Synthesis of Parallel and System-on-Chip Designs With Behavioural High-Level Hardware-Synthesis Using Communicating Sequential Processes and the ConPro-Framework

Dr. Stefan Bosse

Technical Paper

4.3.2010

Abstract

Traditionally, there are two different ways to model and implement System-On-Chip-Designs (SoC): using a structural and/or a behavioural level. The structural level decomposes a SoC into independent submodules interacting with each other using centralized or distributed networks and communication protocols. The behavioural level usually describes the behaviour of the full design interacting with the environment. Complex reactive systems with dominant and complex control paths play an increasing role in SoC-design. The major contribution to concurrency appears on control path level. This article gives an introduction to SoC-design methodology using the behavioural hardware compiler ConPro providing a programming model based on concurrent communication sequential processes (CSP) with an extensive set of interprocess-communication primitives. An extended case study of a communication protocol used in high density sensor-actuator-networks should demonstrate the design of a SoC for a robot actuator. The communication protocol is suited for high-density intra- and interchip networks.

Key Words and Phrases: Circuit Design, Digital Logic, SoC, NoC, Register-Transfer-Logic, Communicating Sequential Processes, Higher-Level-Synthesis, Multiprocessing, Parallel Programming, FPGA, ASIC

Table of Content

1 Introduction and State of the Art

Today there is an increasing requirement for the development of System-On-Chip-Designs (SoC) using Application-Specific Digital Circuits, with increasing complexity, too, serving low-power and miniaturization demands. The structural decomposition of such a SoC into independent submodules requires smart networks and communication (Network-on-Chip, NoC) serving chip area and power limitations. Traditionally, SoCs are composed of microprocessor cores, memory and peripherial components.

But in generally massiv parallel systems require modelling of concurrency both on control- and data-path level. Digital logic systems are preferred for exploration and implementation of concurrency.

Traditionally, digital circuits are modelled on hardware behaviour or gate level, but usually the entry point for a reactive or functional system is the algorithmic level. The Register-Transfer-Logic (RTL) on architecture and hardware level must be derived from the algorithmic level, requiring a raise of abstraction of RTL **ZHU01**.

With increasing complexity, higher abstraction levels are required, moving from hardware to algorithmic level. Naturally imperative programming languages are used to implement algorithms on program-controlled machines which process a sequential stream of data- and control operations. Using this data-processing architecture, a higher-level imperative language can be simply mapped to a lower-level imperative machine language, which is a rule-based mapping, automatically performed by a software compiler.

But in circuit design, there is neither an existing architecture nor an existing low level language that can be synthesized directly from a higher level one.

An imperative programming approach provides both abstraction from hardware and direct implementation of algorithms, but usually reflects the memory-mapped von-Neumann computer architecture model.

Another important requirement of a programming language in circuit design (in contrast to software design) is the ability to have fine-grained control over the synthesis process, usually transparent.

Using generic memory-mapped languages like C makes RTL hardware synthesis difficult because of transparency of object references (using pointers) preventing RTL mapping. Additionally, concurrency models are missing in most software languages. There are many attempts to use C-like languages, but either with restrictions, prohibiting anonymous memory access with pointers, or using a program-controlled (multi-) processor architecture with classical hardware-software-co-design, actually dominant in SoC-Design. But SoC-designs using generic or application-specific processor architectures complicate low-power designs and concurrency is coarse grained.

One example is PICO **KAT02**, addressing the complete hardware design flow targeting SoC and customizable or configurable processors, enhanced with custom-designed hardware blocks (accelerators). The RTL level is modelled with C. The program-controlled approach with processor blocks enables software compilation and unrestricted C (functions, pointers) but lacks support of true bit-scaled data objects.

Another example is SPARK **SPA04**, a C-to-VHDL high-level framework, currently with the restrictions of no pointers, no function recursion, and no irregular control-flow jumps. It is embedded in a traditional hardware-software-co-design flow. It is based on speculative code motions and loop transformations used for exploration of concurrency. SPARK generates pure RTL. Only a single-threaded control-flow is provided.

Though SystemC provides many features suitable for higher-level synthesis, it is primarily used for simulation and verification, and only a subset can be synthesized to circuits. True bit-scaled data types are supported. Concurrency can be modelled using threaded processes, for example used in Forthes commercial synthesis tool Cynthesizer **HLS08**. Interprocess communication is modelled on transaction level (TLM). SystemC provides a high-level-approach to model hardware behaviour and structure, rather than algorithms.

Non of these approaches fully satisfy the requirements for pure RTL circuit design while using C-based languages, especially providing a consistent hardware, software, and concurrency model.

Efficient hardware design requires more knowledge about objects than classical languages like C can provide, for example true bit-scaled registers, access, and implementation models on architecture level (for example singleport versa dualport RAM blocks, static versa dynamic access synchronization). The generic software approach only covers the implementation of algorithms, but in hardware design the synthesized circuit must be connected to and react with the outside world (other circuits, communication links and many more), thus there must be a programming model to interface to hardware blocks, consistent with the imperative programming model. Furthermore, there must be a way to easily implement synchronization always required in presence of concurrency (at least on control path level). A multi-process model, established in the software programmer community, provides a common approach for modelling parallelism, which is the preferred approach to implement and partition reactive systems on algorithmic level.

This article focuses 1. on the design-methodology of SoCs using a concurrent multi-process model and the behavioural programming language ConPro **CON08**, 2. the synthesis methods and architecture models for compiling mainly reactive systems using this imperative programming language towards RTL level (modelled on hardware behaviour level VHDL) **CON09**.

Concurrency is modelled explicitly but can be exploited implicitly, too.

The following section *2* describes the used concurrency process model and interprocess communication, and section *3* explains basics of the ConPro programming language. The synthesized RTL architecture with relation to the programming model is described in section *5*, and finally section *6* gives an overview of the synthesis process and the synthesis rules.

An extended design study of a protocol implementation suited for sensor- and actuator networks is presented in section *7* and demonstrates the power and suitability of the synthesis approach and tool for complex circuit designs.

2 Modelling and Implementing Concurrency

Many algorithms used in applications like communication protocolls, sensors- and actuators are in general massiv parallel control systems **LEE06**, with many different tasks to be performed. Therefore, modelling concurrency is an essential part of a complex SoC-design. Concurrency can be either modelled explicitly (not transparent) or implicitly (transparent) by the synthesis tool:

Explicit Parallelism

The programming model explicitly describes parallelism. *Usually this is the preferred method for exploration of coarse-grained parallelism, which requires partitioning on algorithmic level, well done by the programmer, rather by the synthesis tool. No further computational effort must be made by the synthesis tool.*

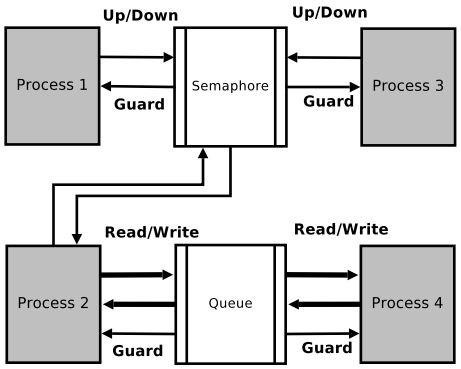
Implicit Parallelism

The compiler tries to explore and derive parallelism from an initially sequential program specification, described with an imperative language, or using functional languages with (hidden) inherent concurrency **SHA98**. Mostly, concurrency is derived from loops using unroll techniques with allocation of resources in parallel, but concurrency can be explored in basicblocks of data-independent expressions, too. For example, both expressions $x \leftarrow x + 1$ and $y \leftarrow y + 1$ can be scheduled (using RTL only) in one time step requiring two adders. *Usually this is the preferred method for exploration of fine-grained parallelism on data path level. High computational effort must be made for balancing area and time constraints, usually done with an iterative approach* **KU92**.

There are several advantages of the explicit concurrency model versa the implicit model derived from an initially pure sequential code, found in most extended C-like approaches **HLS08**, especially in the context of reactive systems. Knowledge-based modelling of concurrency can lead to a higher degree of concurrency.

A multi-process model with communicating sequential processes provides a concise way, 1. to directly map imperative programming languages to RTL, and 2. to provide parallelism on control path level, required for massive parallel control systems. The multi-process model requires explicit synchronization, shown in figure **1**. *Interaction between processes, mainly access of shared resources, is request-acknowledge based.*

FIGURE 1: **The multi-process model with request-based synchronization (IPC).**

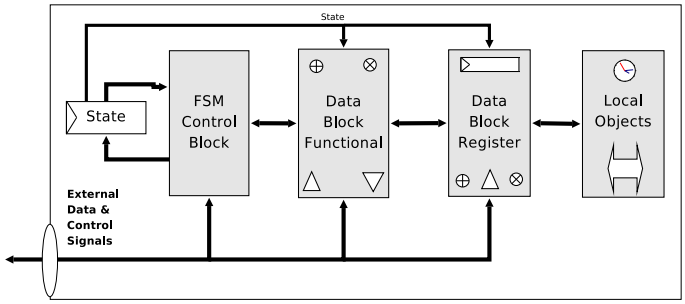


Concurrency reduces latency of an algorithm, and minimized latency is a precondition for the design and implementation of real-time systems.

Process Model and RTL-Architecture

A process ϕ provides an execution environment consisting of a control path Γ implemented with a Finite-State-Machine (FSM) and a data path Δ performing calculations, shown in figure **2**.

FIGURE 2: **The process implementation on hardware architecture level.**



A process ϕ bounds a sequence of instructions κ={κ_1,κ_2,...} to this execution environment. Process instructions on programming level must be executed in the order they appear (imperative nature). Therefore, the set of program instructions κ can be directly mapped to a set of states Σ of the FSM, implemented entirely in RTL.

An algorithm can be partitioned on control path level using a set of N processes Φ={ϕ_1,ϕ_2,...,ϕ_N}, initially executing independently and concurrently, doing communication, based on the model of communicating sequential processes (CSP) proposed in **HOA85**. A set of interprocess-communication (IPC) $\Im$ is required for synchronization. IPC creates control relations between processes: $\Im_i$:$\phi_n \leftrightarrow \phi_m$. *Using ConPro, it is possible to map multi-processing*

*and interprocess communication to RTL directly with low resource requirements, shown and proved in this article. The **ConPro** language* **CON09** *explained in this article provides concurrency both on control path level using processes and on data path level using bounded basicblocks, either specified on programming level or derived automatically by a basicblock scheduler.* Synthesis of RTL from an imperative programming language providing the multi-process model can be superior compared with traditional hardware-software-co-design using multi-processor architectures because abstract objects, especially all kind of interprocess-communication, can be implemented more efficiently in hardware than in software, both concerning resources and latency.
The set$\Phi$of processes belongs to a module. On module level, a set of global shared objects$\alpha=\Re\cup\Im$ can be defined, and on process level, local objects can be defined. Processes can access both their local and the global objects. These objects$\alpha$are either used for data storage ($\Re$={registers, variables in RAM blocks}), or for IPC ($\Im$={mutex, semaphore, queue, timer,...}).
The ConPro synthesis tool maps programming level processes and the instruction sequence$\kappa$to hardware components (entities in VHDL terminology), each consisting of:

1. a FSM (state register and state transition network) whose states representing the initial program flow of this particular process,
2. combinational data path of RTL (data path multiplexer, demultiplexer, functional units), and
3. transitional data path of RTL (data path multiplexer, demultiplexer, functional units, and local registers), shown in figure **2**.

Processes can be controlled by other processes. A process is treated like an abstract data type object (ADTO). Process control is established with the appropiate methods. Starting and stopping of processes are non-blocking operations, thereby calling a process which suspends the caller process untill the called (started) process reaches its end state.

Interprocess-Communication

Concurrency on control path level requires **synchronization AND00**. In the context of a SoC-design consisting of a multi-processor architecture, (coarse-grained) synchronization between different submodules is performed using a distributed or centralized network infrastructure and message passing. In the context of a behavioural multi-process-model (CSP), synchronization is handled by interprocess-communication using abstract objects with an appropiate set of methods applied to, for example, queues (read,write) or semaphores (up,down).
These IPC objects are **content-based**, in contrast to network communication for example in program-controlled multi-processor architectures. Therefore,

these IPC objects are optimally matched to the communication purpose producing lowest communication overhead, and can be implemented directly with hardware blocks. Message passing is reduced to hardware signals.
Realtime-systems require **temporal synchronization**, provided by the IPC objects with time skew and latency lower than one clock cycle, not achievable by generic network architectures and software implementations.
At least the access of shared resources must be protected using mutual exclusion locks (mutex). Access of all global objects is implicitly protected and serialized by a mutex scheduler. IPC and external communication objects are abstract object types, they can only be modified and accessed with a defined set of methods υ={υ_1,υ_2,...}, shown in table **1**. Queues and channels can be used in expressions and assignments like any other data storage object.

IPC Object $\Im$	Description	Methods υ
`mutex`	Mutual Exclusion Lock	`lock, unlock`
`semaphore`	Counting Semaphore	`init, up,down`
`barrier`	Counting Barrier	`init, await`
`event`	Signal Event	`init, await, wakeup`
`timer`	Periodic Timer Event	`init, set,start, stop, await`
`queue (*)`	FIFO queue	`read, write`
`channel (*)`	Handshaken Channel	`read, write`
`link`	External Handshaken Channel	`read, write`

TABLE 1: **Available IPC objects. Queues and channels belong both to the core and abstract object class, and can be used within expressions and assignments (*).**

The link IPC object is used for communication between different clock domains, either within a SoC (Interprocess), or between different components (Intraprocess). The link object provides a fast parallel communication channel for Global-Asynchronous-Local-Synchronous (GALS) system design. The link uses dual-rail encoding of data lines and a request-acknowledge handshake, derived from asynchronous circuit design using Muller-C-Gates **BAI01**. Only full-custom ASIC design supports synthesis of asynchronous circuit parts, like C-gates. To provide synthesis of such link components for all technologies (FPGA, standard-cell and custom-designed ASICs) using only synchronous circuits, the Muller-C-Gates are implemented with synchronous FSMs.
Concurrent access of shared resources (including IPC itself) is serialized by a scheduler, which prevents realtime capability on clock resolution, because

the upper limit of access time is unknown and depends on the program and algorithm. But realtime is possible using mutual exclusion which (spatial and temporal) protects parts of a program.

3 Behavioural Programming Language ConPro

The ConPro programming language consists of two classes of statements: 1. process instructions mapped to FSM/RTL, and 2. type, and object definitions. It is an imperative programming language with strong type checking. Beneath algorithmic statements, the programming language provides some kind of relation to the hardware circuit synthesized from the programming level. Additionally, there is a requirement to get full programmability of the design activities themselves, i. e. of the synthesis process, too **RU87**, implemented here with constrained rules on block level, providing fine-grained control of the synthesis process. The synthesis process can be parameterized by the programmer globally or locally on instruction block level, for example, scheduling and allocation.
The set of objects is splitt into two classes: 1. data storage type set $\mathfrak{R}$, and 2. abstract data type object set (ADTO)Θ, with a subset of the IPC objects $\mathfrak{I}$, providing object-orientated programming features with method access. Though it is a traditional imperative programming language, it features true parallel programming both in control and data path, explicitly modelled by the programmer.
Processes provide parallelism on control path level, whereby arbitrary nested bounded blocks inside processes provide parallelism on data path level.
There are two extended interfaces connecting the behavioural programming model to external hardware objects: 1. using hardware signals and component structures, or 2. using the External Module Interface (EMI). EMI provides two interface levels: A. the high-level ADTO level with method-based object access, and B. on low-level a behavioural hardware description using a script language extended subset of VHDL. For example. all IPC objects are modelled this way. User objects can be added, too.
There are different hierarchy levels provided by the programming model (shown in graph **1**):

1. The structural module (Module-S) level provides composition of behavioural modules (circuits) to a system-on-chip (SoC) design.

2. The behavioural module (Module-B) level contains processes and global objects. A toplevel port defines the interface of the circuit to the outside world.

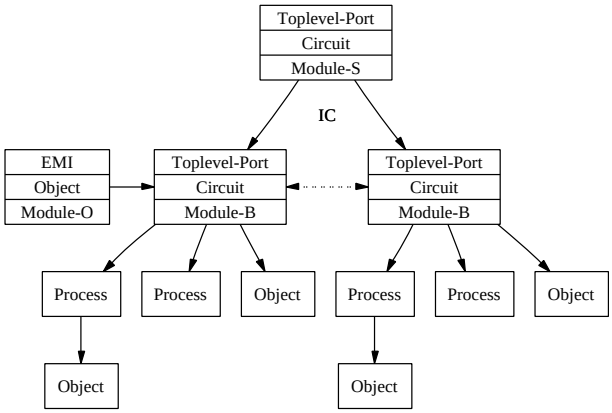
3. The process level contains a finite-state-machine, computational units, and local objects.

4. The abstract object Module-O, belonging both to process and behavioural module level. It defines and implements abstract data types objects and the provided method set υ.

A module represents a circuit component with an associated toplevel interface hardware port. At least the system clock and reset signals are connected. Some storage objects can be exported with interconnect signals

appearing in the toplevel port. Some abstract objects, for example communication links, have internal input-output signals routed to the toplevel port. A system-on-chip (SoC) can be composed of behavioural module component instantiations using a structural module environment. Either all toplevel port signals of each subcomponent or only a subset is routed to the SoC-toplevel-port. In the latter case, there is an interconnection component internally connecting module signals (which can be automatically generated using map instructions).

GRAPH 1: **Different design hierarchy levels and object visibilities.**



A **process environment** consists of a unique process name, local object definitions and process instructions. Single processes or an array of processes can be defined using the process environment. Each process executes the associated instruction sequence independently.
Objects can be defined within processes with local visibility, or globally on module level. The set of object types α contains storage $\mathfrak{R}$, signals $\wp$, and abstract objects Θ={$\mathfrak{I}$,D,E}: α={$\mathfrak{R}$,Θ}. The set D contains data computational objects, for example, random generators and DSP units, and the set E contains external communication objects.
Data storage can be implemented with single **registers** or with **variables** sharing one or more memory blocks. Choosing one of these object types is a contraint for synthesis, not a suggestion (in contrast to software programming). Registers provide concurrent-read-exclusive-write (CREW) access behaviour, whereby variables provide only exclusive-read-exclusive-write access behaviour (EREW). Both data storage types can be defined locally on process level or globally on module level. Both registers and variables are **true bit-scaled**, that means, any width ranging from 1 to 64 bit can be used. In the case of variable storage, the data width of the associated memory block is scaled to the widest object stored in this block. Fragmented variable objects are supported.
A strong typed expression model is provided. There is a set of core data

types: β={`logic, int, bool, char`}. Product types, both structures and arrays, can be defined to provide user-defined types.
A structure type binds different named elements with defined data types β to a new data type T. The structure type must be defined before an object of this type can be defined: `type T: { E1:` β`1; E2:` β`2; ...}.`
The object type α (register, variable, or signal) is associated during object definition. For each structure element a separate storage element is created. Array definitions consist of object and cell data type specifications in the form: `array A:` α`[N] of` β. Arrays can be accessed dynamically selected. In the case of register or object arrays, index-selected multiplexer and demultiplexer are created. Multi-dimensional storage arrays and arrays of abstract objects including processes are supported.
Example **1** shows some object definitions.

```
1:   ◆ Storage Objects ◆
2:   reg x,y: int[8]; Defines registers
3:   block ram1; Defines a block RAM
4:   var a,b,c: int[10] in ram1; Defines variables in RAM
5:   array mat1: reg[10] of int[23]; Defines an array
6:   array mat2: var[10] of int[8] in ram1;
7:   type complex: {
8:     real: int[16];
9:     imag: int[16];
10:  }; Defines a new data type
11:  reg zcmp: complex; Defines registers of this type
12:  process xyz:
13:  begin
14:    reg t: int[8]; Local data storage
15:    array ta: var[8] of int[8];
16:  end; Defines a new process
17:  ◆ Abstract Objects ◆
18:  open Mutex; Opens Mutex module required below
19:  object mu1: mutex; Defines ADTO
```

EXAMPLE 1: **Examples of different object definitions distinguished by their object and data type.**

```
1:   process p1:
2:   begin
3:     a←1, b←3, z←x-1; Bounded instruction block
4:   ⇔
5:     begin
6:       a←1;
7:       b←3;
8:       z←x-1;
9:     end with bind; Bounded instruction block, too
10:    x←(a+b)*4;
11:  end;
```

```
12:  process p2:
13:  begin
14:    a←1;
15:    b←3;
16:    z←x-1;
17:    x←(a+b)*4;
18:  end with schedule="basicblock";
```

EXAMPLE 2: **Example of assignments. Lines 3 and 5..9 (parameterized block) reflect equivalent syntax for concurrent statements with identical behaviour. Automatic basicblock scheduling is applied to the second process (parameterized process body block).**

Expressions contain data storage objects, constants, and operators. Supported are all common arithmetic, Boolean (logical), and relational operators. Most of them are directly mapped to hardware behaviour level (VHDL operators). Initially, assignments to data storage objects are scheduled in one time step, and the order of a sequence of assignments is preserved. A sequence of data-independent assignments can be bound to one time unit either explicitly by the programmer (bounded block), or implicitly evaluated by the basicblock scheduler (preserving data dependencies, but violating sequence order). A semicolon (without further scheduling constraints) schedules an assignment, whereby a colon-separated list binds assignments to one time unit, shown in example **3**, e.g. RTL scheduling originally proposed by Barbacci **BAR73**. There are different expression models which can be set on block level using the parameter: `expr={"flat","binary","shared"}`. The flat model maps operators of a (nested) expression 1:1 to hardware blocks (no shared resources), the binary mode splits nested expressions into single two-operand subexpressions using temporary registers, improving combinational path delay, and the shared model provides resource sharing of functional operators using ALUs.

Control Statements There are conditional branches, both Boolean and multivalue branching, conditional, unconditional and counting loops, conditional blocking wait-for statements, function calls, and exceptions. Exceptions are abstract (symbolic) signals which can be raised anywhere inside a process, and caught either inside the process where the signal is raised, or outside from any other process calling this respective process. Exceptions are propagated accross process boundaries. Exceptions are the only structural way to leave a control environment, there is no break, continue, or goto statement.

Functions User-defined functions can be implemented in two different ways: 1. as inlined not-shared function macros and 2. as shared function blocks. In the first case, the function call is replaced by all function instructions, and function parameters are replaced by their respective

arguments. In the second case, a function is modelled using the described process with an additional function control block containing a function call lock bound to an access scheduler and registers required for passing function arguments to parameters and returning results. Only call-by-value arguments of atomic objects can be used. The remaining functionality is provided by the underlying process model using the `call` method. Figure **4** shows the system architecture of a shared function block.
Functions are restricted to non-recursive calls due to a missing stack environment.

Example **3** shows a complete ConPro design. It is the implementation of the dining philosophers' problem using counting semaphores demonstrating resource sharing and scheduling. The story is: five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table is a large platter of spaghetti. Each philosopher needs two forks two eat. But there are only five forks for all. One fork is placed between each pair of philosophers, and they agree that each will use only the forks to the immeadiate left and right **AND00**, here implemented with a semaphore array `fork`, defined on line 17. The depth parameter specifies the datawidth of the semaphore counter. A register is defined in line 5, and an array of registers in line 7. Though each semaphore is an independent object (in hardware, too), the array can be accessed with dynamic selectors (register, variable), shown for example in lines 21-22 inside a for-loop.
The read ports of the shared registers `eating` and `thinking` are exported to the module toplevel port (SoC hardware port level). The design consists of seven processes. The philosophers are implemented with the process array `philosopher`.
A user-defined structure type is defined in lines 8-14, and finally a register of this type is created in line 15.
A mutex `num_eat_lock`, defined in line 6, is used to protect the incremental and decremental operation of the shared counter `num_eat`.
The event `ev` assures the same starting point for all philosopher processes (synchronization boundary).

```
1:   open Core, Process, Semaphore, System, Event, Mutex;
2:   object sys: system;
3:     sys.simu_cycles (500);
4:   object ev: event;
5:   reg num_eat: int[8];
6:   object num_eat_lock: mutex;
7:   array eating,thinking: reg[5] of logic;
8:   type stat : {
9:     phil1: int[8];
10:    phil2: int[8];
11:    phil3: int[8];
12:    phil4: int[8];
13:    phil5: int[8];
```

```
14:   };
15:   reg stats: stat;
16:
17:   array fork: object semaphore[5] with depth=8 and scheduler="fifo";
18:
19:   process init:
20:   begin
21:     for i = 0 to 4 do
22:       fork.[i].init (1);
23:     ev.init ();
24:   end;
25:
26:   function eat(n: natural):
27:   begin
28:     num_eat_lock.lock ();
29:       num_eat ← num_eat + 1;
30:     num_eat_lock.unlock ();
31:     match n with
32:     begin
33:       when 1: stats.phil1 ← stats.phil1 + 1;
34:       when 2: stats.phil2 ← stats.phil2 + 1;
35:       when 3: stats.phil3 ← stats.phil3 + 1;
36:       when 4: stats.phil4 ← stats.phil4 + 1;
37:       when 5: stats.phil5 ← stats.phil5 + 1;
38:     end;
39:     eating.[n] ← 1,
40:       thinking.[n] ← 0;
41:     wait for 5;
42:     eating.[n] ← 0,
43:       thinking.[n] ← 1;
44:     num_eat_lock.lock ();
45:       num_eat ← num_eat - 1;
46:     num_eat_lock.unlock ();
47:   end with inline;
48:
49:   array philosopher: process[5] of
50:   begin
51:     if # < 4 then
52:     begin
53:      ev.await ();
54:      always do
55:      begin
56:        - get left fork then right
57:        fork.[#].down ();
58:        fork.[#+1].down ();
59:        eat (#);
60:        fork.[#].up ();
61:        fork.[#+1].up ();
62:      end;
63:     end
64:     else
65:     begin
66:      always do
67:      begin
68:        - get right fork then left
69:        fork.[4].down ();
```

```
70:         fork.[0].down ();
71:         eat (#);
72:         fork.[4].up ();
73:         fork.[0].up ();
74:       end;
75:     end;
76:   end;
77:
78:   process main:
79:   begin
80:     init.call ();
81:     for i = 0 to 4 do
82:     begin
83:       philosopher.[i].start ();
84:     end;
85:     ev.wakeup ();
86:   end;
87:
88:   export eating,thinking,num_eat,stats;
```

EXAMPLE 3: **A complete ConPro example: the dininig philosopher problem. This implementation demonstrates resource sharing and synchronized access of shared resources using mutex and semaphore objects.**

Synthesis Gate-level synthesis with a standard cell technology using Leonardo Spectrum and SXLIB standard-cell library results in a circuit with 3919 gates, 235 D-flip-flops, and an estimated longest combinational path of 17 ns (55 MHz maximal clock frequency). The implementation of the semaphore array requires 1733 gates and 70 register, the event requires only 122 gates and 10 register. Each loop iteration of process philosopher up to the and from the eat function requires each 6 clock cycles to complete if all shared resources are immediately granted.

4 Abstract Objects and the External Module Interface

Abstract data type objects provide a method based interface to functional and operational hardware blocks, part of the SoC-design. They can be accessed concurrently, requiring an access scheduler serializing and guarding the access to the particular object. A process trying to access the object using a method is blocked (suspended) untill the resource is available and the request was serviced.
Beneath a set of standard objects required for interprocess-communication, the programmer/user can define its own objects and methods using the External Module Interface (EMI).
The EMI defines the object, the available methods and their programming interface, the access of the object and the implementation on hardware behaviour level, at least the access scheduler.
Example **4** shows the definition and implementation of the mutex object. The EMI is divided into different sections.
The EMI language is a combination of a script language (for example lists and list operations) and a subset of VHDL. There are parameter lists, for example the list of all processes `$P` accessing the object, or only processes using a particular method M (`$P.M` is a subset of `$P`). Other core parameters like the object name `$O` or the clock name `$CLK` get a value during synthesis.

Paramter This section defines local parameters evaluated during synthesis.

Methods The methods section defines the interface of the provided methods applied to the object.

Assert During synthesis sanity checks can be performed (optional).

Interface The interface section defines interconnect hardware signals required for the RTL implementation of each ConPro process accessing this object.

Mapping Signals rqeuired for access of the object (defined in interface section) must be mapped from local process to global module level (in terms of VHDL entity ports and port mappings)

Access This section defines for the object signal access during method call (RTL level).

Signals The signal section defines hardware signals required for the implementation of the object.

Process One or more hardware processes required for object implementation (functional/operational block, at least the scheduler) can be defined and modeled using the process section using a VHDL style language.

```
1:  #parameter
2:  begin
3:    $scheduler["static","fifo"] <= "static";
```

```
4:    end;
5:    -
6:    - Supported object methods
7:    -
8:    #methods
9:    begin
10:     init();
11:     lock();
12:     unlock();
13:   end;
14:   #assert
15:   begin
16:     size($P.init) >= 1;
17:     size($P.lock) >= 1;
18:     size($P.unlock) >= 1;
19:   end;
20:   -
21:   - Interface for processes accessing methods
22:   - of object (process port, local process context)
23:   -
24:   #interface
25:   begin
26:     foreach $p in $P.init do
27:     begin
28:       signal MUTEX_$O_INIT: out std_logic;
29:     end;
30:     foreach $p in $P.lock do
31:     begin
32:       signal MUTEX_$O_LOCK: out std_logic;
33:     end;
34:     foreach $p in $P.unlock do
35:     begin
36:       signal MUTEX_$O_UNLOCK: out std_logic;
37:     end;
38:     foreach $p in $P do
39:     begin
40:       signal MUTEX_$O_GD: in std_logic;
41:     end;
42:   end;
43:   -
44:   - Process mapping of object signals (global module context)
45:   -
46:   #mapping
47:   begin
48:     foreach $p in $P.init do
49:     begin
50:       MUTEX_$O_INIT => MUTEX_$O_$p_INIT;
51:     end;
52:     foreach $p in $P.lock do
53:     begin
54:       MUTEX_$O_LOCK => MUTEX_$O_$p_LOCK;
55:     end;
56:     foreach $p in $P.unlock do
57:     begin
58:       MUTEX_$O_UNLOCK => MUTEX_$O_$p_UNLOCK;
59:     end;
```

```
60:     foreach $p in $P do
61:     begin
62:       MUTEX_$O_GD => MUTEX_$O_$p_GD;
63:     end;
64:   end;
65:   -
66:   - Object method access (local process context)
67:   - for each method ...
68:   -
69:   init: #access
70:   begin
71:     #data
72:     begin
73:       MUTEX_$O_INIT <= MUTEX_$O_GD when $ACC else '0';
74:     end;
75:     #control
76:     begin
77:       wait for MUTEX_$O_GD = '0';
78:     end;
79:   end;
80:   lock: #access
81:   begin
82:     #data
83:     begin
84:       MUTEX_$O_LOCK <= MUTEX_$O_GD when $ACC else '0';
85:     end;
86:     #control
87:     begin
88:       wait for MUTEX_$O_GD = '0';
89:     end;
90:   end;
91:   unlock: #access
92:   begin
93:     #data
94:     begin
95:       MUTEX_$O_UNLOCK <= MUTEX_$O_GD when $ACC else '0';
96:     end;
97:     #control
98:     begin
99:       wait for MUTEX_$O_GD = '0';
100:    end;
101:  end;
102:  -
103:  - Implementation (global module context)
104:  - VHDL signals required, both for mapping processes
105:  - and auxilliary signals.
106:  -
107:  #signals
108:  begin
109:    -
110:    - Implementation signals
111:    -
112:    foreach $p in $P.lock do
113:    begin
114:      signal MUTEX_$O_$p_LOCK: std_logic;
115:    end;
```

```
116:    foreach $p in $P.unlock do
117:    begin
118:      signal MUTEX_$O_$p_UNLOCK: std_logic;
119:    end;
120:    foreach $p in $P.init do
121:    begin
122:      signal MUTEX_$O_$p_INIT: std_logic;
123:    end;
124:    foreach $p in $P do
125:    begin
126:      signal MUTEX_$O_$p_GD: std_logic;
127:      signal MUTEX_$O_$p_LOCKed: std_logic;
128:    end;
129:  end;
130:  #signals ($scheduler="static")
131:  begin
132:    signal MUTEX_$O_LOCKed: std_logic;
133:  end;
134:  -
135:  - Implementation (global module context)
136:  - Scheduler process: access serialization
137:  -
138:  MUTEX_$O_SCHED: #process ($scheduler="static" and $fsm="moore")
139:  begin
140:    if $CLK then
141:    begin
142:      if $RES then
143:      begin
144:        MUTEX_$O_LOCKed <= '0';
145:        foreach $p in $P do
146:        begin
147:          MUTEX_$O_$p_GD <= '1';
148:        end;
149:        foreach $p in $P.lock do
150:        begin
151:          MUTEX_$O_$p_LOCKed <= '0';
152:        end;
153:      end
154:      else
155:      begin
156:        foreach $p in $P do
157:        begin
158:          MUTEX_$O_$p_GD <= '1';
159:        end;
160:        sequence
161:        begin
162:          foreach $p in $P.init do
163:          begin
164:            if MUTEX_$O_$p_INIT = '1' then
165:            begin
166:              MUTEX_$O_LOCKed <= '0';
167:              MUTEX_$O_$p_GD <= '0';
168:              foreach $l in $P.lock do
169:              begin
170:                if MUTEX_$O_$l_LOCKed = '1' then
171:                begin
```

```
172:                    MUTEX_$O_$l_LOCKed <= '0';
173:                    MUTEX_$O_$l_GD <= '0';
174:                  end;
175:                end;
176:              end;
177:            end;
178:            foreach $p in $P.lock do
179:            begin
180:              if MUTEX_$O_$p_LOCK = '1' and MUTEX_$O_LOCKed  = '0' then
181:              begin
182:                MUTEX_$O_LOCKed <= '1';
183:                MUTEX_$O_$p_LOCKed <= '1';
184:                MUTEX_$O_$p_GD <= '0';
185:              end;
186:            end;
187:            foreach $p in $P.unlock do
188:            begin
189:              if MUTEX_$O_$p_UNLOCK = '1' then
190:              begin
191:                MUTEX_$O_LOCKed <= '0';
192:                MUTEX_$O_$p_LOCKed <= '0';
193:                MUTEX_$O_$p_GD <= '0';
194:              end;
195:            end;
196:          end;
197:        end;
198:      end;
199:    end;
200:
```

EXAMPLE 4: **Extract from the mutex EMI implementation.**

Finally, objects can be created and used within ConPro modules and processes, shown in example **5**. A mutex is used to guard a global counter x, though the access of this register is implicitly guarded, the expression x ← x + 1 is not mutual, and must be guarded explicitly to guarantee data consistency.

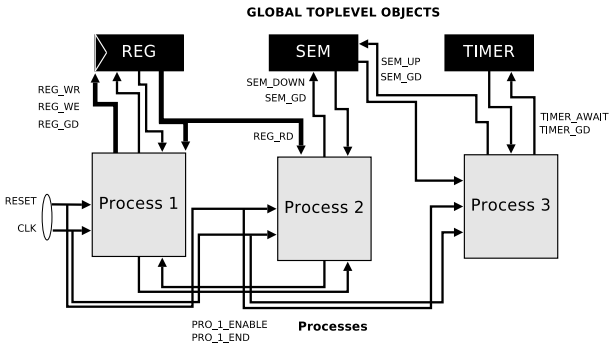
```
1:    open Core; open Process; open Mutex;
2:    object mu_x: mutex with schedule="fifo";
3:    reg x: int[6];
4:    process p1:
5:    begin
6:      for i = 1 to 10 do
7:      begin
8:        mu_x.lock (); x ← x + 1; mu_x.unlock();
9:      end;
10:   end;
11:   process p2:
12:   begin
13:     for i = 1 to 10 do
14:     begin
```

```
15:       mu_x.lock (); x ← x - 1; mu_x.unlock();
16:     end;
17:   end;
18:   process main:
19:   begin
20:     mu_x.init (); x ← 0;
21:     p1.start (); p2.start ();
22:   end;
```

EXAMPLE 5: **Abstract Data Type Object creation and method access inside ConPro processes.**

Each high-level process is mapped to a FSM and RTL, already shown in figure **2**. Process instructions are mapped to states of the FSM. Figure **3** shows the process system interconnect using signals. Access of objects is request-based and requires a request signal fed into a mutex-guarded access scheduler, responsible for serialization of concurrent access by different processes. A guard signal is read by the process FSM, providing a simple and efficient two-signal handshake (REQ↔ACT). Each shared object implements an access scheduler block, consisting of an FSM, providing the interface between processes accessing a resource and the resource itself.
The process block interface and system interconnect shown in figure **3** require different signals for the control and data path. Shared objects can be connected to different processes, requiring control signals for atomic access (called guards). All processes and objects are sourced by one system clock and reset signal, thus all functional blocks operate synchronously.

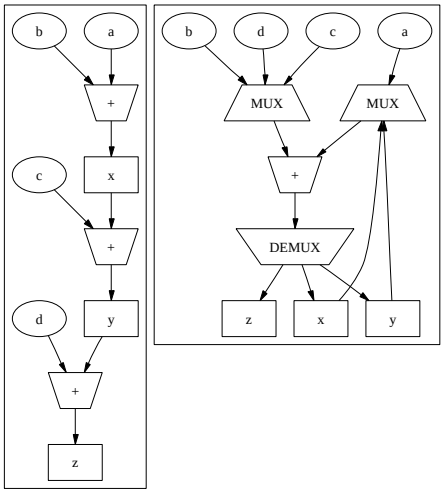
FIGURE 3: **The process block interface and system interconnect.**



Two different scheduling policies are supported: a simple static priority scheduler and a dynamic FIFO-based scheduler. The first one assigns each process a static priority during compile time. The resource is scheduled in priority order. This can lead to race conditions, whereby one or some processes always get access to the resource, and others never. The dynamic scheduler stores process identifiers in a queue and guarantees resource access in the order the requests arrived.
Local objects are directly implemented in RTL of a process, whereby global shared objects are implemented in separated hardware blocks, connected to processes using signals and to external circuit signals (at least clock and reset). The hardware architecture of a global object consists of the access scheduler block explained above, and the implementation blocks of this object, for example a RAM or communication transmitter. The access scheduler is the interface between the processes (accessing this object) and the implementation blocks (processing the object request).
The structure of the data path depends of chosen expression and temporary register model (shared versa exclusive), shown in graphs **2** and **3**. The

structure of the control path depends on instruction binding and scheduling using bounded blocks and different scheduling strategies, on loop parameters (loop-unrolling), and finally on optimization.

GRAPH 2: **Comparison of allocation in different expression models: flat (left) versa shared (right). Instruction sequence:** `x←a+b;y←x+c;z←y+d`.



GRAPH 3: **Comparison of allocation in different expression models: flat (left) versa shared (right) and shared temporary register model. Instruction: `z←a+b+c+d`. Additionally the binary expression model is shown in the middle subgraph.**

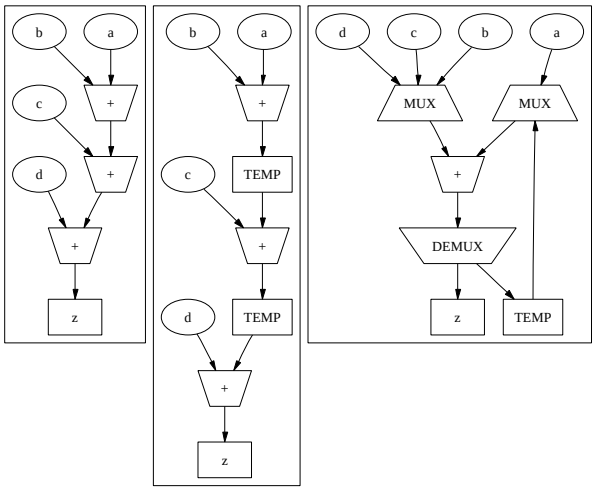


Figure **4** shows the system architecture of a shared function block.

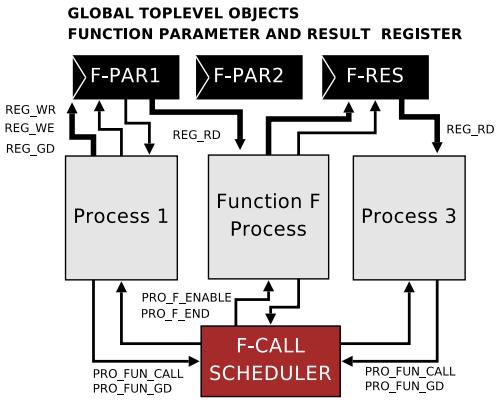


FIGURE 4: **Shared function blocks are implemented with a process block and a function call scheduler. Only some of the interconnect signals are displayed.**

6 Synthesis

The synthesis process is a traditional software compiler flow. The synthesis of RTL circuits from high-level imperative programs passes different phases:

1. First, the source code is **parsed and analyzed**. For each process, an abstract **syntax graph** preserving complex statements is built. Global and local objects are stored in symbol tables (one globally for module level, and one for each process level).

   First optimizations are performed on the process instruction graph, for example, constant folding and dead object checking, and elimination of those objects and superfluous statements.

   Several program transformations (based on rules and pattern matching) are performed, for example inference of temporary expressions and registers.

   A symbolic source code analysis method, called **reference stack scheduler KU92**, examines (local) data storage objects and their history in expressions. The **reference stack scheduler** analyzes the evaluation of data storage expressions with an expression stack, one for each object.

   The reference stack transforms a sequence of storage assignments with expression E κ=$\{\Theta \leftarrow E_1, \Theta \leftarrow E_2, ...\}$ of a particular storage object Θ to a sequence of immutable and unique symbolic variables Θ_i: $\{\Theta_1 \leftarrow E_1, \Theta_2 \leftarrow E_2, ...\}$. The aim is to reduce statements (using backward substitution and constant folding) and superfluous storage. The reference stack scheduler has an ALAP scheduling behaviour.

2. After analysis and optimization on instruction graph level, these complex instructions (ranging from expressions to loops) are transformed into a linear list of μ**Code** instructions, shown in table **2**. The μCode level is an **intermediate representation** of the program code, used in software compilers, too, though no architecture-specific assumption is made on this level, except constraints to the control flow. *The μCode can be exported and imported, too. This feature enables a different entry level for other programming language frontends, for example, functional languages* **SHA98**.

   This intermediate representation allows more fine-grained optimization, allocation, and scheduling. The transformation from syntax graph to μCode infers auxiliary instructions and register (suppose for-loops which require initialization, conditional branching, and loop-counter statements).

   Parallelism on data path level is provided by the bind instruction which binds N instructions to one FSM state (one time unit).

   The transformation is based on a **set of rules** $\chi_{\kappa \rightarrow \mu}$, consisting of default rules and user-selectable rules (constrainted rules). This is the

first phase of architecture synthesis by replacing the paradigms of the source language with paradigms of the target machine, in this case a FSM with statements mapped to states and expressions mapped to the data path (RTL). Additionally, the first phase of allocation is performed here.

Data path concurrency is explored either by user-specified bounded blocks or by the **basicblock scheduler**. This scheduler partitions the μCode instructions into basicblocks. These blocks have only one control path entry at the top and one exit at the tail. The instructions of one basicblock (called major block) are further partitioned into minor blocks (containing at least one instruction or a bounded block). From these minor blocks data dependency graphs (DDG) are built. Finally the scheduler selects data-independent instructions from these DDGs with ASAP behaviour.

3. After the first synthesis level, the intermediate **μCode is mapped to an abstract state graph RTL** using a **set of rules** $\chi_{\mu\rightarrow\Gamma\Delta}$, too, again consisting of default and user-selectable rules. A final conversion step emits VHDL code. This design choice provides the possibility to add other/new hardware languages, like Verilog, without changing the main synthesis path.

   The rule set determines resource allocation of temporary registers and functional blocks providing different allocation strategies: shared versa non-shared objects and flat versa shared functional operators and inference of temporary registers. Shared registers and functional blocks introduce signal selectors inside the data path.

   RTL is partitioned into a state machine FSM (two hardware blocks, one transitional implementing the state register and one combinational implementing the state switch network), providing the control path, and the data path (consisting of transitional and combinational hardware blocks, implementing functional operators, access of global resources and local registers).

   Using the default set of rules, each μCode instruction (except those bounded) is mapped to one state of the FSM requiring one time unit ($\geq$ 1 clock cycle, depending on object guards). Scheduling is mainly determined by the rule set $\chi_{\kappa\rightarrow\mu}$, rather by $\chi_{\mu\rightarrow\Gamma\Delta}$.

Mnemonics	Descriptions	Effect
`move(dst,src)`	Data transfer	Δ:dst←src
`expr(dst,op1,op,op2)`	Data transfer with binary expression evaluation	Δ:dst←op(op1,op2)
`jump(label)`	Unconditional branch	Γ:σ←σ(label)
`falsejump(cond,label)`	Conditional branch	Γ:σ(label)$\|_{\neg cond}$
`bind(n)`	Bind n following instructions to a parallel execution block	$\{\mu_1,\mu_2,...\}\rightarrow\sigma$
`fun obj.meth(args)`	Abstract Object Method Call	Γ:$\sigma\|_{obj}$ Δ:$params_{obj}\leftrightarrow$args
`nop`	No operation place holder, mostly a result of optimization	-

TABLE 2: **μ-Code instructions and their effect on data- and control path (Δ, Γ).**

Though no traditional iterative scheduling and allocation strategies are used in this software compiler flow, the non-iterative constraint selective rule based synthesis approach provides inherent scheduling and allocation with strong impact from different optimizers. Summarized there are differenet levels of scheduling and allocation:

Reference Stack Scheduler

Operates on syntax graph level and tries to reduce statements, functional operators, and storage, and has impact on scheduling and allocation.

Basicblock Scheduler

Operates on intermediate μCode level and tries to reduce operational time steps of statements and has only impact on scheduling.

Expression Scheduler

To meet timing constraints, mainly clock-driven, complex, and nested flat expressions must be partitioned into subexpressions using temporary registers and expanded scheduling. This scheduler has impact both on scheduling and allocation.

Optimizer

Classical constant folding, dead code and object elimination, and loop/branch-invariant code transformations further reduce time steps and resources (operators and storage).

Synthesis Rules

But finally the largest impact on scheduling and allocation comes from the set of synthesis rules $\chi = \chi_{\kappa \to \mu} \cup \chi_{\mu \to \Gamma\Delta}$.

The ConPro synthesis tool was entirely implemented using the functional language ML (OCaML, about 70000 source code lines).
Figure **5** summarizes the synthesis flow. During the synthesis process **SYNTAX** $\to\mu$**CODE** $\to$**FSM-STATES** $\to$**RTL-VHDL**, initial block structures from high level control environments (for example branches) are kept, together with source code informations.

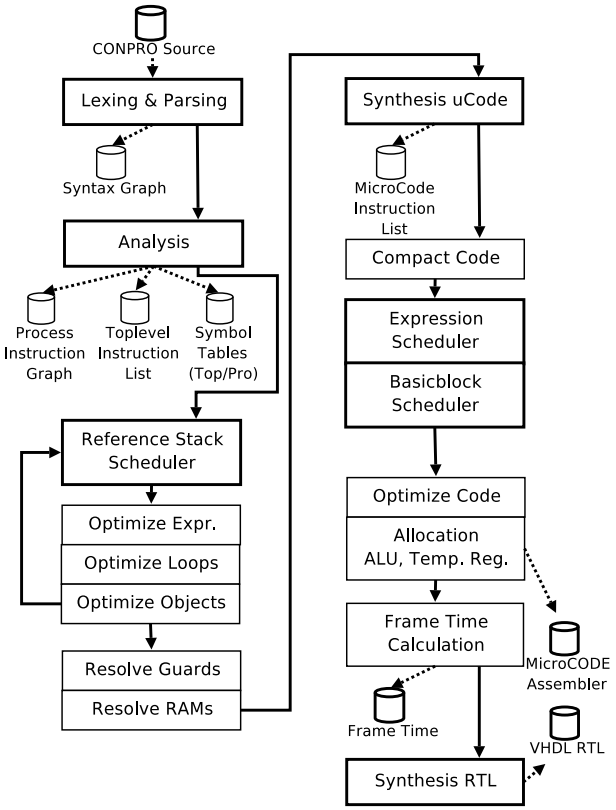


FIGURE 5: **This figure gives an overview about the ConPro synthesis process, from source code to VHDL RTL.**

7 Case Study: A Robot Actuator Network

This case study is part of an ongoing project called ModACT founded by ISIS. Goal of the project is the design and development of a modular robot actuator with a decentralized overall control architecture. Several actuators and sensors are connected in a network. Apart from system architecture and design issues, modern microsystem technologies and integration are related work in this project. Thus, miniaturization and low power design must be satisfied. The high density sensor and actuator network consist of independent nodes, each equipped with: 1. data processing, 2. control and 3. communication blocks **SSI10**. The functionality of each node is implemented entirely in register-transfer-digital-logic using FPGAs (Xilinx, SpartanIII-1000/500).

An actuator node consists of: 1. a brushless DC-motor, 2. a harmonic-drive-gear, 3. highly miniaturized electronics performing data processing and control, 4. different sensors for angular position, current and temperature, and finally 5. communication.

A sensor node consists of: 1. a sensor (for example a force/pressure sensor), 2. electronics, and 3. communication.

The ModuACT actuator controller was designed with the multi-process programm model explained in this article and partitioned into 46 behavioural processes and 9 shared functions. Figure **6**, **8** and **9** show most of the processes and their interconnection using IPC, mainly queues. The SoC-hardware-design can be synthesized for FPGA and ASIC technologies.

Control Design

Figure **6** shows the main parts of the actuator controller consisting of data acquisition (component ADC), data processing performed by process `service`, a reactive loop activated periodically by the timer `service_timer` (time period can be set between 10 μs and 10ms). The controller uses a register file consisting of 256 registers (implemented with a scheduled RAM block), and each cell has a datawidth of 12 bit. Each register is assigned to a special purpose, for example PID controller constants, actual measuring results (current, position, temperature and many more).
The position controller is a traditional PID calculator, activated periodically from timer `service_timer`, too.
The brushless DC motor requires a three-phase pattern generator to supply the phase-synchronous driving voltages (approximated sinus waves). Each motor coil voltage is generated by PWM generators, performed with process `pwm_gen.[0]` to `pwm_gen.[2]`. The PWM period time is about 30 μs, activated by the timer `pwm_timer`. Each PWM generator process is synchronized by the process `pat_update` with actual data, getting itself data from the main pattern generator process `pat`. This process also linearizes the PWM amplitudes (electronic H-bridge compensation). Data is passed by global shared registers. The sinus wave period is different from the PWM period, therefore the process `pat_syncer` activates the pattern generator.

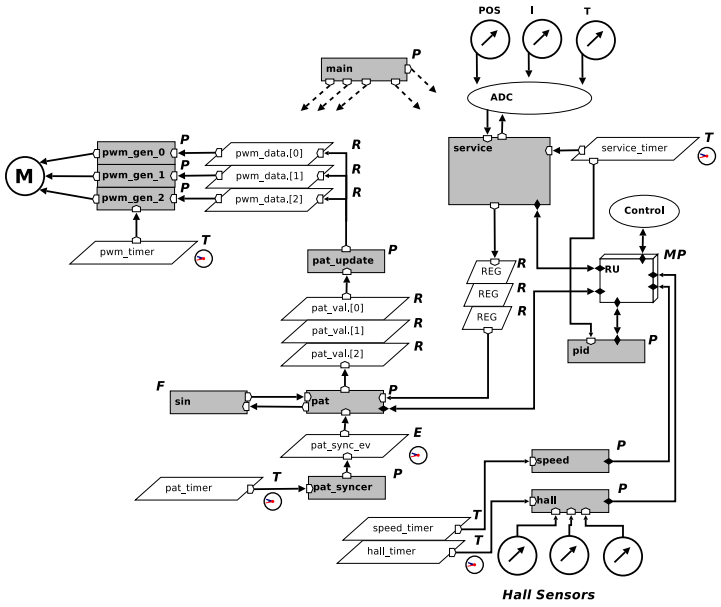


FIGURE 6: **ModuACT architecture, part 1: control and data processing.**

The pattern geneator requires feedback from rotor position of the motor, detected by three magnetic hall sensors, whose data is processed by processes `hall` and `speed`. They are activated by timers `hall_timer` (1 μs) and `speed_timer` (10 μs), respectively.

Communication Design: SLIP

Communication is an essential part of a complex robot systems, consisting of several actuators and sensor nodes. The Scalable Local Intranet Protocol (SLIP) and a communication controller design was developed. The goals were: 1. low power design and efficiency, 2. low resource design, SoC capable, and 3. adaptable to local communication requirements. SLIP is scalable with respect to network size (address size class ASC), maximal data payload (data size class DSC) and the network topology dimension size (address dimension class ADC).

Network nodes are connected using (serial) point-to-point links, and they are arranged along different metric axes of different geometrical dimensions: a one dimensional network (ADC=1) implements chains and rings, a two-dimensional network (ADC=2) can implement mesh grids, a three-dimensional (ADC=3) can implement cubes, and so on. Both regular (complete) and irregular networks (with missing nodes) are supported for each dimension.

The main problem in message-based communciation is routing and thus addressing of nodes. Absolute and unique addressing of nodes in a high-density sensor-actuator-network is not suitable. An alternative routing strategy is delta-distance routing, used by SLIP. A delta vector Δ specifies the way

from the source to a destination node.
An example network (ADC=2) with five nodes connected by bidirectional point-to-point links is shown in figure **7**. Suppose a message should be sent from Node 1 to Node 5. The relative distance is Δ=(2,-1).

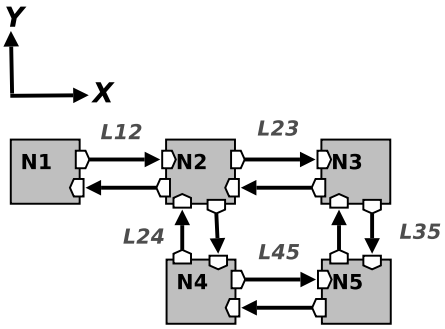


FIGURE 7: **An example network (ADC=2) with five nodes connected by bidirectional point-to-point links.**

A message packet contains a header descriptor specifying the type of the packet and the scalable parameters ASC, DSC and ADC.
A packet descriptor follows the header descriptor, containing: the actual delta-vector Δ, the original delta-vector Δ^1, a backward-propagation vector Γ, a preferred routing direction ω, an application layer π, and the length of the following data part.
Each time a packet is forwarded (routed) in some direction, the delta-vector is decreased in the respective dimension entry. For example, routing in x-direction results in: $\Delta_1 \leftarrow \Delta_1$-1.
A message has reached the destination if Δ=0 and can be delivered to the application layer π.
There are different smart routing rules, applied in order showed below untill the packet can be routed (or discarded):

route_normal This is the main routing rule with the goal to decrease the distance from the actual node to the destination node, finding the shortest path: min|Δ|.
Each i-dimension of the Δ-vector is checked: if is $\Delta_i > 0$, and there is a link in i-direction, route packet with $\Delta_i \leftarrow \Delta_i$-1, else if $\Delta_i < 0$, and there is a link in -i-direction, route packet with $\Delta_i \leftarrow \Delta_i$+1, else try next rule. The starting dimension is specified in the packet descriptor.

route_opposite Try to send the packet on any other link (but not the direction from which the packet has arrived), resulting in a partial increase of distance. The opposite travel is marked in the Γ-entry.

route_backward The packet is trapped, no further way to route the packet. Send back the packet on the direction from wwhich it has arrived. The backward travel is marked in the Γ-entry.

For example, the path N1→N5 can be directly routed using normal routing, first x-direction, finally y-direction. Now assume the link L35 is down, and the packet arrives at node N3, but must be send back to node N2, to change the routing direction, and finally arrives N5 over link L45. The above set of smart routing rules solve such backend-trap problems, supporting irregular network topologies and robustness against link failures. The way back from N5→N1 requires opposite routing at node N4!
The SLIP protocol stack was implemented for ADC=2, ASC=8, and DSC=8, using a partition of 21 processes and 4 functions. Queues are used to supply buffering and synchronized data exchange between processes. Packet structures and packet data are managed in different pools (uisng scheduled RAM arrays).
Figures **8** and **9** show the architecture of the protocol stack. The application layer is implemented with process `proto`, providing an RPC-based interface to the controller register file.

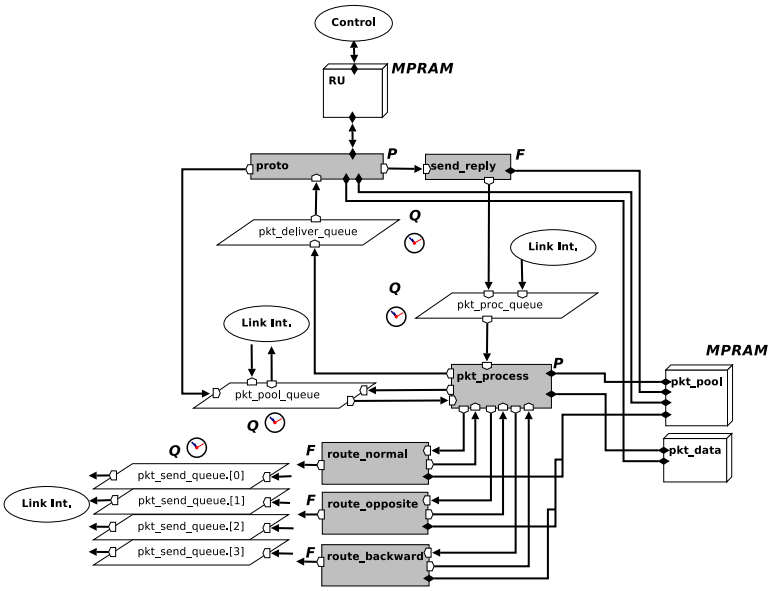


FIGURE 8: **ModuACT architecture, part 2: communication and SLIP protocol stack.**

Incoming data (from a serial link `link.[i]` with i=1..4) is stored in the first data queue `link_rx.[i]`. The data is parsed by the process `link_rx_proc.[i]`, reading from the data queue. The process `link_tmo.[i]` provides timeout management. If a packet was successfully received, the packet header is stored in the queue `pkt_proc_queue`. Packet are processed by the process `pkt_process`, which tries to route the packet using the above routing rules (implemented in three function blocks) or delivers the packet to the application layer. Outgoing packets are processed also by `pkt_process`. The routing functions pass the packet to the appropi-

ate queue `pkt_send_queue.[i]`, wich is processed by the packet encoder `link_tx_proc.[i]`. This process sends the data stream to the link transmitter.

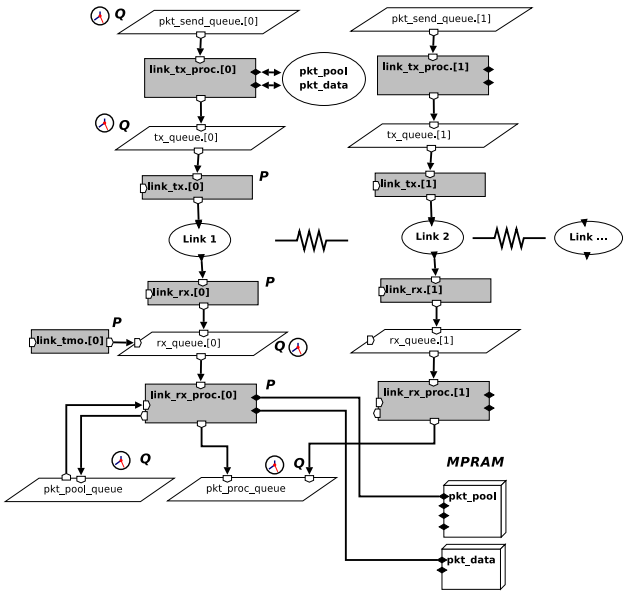


FIGURE 9: **ModuACT architecture, part 3: communication link interface.**

Synthesis and Results

The full design consisting of the actuator controller, data processing and communication was synthesized for two target technologies using two different design flows: 1. XilinX-FPGA, Spartan III-1000, Xilinx-ISE synthesis with place & route backends, and 2. standard-cell-library ASIC using the lsi_10k library and the design compiler from Synopsys. The second one was only used to get a good area estimation of the SoC-design.

Table **3** shows results of the FPGA synthesis. Though a high complex design was implemented, the design fits in a medium sized FPGA. The FPGA resources are fully mapped, but the overall performance (longest comb. path) is still high (up to 60 MHz clock frequency). Table **4** shows results of the ASIC synthesis. This design requires about 300k gates area. In contrast one microprocessor core (for example Leon-2, Sparc V8, using Virtex XCV800 FPGA) requires about 600k gates and achieves only 20 MHz clock frequency **LEO06**.

Table **5** shows synthesis results of some selected processes. Though FSMs have a high number of states, the required area and register count is still low.

Parameter	Value
number of source code lines (ConPro)	3381
number of synthesized VHDL lines	36531
total eq. gate count	1433516
longest path time	15 ns (66 MHz clock)
number of 4-input LUTs	13448 (15360)
number of block RAMs	20 (24)
number of FSMs	48
number of flip-flops	3666

TABLE 3: **Summarization of VHDL synthesis results of robot joint controller using XIlinx ISE 9.2 and a XIlinx Spartan-III FPGA**

Parameter	Value
number of cells	71870
number of nets	81624
combinatorial area	118829 eq. gates
non-comb. area	188940 eq. gates
total area	307769 eq. gates

TABLE 4: **Summarization of VHDL synthesis results of robot joint controller using Synopsys Design Compiler and the lsi_10k standard-cell library.**

Process	States	Area	Register
pkt_process	32	587	42
FUN_route_normal	72	1269	57
FUN_route_opposite	55	879	31
FUN_route_backward	86	1244	34
FUN_sin	33	2318	71
pid	19	3723	86
proto	135	3404	136
service	90	2438	104

TABLE 5: **Summarization of synthesis results of some selected processes. The FSM state number is calculated by ConPro, area in equivalent gates and registers by the design compiler (lsi_10k library).**

8 Conclusion and Outlook

In this paper, a multi-process programming model was presented for SoC design to explore concurrency required in complex systems like Cyber Physical Systems for control, data processing and communciation.
A new high-level language and a synthesis compiler ConPro used for complex circuit and SoC design was presented, closing the gap between software and hardware level. The programming language provides an algorithmic entry level with additional features for synthesis control concerning scheduling and allocation. True bit-scaled data types are supported. The programming model is based on a concurrent multi-process architecture with interprocess-communication primitives, providing coarse-grained parallelism explicitly modelled. Fine-grained parallelism is supported on data path level and can be explored by the synthesis tool, too.
The synthesis process maps process instructions to states of an FSM and RTL on hardware behaviour level with good performance and resource coverage, using a user selectable set of rules. VLSI design with about 1M gates and beyond can be designed. Synthesis results show good performance of the compiler and good matching results to target technologies like FPGAs compared with traditional microprocessor designs. Though a traditional software compiler design flow is used, the optimizers can reach well-optimized circuit designs. Main application fields are reactive systems, rather than functional and pipelined systems.
The programming-model and synthesis approach was proved with a complex robot actuator design.
In the future, pipelining of the data path must be supported to provide high performance synthesis of functional units.

Bibliography

LEE06 Edward Lee
Cyber-Physical Systems - Are Computing Foundations Adequate?
Position Paper for NSF Workshop on Cyber-Physical Systems: Research Motivation, Techniques and Roadmap, Austin, Texas, October 16-17, 2006

SHA98 Richard Sharp
Higher-Level Hardware Synthesis
Springer, 1998

HOA85 C. A. R. Hoare
Communicating Sequential Processes
Prentice Hall, 1985

VAN93 Jan Vanhoof, Karl Van. Rompaey, Ivo Bolsens, Gert Goossens, Hugo De Man
High-Level Synthesis for Real-Time Digital Signal Processing
Kluwer, 1993

KU92 David C. Ku, Giovanni De Micheli
High Level Synthesis of ASICs Under Timing and Synchronization Constraints
Kluwer, 1993

CON08 Stefan Bosse
ConPro: High-Level Hardware Synthesis With An Imperative Multi-Process Approach
Technical Paper, BSSLAB, Bremen, 2008

CON09 Stefan Bosse
ConPro: Rule-Based Mapping of an Imperative Programming Language to RTL for Higher-Level-Synthesis Using Communicating Sequential Processes
Technical Paper, BSSLAB, Bremen, 2009

SSI10 Stefan Bosse, Dirk Lehmhus
Smart Communication in a Wired Sensor- and Actuator-Network of a Modular Robot Actuator System using a Hop-Protocol with Delta-Routing
Smart Systems Integration, 23-24.3.2010, Italy, Como

ZHU01 Jianwen Zhu
MetaRTL: Raising the abstraction level of RTL Design
DATE ’01: Proceedings of the conference on Design, automation and test in Europe (2001), pp. 71-76

RU87 Steven M. Rubini
Computer Aids For VLSI Design
Addision Wesley 1987

CLA09 J. Hilljegerdes, P. Kampmann, S. Bosse, and F. Kirchner
Development of an Intelligent Joint Actuator Prototype for Climbing and Walking Robots
12th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines, 09-11 September 2009, Istanbul, Turkey

KAT02 V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist
PICO: Automatically Designing Custom Computers
IEEE Computer, 35 (9), pp 39-47, 2002

SPA04 Sumit Gupta, R.K. Gupta, N.D. Dutt, A. Nicolau
SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits
Kluwer Academic Publishers 2004

BAR73 Mario R. Barbacci
Automated Exploration of the Design Space For Register Transfer (RT) Systems
Thesis, 1973, Carnegie-Mellon-University

AND00 Greg Andrews
Multithreaded, Parallel, and Distributed Programming
Addison Wesley, 2000

HLS08 Philippe Coussy, Adam Morawiec (Ed.)
High-Level Synthesis - from Algorithm to Digital Circuit
Springer 2008

BAI01 John Baindridge
Asynchronous System-on-Chip Interconnect
Springer, 2001

LEO06 H. Miranda, J. Tombs, M. A. Echánove
Implementation of a low cost embedded system using the Leon2 processor
XXI Conference on Design of Circuits and Integrated Systems. Conference on Design of Circuits and Integrated Systems (21). Num. 21. Barcelona. Departamento de Electrónica. 2006.