterized by appling a parameter list (paramter P and value V). Except the main process each process must be started explicitly.
`array P: process[N] of` `begin` `  definitions` `  statements` `end;` `# ≡ process number`	Definition of a process array with object definitions (optional) and a sequence of statements. Processes can be parameterized by appling a parameter list (paramter P and value V). N is the size of array (number of processes to be created).
`P.call();` `P.start();` `P.stop();`	Process control statements (process methods). Process calling is a synchronous operation. If a process P1 calls a process P2, the process P1 is blocked untill 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) intialized 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 untill 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

Statement	Description
`function f(p1:DT,..)` `  return (pn:DT):` `begin` `  definitions` `  statements` `end;` `..` `end with P=V and .. ;` `..` `end with inline;`	Definition of a function with (optional) formal parameters `p1`, `p2`,.. and a return parameter `pn`. There is no return statement. The value of teh return parameter must be modfied within the function body. 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.
`function p(p1:DT,..):` `begin` `  definitions` `  statements` `end;`	Definition of a procedure with (optional) formal parameters `p1`, `p2`,.. but without a return parameter. There is no return statement.
`P(v1,v2,...);` `P();` `X <- f(v1,v2,..);` `X <- f();`	Function and procedure application with and w/o arguments.

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
`begin` `  stmt1;` `  stmt2;` `  ..` `end;`	Sequetntial statement composition.
`begin` `  stmt1;` `  stmt2;` `  ..` `end with P=V and ..;`	Sequetntial statement composition with additional behaviour or synthesis parameterization (paremeter P with value V).
`begin` `  stmt1;` `  stmt2;` `  ..` `end with bind[=true];` ⇔ `  stmt1,` `  stmt2,...;`	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).

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

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

Tab. 15. Loop statements

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

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. Excption handler envionments can be nested, too. Exception not caught by a particular handler (without default-branch) are re-raised.

Tab. 16. Exception handler statements

Statement	Kind	Description
`exception ex1,..;`	Type	Defintion of a named exception signal.
`try` `  statement;` `with` `begin` `  when e1: stmt1;` `  when e1,e2,..:` `  stmt123;` `  others: stmte;` `end;`	Exception Handler Branch	Execptions raised in *statements* are matched and the respective statements are selected on successfull matching. The default handler `else` (matching all other exceptions) is optional.
`raise ex;`	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 nou suitbale for process synchronization. There 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";`	*q.write*: `q <- expr;` *q.read*: `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}`	*c.write*: `c <- expr;` *c.read*: `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.

Object	Methods	Description
mutex `open Mutex;` `object o: mutex` `?with` `schedule=S;` `S={fifo}`	`o.lock()` `o.unlock()` `o.init()`	A mutex implementes mutual exclusion access to shared resources. A mutex is eithe rlcoked 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 proecss. 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 semaphpore implements asynchronized counter with a value range [0..depth-1] used in producer-consumer applications. The counter value never becoms negative. The `down` opration 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;`	`o.await()` `o.wakeup()` `o.init()`	An event implements simple signal-based process synchronization. Multiple processes can wait for an event by using the `await` operation. Another process can wakeup those blocked processes at the same time by using the `wakeup` operation. A latched event prevent race conditions if the waiting request arrives after the wakeup operation. Initialization is required by using the `init` operation.
barrier `open barrier;` `object o:` `  barrier;`	`o.await()` `o.init()`	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 `await` 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 `init` operation.

CP

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 continously, 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).

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 statements 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 conists 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
definitions *declarations* *processes* *top-level stmts*	Declares and creates a top-level behavioural module *M* (from source file *m*.cp).
module *MC* begin *import* *component* *connect* *mapping* end;	Declares and creates a new compound module *MC* from a behavioural module with import, component instantiaition, inter-connect and mapping parts.
import *M*; **component** *C1*, *C2*,..: *M*;	Import of a behavioural module *M* (top-level main module) and module component instantiations (replicated module components).
type *ic*: { port *S*: *dir typ*; ... }; **component** *IC* : *ic* := { C1.S1, ... }; IC.S1 << IC.S2;	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: October 29, 2013