ConPro: Rule-Based Mapping of an Imperative Programming Language to RTL for Higher-Level-Synthesis Using Communicating Sequential Processes

Dr. Stefan Bosse
BSSLAB - Independent Research Laboratory

The ConPro programming language, an new enhanced imperative programming language is mapped to Register-Transfer-Logic using a higher-level-synthesis approach performed by the synthesis tool ConPro. In contrast to other approaches using modified existing software languages likeC, this language is designed from scratch providing a consistent model for both hardware design and software programming. The programming model and the language provide parallelism on control path level using a multi-process model with communicating sequential processes (CSP), and on data path level using bounded program blocks. Each process is mapped to a Finite-State-Machine and is executed concurrently. Additionally, program blocks can be parameterized and can control the synthesis process (scheduling and allocation). Synthesis is based on a non-iterative, multi-level and constraint selective rule-set based approach, rather than on a traditional constrained iterative scheduling and allocation approach. Required interprocess communication is provided by a set of primitives, entirely mapped to hardware, already established in concurrent softwareprogramming (multi-threading), implemented with an abstract data type object model and method-based access. It is demonstrated that this synthesis approach is efficient and stable enough to create complex circuits reaching themillion gates boundary.

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

1. INTRODUCTION AND STATE OF THE ART

Today there is an increasing requirement for the development of application-specific digital circuits, with increasing complexity, too. Traditionally, these 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 [6].

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 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 which 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 about 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.

One example is PICO [10], adressing 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 of support of true bit-scaled data objects.

Another example is SPARK [12], 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.

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 [15]. Interprocess communication is modelled on transaction level (TLM). SystemC provides a high-level-approach still to model hardware behaviour and structure, rather than algorithms.

Non of these approaches fully satisfy the requierements for pure RTL circuit design while using C-based languages, especially providing a consistent hardware, software, and concurrency model. A more dedicated programming language can close the gap between software and hardware design rules, explained in this article. 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 on the design of a programming language, 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), reflecting both the algorithmic level using storage objects, the hardware level using signals, and finally fine-grained synthesis control using constrained rules.

Summarized the ConPro language and synthesis tool tries to break the limitations of extendend C-like language approaches for hardware design, while keeping an intuitive way to implement algorithms by software developers. 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 *4*, and finally section *5* gives an overview about the synthesis process and the synthesis rules. Examples and experimental results are discussed in section *6* and demonstrate the power and suitability of the synthesis approach and tool for complex circuit designs.

2. MODELLING AND IMPLEMENTING CONCURRENCY

Concurrency can be either modelled explicitly (not transparent) or implicitly (transparent) by the synthesis tool:

Explicit Parallelism

The programming model explicitly describes parallelism which means the programmer is responsible for modelling concurrency using for example processes or threads and synchronization primitives. 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.

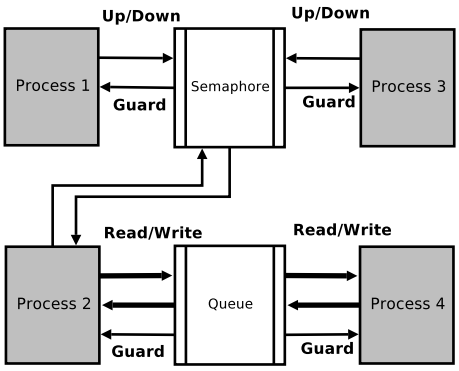
Implicit Parallelism

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 [1]. 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` ←`x +1` and `y` ←`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 [4].

There are several advantages of the explicit concurrency model versa the implicit model derived from initially pure sequential code, found in most extended C-like approaches [15], especially in the context of reactive systems. Knowledge based modelling of concurrency can lead to 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. The multi-process model requires explicit synchronization, shown in figure **1**. *Interaction between processes, mainly access of shared resources, is request-acknowledge based.*

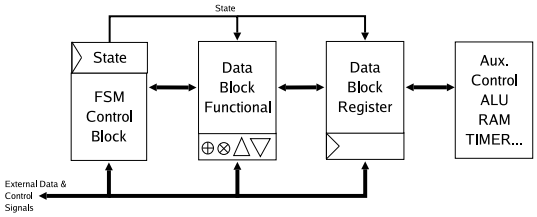
FIGURE 1: **The multi-process model with request-based synchronization (IPC).**



2.1 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.**

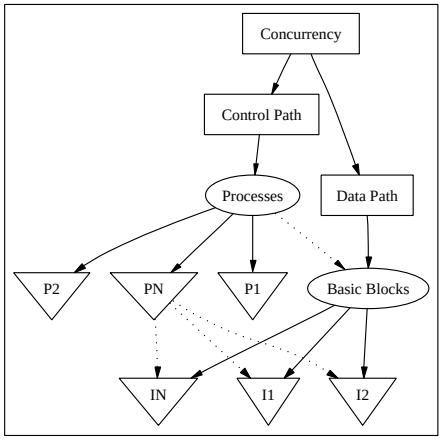


A process ϕ bounds a sequence of instructions $\kappa=\{\kappa_1,\kappa_2,...\}$ to this execution environment. Process instructions on programming level must be executed in the order they appear (imperative nature). Therefore, the set of program instructions κ can be directly mapped to a set of states Σ of the FSM, implemented entirely in RTL.

An algorithm can be partitioned on control path level using a set of N processes $\Phi=\{\phi_1,\phi_2,...,\phi_N\}$, executing initially independently and concurrently, doing communictaion, based on the model of communicating sequential processes (CSP) proposed in [2]. 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 proofed in this article. The ***ConPro*** *language [5] 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, shown in graph* **1**. Synthesis of RTL from an imperative programming language providing the multi-process-model can be superior compared with traditional hardware-software-co-design using microprocessors because abstract objects, especially all kind of interprocess-communication, can be implemenetd more efficiently in hardware than in software, both concering resources and latency.

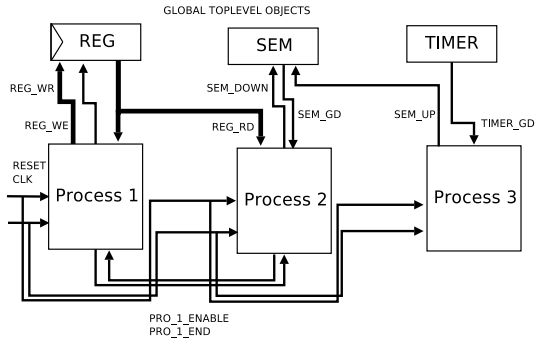
GRAPH 1: **Different levels of concurrency appearing in the control and data path.**



The set Φ of processes belong to a module. On module level, a set of global shared objects α=$\Re \cup \Im$ can be defined, and on process level, local objects can be defined. Processes can access both their local and the global objects. These objects α 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 to hardware components (entities in VHDL terminology), each consisting of 1. a FSM (state register and state transition network), 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**. 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.**



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 suspends the caller process untill the called (started) process reaches its end state.

2.2 Interprocess-Communication

Concurrency on control path level requires synchronization [14]. 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`

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

3. PROGRAMMING LANGUAGE

The ConPro programming language consist 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. Imperative programs which describe algorithms that execute sequentially from one statement to the next, are familiar to most programmers. But beneath algorithmic statements the programming language must provide some kind of relation to the hardware circuit synthesized from the programming level. *The syntax and semantics of the programming language is consistently designed and mostly self-explanatory, without cryptic extensions, required in most hardware C-derivates, like Handel-C or System-C, providing easy access to digital circuit development, also for software programmer.* Additionally there is a requirement to get full programmability of the design activities themselves, that means of the synthesis process, too [7], 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 splitted into two classes: 1. data storage type set $\Re$, and 2. abstract data object type set (ADTO) Θ, with a subset of the IPC objects $\Im$. Though it is a traditional imperative programming language, it features true parallel programming both in control and data path, explicitly modelled by the programmer.

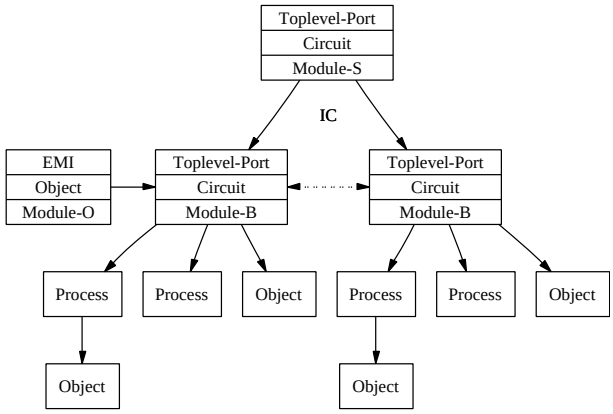
Processes provide parallelism on control path level, whereby arbitrary nested bounded blocks inside processes provide parallelism on data path level.

There is an extended interface to connect to external hardware objects.

3.1 Modules, Processes and Circuits

The hierarchical structure of a design is shown in graph **2**.

GRAPH 2: **Different design hierarchy levels and object visibilities.**



There are different hierarchy levels:

(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 finit-state-machine, computational units and local objects.

DEFINITION 1: **Definition of processes and process arrays.**

```
◆ object definition ◆
object o1,o2,: object type;
◆ process definition ◆
process pname:
begin
  ⟨process object definitions⟩
  ⟨process instructions⟩
end [with parameters];
◆ process array definition ◆
array paname: process[N] of
begin
  ⟨process object definitions⟩
  ⟨process instructions⟩
end [with parameters];
```

DEFINITION 2: **Definition of structural modules.**

```
◆ Structural module definition ◆
module mname:
begin
  ◆ Import one or more modules ◆
  import M,...;
  ◆ Instantiate components of each module ◆
  component C1,C2,...:M;
  component ...:...;
  ◆ Some interconnect between components C1... ◆
  type inter_connect: {
    port C1_o1_WR: output logic[8];
    port C1_o1_RD: input logic[8];
    port C2_o1_WR: output logic[8];
    port C2_o1_RD: input logic[8];
    ...
  };
  ◆ Instantiate interconnect component ◆
  component M_c: inter_connect := {
    C1.TOP.o1_WR,
    C1.TOP.o1_RD,
    C2.TOP.o1_WR,
    C2.TOP.o1_RD,
  };
  ◆ Define some interconnections ◆
  M_c.C1_o1_WR >> M_c.C2_o1_RD;
  ...
end;
```

Single processes or an array of processes can be defined using the process environment, behavioural modules do not require special defintions, and structural modules can be defined using the module environment, shown in definitions **1** and **2**.

A process environment consists of a unique process name, local object definitions and process instructions. Processes are bound to a behavioural module, containing additional shared objects.

Behavioural modules are defined immediately by their respective source code file, which means a source file m.cp defines module M. A SoC is composed of imported behavioural module instantiations (import and component statement). An internal interconnect component is instantiated (type and component statement).

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 connecting module signals internally (which can be automatically generated using map instrcutions).

Finally, there is a third kind of subtype of a module: an abstract object module Module-O. It defines and implements abstract data types and the provided methods υ

Already mentioned above, processes are handled like ADTOs, too, and can be started, stopped or synchronously called by other processes using the appropiate object method: {`start,stop,call`}. Initially, a process is blocked in his start state, waiting for a start or call request. There is a main (master) process always present (like main in C) and started after system reset automatically.

3.2 Object Types

The set of object types α contains storage $\Re$, signals $\wp$, and abstract objects Θ={$\Im$,D,E}: α={$\Re$,Θ}. The set D contains data computational objects, for example, random generators and DSP units, and the set E contains external communication objects. Some object definitions are shown in example **1**.

3.2.1 *Data Storage* . 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.

Additionally, **queues** can be used for intermediate storage and synchronized

data exchange between processes preserving data order. Simple handshaken data transfer can be done with **channels** (buffered or unbuffered without intermediate register).

Storage objects (like any other objects) can be parameterized on object definition, for example the kind of scheduler used to serialize concurrent access.

Though there are variables (and arrays) implemented in RAM blocks, there are no pointer like objects.

3.2.2 *Abstract Object Data Types .* Beneath data storage objects which can be used within expressions (read and write access), there are **abstract objects**, which can be accessed with a set of methods υ only. Each abstract object belongs to a module definition and implementation (Module-O), which must be opened before first object definition. A method is applied to an object using the selector operator and a list of arguments passed to method parameters (or empty list for pure reactive methods): Θ`.`υ`(arg1,arg2,...)`.

An object definition (≡resource allocation) requires the specification of data and object type, β and α, respectively.

3.2.3 *Signals .* The signal type $\wp$ provides access and control on hardware level. There is no behaviour model behind this signal (in contrast to VHDL), it is just a connection wire with a specified logical signal level and a direction. The signal object can be used in expressions and assignments and provides read and write access like any other storage object, though write access is temporal limited to the activity window of the assignment. The signal type appears in component structures, too.

3.3 Data Types

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 contains different named elements with defined data types β. 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.

Array and structure cells are accessed using the selector operator already introduced for method access: `O.E` for structures, and `O.[I]` for arrays.

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

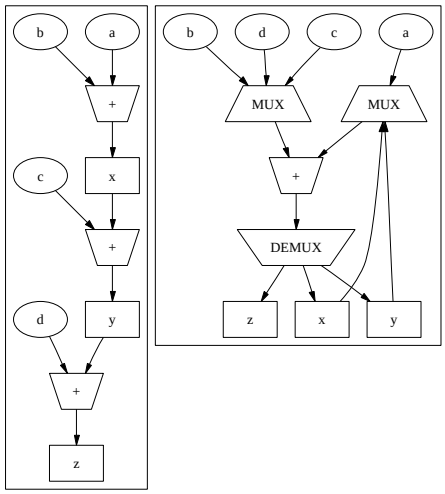
```
1:  ◆ Storage Objects ◆
2:  reg x,y: int[8]; Defines registers
3:  block ram1; Defines a block RAM
4:  var a,b,c: int[10] in ram1; Defines variables in RAM
5:  array mat1: reg[10] of int[23]; Defines an array
6:  array mat2: var[10] of int[8] in ram1;
7:  type complex: {
8:    real: int[16];
9:    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
```

3.4 Expressions, and Assignments

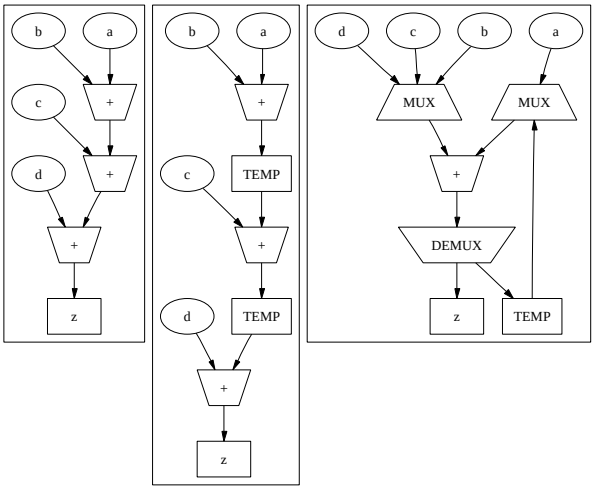
Expressions contain data storage objects, constants, and operators. Supported are all common arithmetic, Boolean (logical), and relational operators. Most of them are directly mapped to hardware behaviour level (VHDL operators). Initially, assignments to data storage objects are scheduled in one time 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 **2**, e.g. RTL scheduling originally proposed by Barbacci [13]. A time unit requires at least one clock cycle, but can consume more clock cycles in the case of access of guarded (shared) objects. Depending on selected synthesis rules, 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.

Graph **3** compares the RTL architecture and allocation of different expression models (flat versa shared) for the instruction sequence `x←a+b;y←x+c;z←y+d`. Graph **4** shows flat versa shared expression model with additional shared temporary register model (enabled with block parameter `temp="shared"`) for one instruction after scheduling with the reference scheduler: z←a+b+c+d. The binary expression model with non-shared temporary register model is also shown.

GRAPH 3: **Comparison of allocation in different expression models: flat (left) versa shared (right). Instruction sequence: x←a+b;y←x+c;z←y+d.**



GRAPH 4: **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.**



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).**

```
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";
```

3.5 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. Tables **2** and **3** summarizes available control statements and their impact on the control path Γ.

The unconditional always-loop is used for request-based server loops (infinite loop). The wait-for statement is used usually with signals, checking the value of an expression of signals and optionally applies a statement (usually signal assignments) until the condition E changes to value true. Because signals are assigned a value to only as long as the assignment is active, a default value can be specified with the optional else-branch. Also time delays can be implemented with the wait-for statement. Is the clock frequency of the system is known, time can be specified in second units.

3.6 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

Control Statement	Description	Γ

if E
  then A1
  [**else** A2];
	Boolean Conditional Branch (false-branch optional)	$\sigma \leftarrow \{\sigma(\texttt{A1})\|_{\texttt{E=true}}$ /
		$\sigma(\texttt{A2})\|_{\texttt{E=false}}$ /
		$\sigma+\|_{\texttt{E=false}}\}$

match E **with**
begin
  when E1: A1;
  when E2: A2;
...
end;
	Multi-Value Conditional Branch	$\sigma \leftarrow \{\sigma(\texttt{A1})\|_{\texttt{E=E1}}$ /
		$\sigma(\texttt{A2})\|_{\texttt{E=E2}}$ /
		...}

TABLE 2: **Available branch statements and impact on state change σ in control path Γ (σ: actual state, $\sigma+$ is next statement, {} is set of conditional state selections, and / mututal alternation).**

Control Statement	Description	Γ

while E
  do A;
always
  do A;
	Conditional and unconditional Loop	$\sigma \leftarrow \{\sigma(\texttt{A})\|_{\texttt{E=true}}$ /
		$\sigma+\|_{\texttt{E=false}}\}$

for i = a **to** b **do** A;	Counting Loop (to or down-to direction)	$\sigma \leftarrow \{\sigma(\texttt{A})\|_{\texttt{i}\in\{\texttt{a,b}\}}$ / $\sigma+\|_{\texttt{i}\notin\{\texttt{a,b}\}}\}$

waitfor E
  [**with** A1
  else A2]
	Conditional Delay (E can be time condition)	$\sigma \leftarrow \{\sigma\|_{\texttt{E=false}}$ /
		$\sigma+\|_{\texttt{E=true}}\}$

try A **with**
begin
  when ε1: A1;
  when ε2: A2;
...
end;
	Exception Catcher (A with **raise** ε statements)	$\sigma \leftarrow \{\sigma+\|_{\neg\texttt{raise}}$ /
		$\sigma(\texttt{A1})\|_{\texttt{raise}(\varepsilon 1)}$
		...}

TABLE 3: **Available loop statements and impact on state change σ in control path Γ (σ: actual state, $\sigma+$ is next statement, {} is set of conditional state selections, and / mututal alternation).**

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.

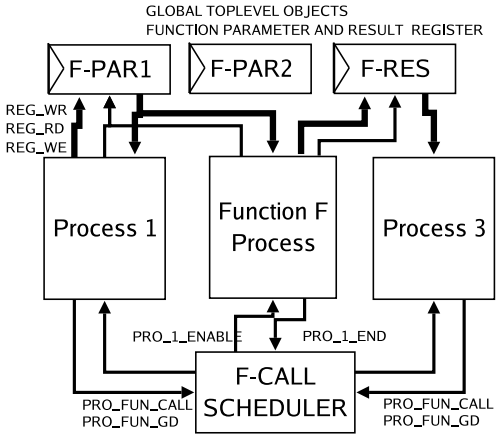


FIGURE 4: **Shared function blocks are implemented with a process block and a function call scheduler.**

Functions are restricted to non-recursive calls due to a missing stack environment.

3.7 Interface to Hardware-Blocks and the Real World

One main feature mostly missing in actual programming languages used in circuit design is the ability to interface non-transparent to exisiting hardware blocks, modelled on hardware behaviour level (VHDL). There are two choices: 1.non-transparent - using component structures and signals, enabling direct access to external hardware signals (including the toplevel port of the actual circuit), and 2. (semi-) transparent - more suitable the External Module Interface (EMI).

Components require a structure type definition with ports specifying the direction and the type of the port: `type T:{ port P: DIR` β`;...}` - and a component can be instantiated from this type: `component C: T;`. If the component is exported, it is part of the port interface of the actual behavioural module.

EMI allows the modelling of devices on hardware behaviour level and access of these devices using the abstract object interface with user supplied methods. An EMI module specification consists mainly of these sections: 1. method declaration, 2. method access on hardware behaviour level (hardware implementation of ConPro ADTO-method call), 3. hardware signal definitions required for access and implementation, 4. implementation of the abstract object itself, also on hardware behaviour level using a modified subset of VHDL. Example **3** shows some fragments of a random generator EMI object specification and the method access. The `#access`, `#signals` and `#process` sections use a modified VHDL language modelling the hardware level. The `#access` section defines data and control path activities of process RTL on hardware level. The `#process` section implements hardware blocks (at least the access scheduler and additional functional hardware blocks) using a modified subset of VHDL.

EXAMPLE 3: **Example of EMI object specification and access inside a process using method call.**

```
1:  open Random;
2:  object rand: random with datawidth=8;
3:  process p1:
4:  begin
5:    rand.read(x);
6:    ...
7:  end;
8:  ====== EMI specification ======
9:  #methods
10: begin
11:   read(#lhs:logic[$datawidth]);
12:   ...
13: end;
14: read: #access
15: begin
16:   #data begin
17:     RND_$O_RE <= RND_$O_GD when $ACC else '0';
18:     $ARG1 <= RND_$O_RD when $ACC else 0; end;
19:   #control begin
20:     wait for RND_$O_GD = '0'; end;
21: end;
22: #signals
23: begin
24:   signal RND_$O_d_in: std_logic;
25:   ...
26: end;
27: RANDOM_$O_SCHED: #process
28: begin
29:   if $CLK then begin
30:     if $RES then begin
31:       RND_$O_shift <= '0';
32:       RND_$O_init <= '0'; ...
```

4. RTL ARCHITECTURE

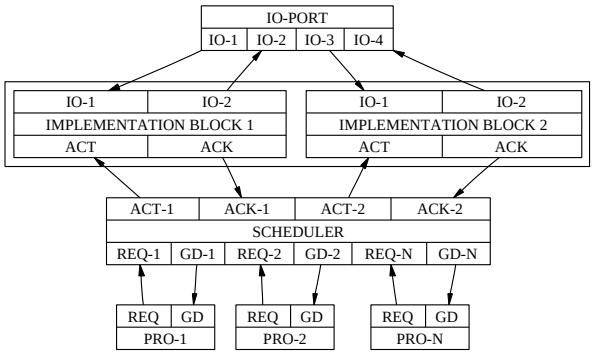
Each high-level process is mapped to a FSM and RTL, already shown in figure **2**. Process instructions are mapped to states of the FSM. Figure **3** showed already the process system interconnect using signals. Access of objects is request-based and requires a request signal fed into a mutex-guarded 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).

The scheduler interconnect is shown in graph **5**. There are two levels of handshakend synchronization: between each process i and the scheduler: {REQ-i,GD-i}, and between the scheduler and the object implementation block: {ACT,ACK}.

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.

Algorithms **1** and **2** compare both scheduling policies. They can be directly

GRAPH 5: **Access scheduler block architecture connecting both the port interface of processes (REQ and GD signals) and the interface of shared object (hardware) implementation blocks (activation and ackknowledge signals ACT/ACK and possible external IO connections).**



mapped to hardware RTL-level wih different resource requirements.

The static scheduler consists of an hierarchical mutual exclusive branch: A process-i request activates `REQ-i`, and if the resource is not locked (`LOCKED=false`), the request is granted to this process. Concurrent access selects the first process request in the if-then-else cascade. If the processing of the object request is finished, then `ACK` is activated and releases the locked object and releases `GD-i` for this respective process indicating that the request is finished.

The dynamic scheduler stores requests in a queue `LOCKED` (though the access scheduler is a hierarchical mutual exclusive branch, too): A process-i request activates `REQ-i`, and if the resource queue `LOCKED` is empty or this process is at the head of the queue, the request is granted to the process in the if-then-else cascade. If the processing of the request is finished, then `ACK` is activated and removes the process from the resource queue and releases `GD-i` for this respective process indicating that the request is finished.

ALGORITHM 1: **Static Priority Scheduler. From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}.**

```
if REQ-1 ∧ ¬LOCKED then
  LOCKED←TRUE;
  Raise ACT; Start Service for Process 1
else if REQ-2 ∧ ¬LOCKED then
  LOCKED←TRUE;
  Raise ACT; Start Service for Process 2
else if ACK ∧ REQ-1 ∧ LOCKED then
  Release GD-1;
  LOCKED ← FALSE;
else if ACK ∧ REQ-2 ∧ LOCKED then
  Release GD-2;
  LOCKED ← FALSE;
...
```

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).

ALGORITHM 2: **Dynamic Queue Scheduler. From/to process i:{REQ-i,GD-i}, from/to shared resource block:{ACT,ACK}.**

```
if REQ-1 ∧ LOCKED=[] ∧ ¬PRO-1-LOCKED then
  LOCKED ← [PRO-1];
  PRO-1-LOCKED ← TRUE;
  OWNER←PRO-1;
  Raise ACT; Start Service for Process 1
else if REQ-2 ∧ LOCKED=[] ∧ ¬PRO-2-LOCKED then
  LOCKED ← [PRO-2];
  PRO-2-LOCKED ← TRUE;
  OWNER←PRO-2;
  Raise ACT; Start Service for Process 2
...
else if REQ-1 ∧ LOCKED ≠ [] ∧ ¬PRO-1-LOCKED then
  LOCKED ← LOCKED @ [PRO-1]; Append Process 1 to Queue
  PRO-1-LOCKED ← TRUE;
else if REQ-2 ∧ LOCKED ≠ [] ∧ ¬PRO-2-LOCKED then
  LOCKED ← LOCKED @ [PRO-2]; Append Process 2 to Queue
  PRO-2-LOCKED ← TRUE;
...
else if REQ-1 ∧ Head(LOCKED)=PRO-1 ∧ OWNER≠PRO-1 then
  Raise ACT; Start Service for Process 1
  OWNER←PRO-1;
else if REQ-2 ∧ Head(LOCKED)=PRO-2 ∧ OWNER≠PRO-2 then
  Raise ACT; Start Service for Process 2
  OWNER←PRO-2;
...
else if ACK ∧ Head(LOCKED)=PRO-1 then
  Release GD-1;
  PRO-1-LOCKED ← FALSE;
  OWNER←NONE;
  LOCKED ← Tail(LOCKED);
else if ACK ∧ Head(LOCKED)=PRO-2 then
  Release GD-2;
  PRO-2-LOCKED ← FALSE;
  OWNER←NONE;
  LOCKED ← Tail(LOCKED);
...
```

5. SYNTHESIS

Synthesis of RTL circuits from high-level imperative programs can be divided into different phases [3]:

(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 optimiziations 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** [4], 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 a 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 **4**. 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* [1].
This intermediate representation allows more fine-grained optimization, allocation and scheduling. The transformation from syntax graph to μCode infers auxilliary 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 bind 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), explained later. 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 datapath (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 an 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 μ**Code 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$\leftarrow$src
`expr(dst,op1,op,op2)`	Data transfer with binary expression evaluation	Δ:dst$\leftarrow$op(op1,op2)
`jump(label)`	Unconditional branch	Γ:$\sigma\leftarrow\sigma$(label)
`falsejump(cond,label)`	Conditional branch	Γ:σ(label)$\vert_{\neg\text{cond}}$
`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:$\sigma\vert_{\text{obj}}$ Δ:params$_{\text{obj}}\leftrightarrow$args
`nop`	No operation place holder, mostly a result of optimization	-

TABLE 4: **μ-Code instructions and theit effect on data- and control path (Δ, Γ).**

Though no traditional iterative scheduling and allocation strategies are used, 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 statementes, 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 reduces 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 $\chi{=}\chi_{\kappa\rightarrow\mu}{\cup}\chi_{\mu\rightarrow\Gamma\Delta}$.

The ConPro synthesis tool was entirely implemented using the functional language ML (OCaML, about 70000 source code lines).

5.1 Synthesis Rules

Mapping of 1. process instruction κ to μCode instructions and 2. of μCode to RTL is done with a set of rules χ. Some rules depend on constraint settings, either globally set by a compiler setting or more fine-grained on block level using block parameters, shown in example **4**. The j-loop is unrolled using rule $\chi_{\mathrm{for-unroll|unroll}}$, and the process p1 is optimized (scheduled) using the reference stack and basicblock scheduler. Additionally, expressions are bound to a shared Arithmetic-Logic-Unit on process level (each process can have its own set of ALUs). Tables **6** and **5** show some of the synthesis rules and their effect and behaviour.

Using a set of simple mapping synthesis rules without iterative scheduling can lead to a non-optimized circuit regarding spatial and temproal requirements, but provide an isomorphic relation between RTL and algorithmic program level, simplifying back-annotation required for verification and simulation.

EXAMPLE 4: **Synthesis rule constraints on block level.**

```
1:  process p1:
2:  begin
3:    reg x,y,z: int[8];
4:    for i = 1 to 10 do
5:    begin
6:     z ← i * 4;
7:     for j = 1 to 5 do
8:     begin
9:       x ← (x + y) * z;
10:      y ← y - 1;
11:    end with unroll;
12:   end;
13: end with expr="shared" and
14:     scheduler="refstack,basicblock";
```

Synthesis Rule	Effect/Behaviour
$\chi_{\text{for-loop\|default}}$	For-loop: κ:for i = a to b do B → μ:move(LOOP_i,a) l1: bind(2) expr($immed.[0],LOOP_i,=,b) falsejump($immed.[0],l2) jump (l3) l2: B expr(LOOP_i,LOOP_i,+1) jump (l1) l3: ...
$\chi_{\text{for-loop\|unroll}}$	Unrolled for-loop: κ:for i = a to b do B → μ:∀ i in [a,b] repeat copy B(substitute LOOP_i with i)
$\chi_{\text{fun-call\|default}}$	Function Call: κ:y←f(x1,x2,..) → μ:move(REG_FUN_f_x1,x1) move(REG_FUN_f_x2,x2)... fun LOCK_FUN_f.lock () fun FUN_f.call () move(y,REG_FUN_f_res) fun LOCK_FUN_f.unlock ()

TABLE 5: **Some μ-Code synthesis rules and their synthesis effect (control models).**

Synthesis Rule	Effect/Behaviour
$\chi_{\text{expr\|expr=flat}}$	Expression with flat model: κ:(x1 op1 (x2 op2 x3) $\rightarrow$ μ:bind(2) expr($immed.[0],x2,op2,x3) expr(res,x1,op1,$immed.[0])
$\chi_{\text{expr\|expr=ALU}}$	Expression with shared model κ:(x1 op1 (x2 op2 x3) $\rightarrow$ μ:bind(2) expr($tmp.[0],x2,op2:ALU,x3) expr(res,x1,op1:ALU,$tmp.[0])

TABLE 6: **Some μ-Code synthesis rules and their synthesis effect (expression models).**

6. EXAMPLES AND EXPERIMENTAL RESULTS

Example **5** shows a complete ConPro design. It is the implementation of the dining philosophers problem using semaphores. 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 [14], here implemented with a semaphore array `fork`.

The read ports of the shared registers `eating` and `thinking` are exported to the module toplevel port. The design consists of seven processes. The philosophers are implemented with the process array `philosopher`. Gate-level synthesis with a standard cell technology [16] 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).

```
1:  open Core;
2:  open Process;
3:  open Semaphore;
4:  open System;
5:  open Event;
6:  object sys: system;
7:    sys.simu_cycles (500);
8:  object ev: event;
9:  array eating,thinking: reg[5] of logic;
10: export eating,thinking;
11:
12: array fork: object semaphore[5] with depth=8 and scheduler="fifo";
13:
14: process init:
15: begin
16:   for i = 0 to 4 do
17:     fork.[i].init (1);
18:   ev.init ();
19: end;
```

```
20:
21:   function eat(n):
22:   begin
23:     begin
24:       eating.[n] ← 1;
25:       thinking.[n] ← 0;
26:     end with bind;
27:     wait for 5;
28:     begin
29:       eating.[n] ← 0;
30:       thinking.[n] ← 1;
31:     end with bind;
32:   end with inline;
33:   array philosopher: process[5] of
34:   begin
35:     if # < 4 then
36:     begin
37:      ev.await ();
38:      always do
39:      begin
40:        -- get left fork then right
41:        fork.[#].down ();
42:        fork.[#+1].down ();
43:        eat (#);
44:        fork.[#].up ();
45:        fork.[#+1].up ();
46:      end;
47:     end
48:     else
49:     begin
50:      always do
51:      begin
52:        -- get right fork then left
53:        fork.[4].down ();
54:        fork.[0].down ();
55:        eat (#);
56:        fork.[4].up ();
57:        fork.[0].up ();
58:      end;
59:     end;
60:   end;
61:
62:   process main:
63:   begin
64:     init.call ();
65:     for i = 0 to 4 do
66:     begin
67:       philosopher.[i].start ();
68:     end;
69:     ev.wakeup ();
70:   end;
```

EXAMPLE 5: **A complete ConPro example: the dininig philosopher problem.**

EXAMPLE 6: **Different expression models EXPR={flat,binary,shared} are examined in the process p1. First for an addition operation (sum in process p1), and second for a multiplacation (prod in process p2).**

```
1:  process p1:
2:  begin
3:    reg sum,n: int[16];
4:    sum <- d,n <- 2;
5:    for i = 0 to 15 do
6:    begin
7:      sum ← sum + i + n;
8:      n ← n + 2;
9:    end with unroll;
10:   d ← sum;
11: end with expr=EXPR;
12:
13: process p2:
14: begin
15:   reg prod,n: int[16];
16:   prod ← d, n ← 2;
17:   for i = 1 to 16 do
18:   begin
19:     prod ← prod * i * n;
20:     n ← n * 2;
21:   end with unroll;
22:   d ← prod;
23: end with expr=EXPR;
```

Different expression models are examined in example **6**. Table **7** summarizes the synthesis results, using 1. standard cell target technology and gate-level synthesis (Mentor Graphics Leonardo Spectrum), and 2. Xilinx FPGA synthesis (ISE 9.2, Spartan III-1000 FPGA). Both addition and multiplication operations are evaluated. Sharing resources is not always usefull, which can be seen in the first example using adders (16 bit data width). In the first example inferring only adders, the additional multiplexer and demultiplexer using the shared expression and allocation model require more resources than the non-shared resource model. Especially if FPGA technolgy is used with embedded arithmetic units (DSP). Most expensive is the binary expression model. In contrast to the second example with multipliers. Here, the shared model decreases the required resources of about five times, in both target technologies (note: FPGA synthesis could not map all multipliers to embedded DSP blocks).

Expression Model	Gates, Flip-Flops, Path Delay
+,expr=flat, SXLIB	1098, 59, 17.0 ns
+,expr=binary, SXLIB	1742, 94, 13.5 ns
+,expr=shared, SXLIB	1278, 65, 14.9 ns
+,expr=flat, FPGA	1386, 57, 10.1 ns
+,expr=binary, FPGA	3270, 103, 11.1 ns
+,expr=shared, FPGA	2284, 89, 12.9 ns
*,expr=flat, SXLIB	20944, 140, 31.6 ns
*,expr=binary, SXLIB	8173, 138, 29.9 ns
*,expr=shared, SXLIB	4570, 94, 34.7 ns
*,expr=flat, FPGA	27220, 119, 21.8 ns
*,expr=binary, FPGA	11252, 103, 15.5 ns
*,expr=shared, FPGA	3879, 84, 22.9 ns

TABLE 7: **Synthesis results for different expression models of example 6.**

Several different complex designs reaching the million gates limit were implemented and synthesized successfully by the ConPro design framework. One main area of application is robotics which requires complex reactive control systems for robot joint actuators and communication [8][9]. Table **8** shows some selected projects with a short description and collected information about synthesis. All designs use a Xilinx Spartan-III-1000 FPGA as target technology. Gatelevel synthesis were performed by Xilinx ISE 9.2 software. The total high-level-synthesis time never exceeds 60 seconds on a Sun BLADE 2500 (2 x UltraSparc III, 1.6GHz) machine, whereby the gate-level synthesis reaches 20 minutes.

There is a ratio of about 1:10 of high-level source code lines to synthesized VHDL. The gatelevel synthesis tool estimates low longest-path delays resulting in high maximal clock frequencies. This value can be treated as a relative estimation factor for the evaluation of the quality of RTL design and architecture. Compared with generic microprocessor implementations in FPGAs usually achieving clock frequencies in the range of 50-100 MHz using hand-coded designs (for example [11] reported $\geq$ 50 MHz, Areoflex Gaisler [http://www.gaisler.com] reported 80 MHz for Leon-2 core using Xilinx Spartan/Virtex FPGAs, own results for Xilinx Spartan-III-1000 device-synthesis leads to 67 MHz prediciton), the automatic synthesized RTL design (without inference of special target technology mappings, magic blocks and vendor IP cores!) is comparable or better in performance. The RTL design incorporates all storage, whereby a microprocesser design exclude storage (data and code). Thus it is difficult to compare pure hardware and pure software designs implementing the same algorithm (program), concerning execution time and resource requirements. Other C-like higher-level-synthesis tools like SPARK [12] do not support concurrency to be modelled explicitly, and hence can not be used for comparing design issues because different algorithms (in terms of parallelism and partitioning concurrent tasks) must be used.

Project Description	Statistics
SLP: Simple Link Protocol Stack Communication Node	CPL=900,VHDL=10933 OBJ=93,PRO=12,RL=52 RG=14, ST=17s GT=696237, FF=1805, CLK=89 MHz
μJTAG: embedded JTAG TAP controller with μCode interpreter and communication	CPL=1563,VHDL=13650 OBJ=206,PRO=12,RL=40 RG=51, ST=19s GT=384878, FF=1967, CLK=84 MHz
ASGUARD: Robot Joint Controller with PID controller and communication [8]	CPL=1211,VHDL=26685 OBJ=233,PRO=22,RL=91 RG=30, ST=49s GT=204911, FF=2281, CLK=71.6 MHz

TABLE 8: **Some selected System-On-Chip projects with source code and synthesis statistics (CPL: total ConPro source code lines,VHL: synthesized VHDL lines, OBJ: number of object blocks ,PRO: numebr of processes, RL: local register, RG: global shared register, ST=synthesis time on UltraSParc III 1.6GHz, GT/FF=Synthesized equivalent Gate Count and D-flip-flops, estimated by Xilinx ISE 9.2 on Xilinx Spartan III-1000, CLK=max. clock frequency estimated by ISE 9.2).**

Example **7** shows a simple function calculating the parity of a bit vector implemented with a for-loop. Table **9** shows synthesis results obtained by the ConPro compiler and finally using gatelevel synthesis with a standard cell target technology (SXLIB [16] and Mentor Graphics Leonardo Spectrum). Different synthesis rule settings both concerning the scheduling strategy (or more precisely: optimization) and the for-loop synthesis rule (unrolled or not unrolled). Note: since the reference stack scheduler may only handle local objects, the function arguments (implemented with global registers) must be copied to local registers, though usually not required.

The default synthesis rules lead to highest computation time (196 TU=clock cycles), medium resource coverage, and suprisingly highest longest combinational path estimated by the gatelevel tool. Basicblock scheduling reduces the computation time (130 TU) without changing resource coverage and longest path. The unrolled loop requires much lesser computational time (67 TU), but nearly highest resource coverage. Using the reference stack and basicblock scheduler results in lowest computational time, lowest resource coverage, and path delay.

EXAMPLE 7: **ConPro source code for the parity calculator and synthesis evaluation under different constraints (scheduling parameters).**

```
1:  const WIDTH: value := 64;
2:  reg d: logic[WIDTH];
3:  function parity (x: logic[WIDTH])
4:           return (p: logic):
5:  begin
6:    reg pl: logic;
7:    reg xl: logic[WIDTH];
8:    xl ← x;
9:    pl ← 0;
10:   for i = 0 to WIDTH-1 do
11:   begin
12:     pl ← pl lxor xl[i];
13:   end <with BLOCK_PARAMETER>;
14:   p ← pl;
15: end <with SCHEDULE_PARAMETER>;
16:
17: process main:
18: begin
19:   reg p: logic;
20:   d ← 0x12345670;
21:   p ← parity(d);
22: end;
```

Block Parameter	Time, Gates, Register, Path Delay
schedule=default	196 TU, 972, 82 , 5.7 ns
schedule=basicblock	130 TU, 937, 79, 5.3 ns
schedule=default& unroll	67 TU, 1525, 138, 4.1 ns
schedule=refstack& unroll	3 TU, 879, 69, 3.4 ns
schedule=basicblock& unroll	65 TU, 1538, 137, 4.1 ns
schedule=refstack, basicblock & unroll	2 TU, 853, 66, 3.4 ns

TABLE 9: **Synthesis results of parity calculator with different for-loop block and schedule parameters. Shown are required time units, gates, and registers for the function implementation using standard cell gatelevel synthesis. The path delay, gate & register count are calculated by the gatelevel synthesis tool, the time values by the ConPro synthesis tool.**

7. CONCLUSION AND OUTLOOK

In this paper, a new high-level language and a synthesis compiler ConPro used for circuit 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 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 a FSM and RTL on hardware behaviour level with good performance and resource coverage, using a 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. Though no iterative scheduling and allocation is performed, the optimizer can reach well optimized circuit designs. Main application fields are reactive systems, rather functional and pipelined systems. High-level-synthesis performed by the ConPro tool is faster than gate-level synthesis of about one order.

In the future, pipelining of the data path must be supported to provide high performance synthesis of functional units.

Actually the rule set χ is static and built in the synthesis tool. In the future, this rule set should be separated from the compiler and be extendable using a rule definition language. This approach enables dynamic modification of rules as part of the design process (having different rules for different designs).

REFERENCES

[1] Richard Sharp
Higher-Level Hardware Synthesis
Springer, 1998

[2] C. A. R. Hoare
Communicating Sequential Processes
Prentice Hall, 1985

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

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

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

[6] 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

[7] Steven M. Rubini
Computer Aids For VLSI Design
Addision Wesley 1987

[8] M. Eich, F. Grimminger, S. Bosse, D. Spenneberg, F. Kirchner
ASGUARD: A Hybrid Legged Wheel Security and SAR-Robot Using Bio-Inspired Locomotion for Rough Terrain.
In IARP/EURON Workshop on Robotics for Risky Interventions and Environmental Surveillance, Benicassim, Spain, January 7-8/2008.

[9] 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

[10] 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

[11] ESMA ALAER, ALI TANGEL, MEHMET YAKUT
MIB-16: FPGA Based Design and Implementation of a 16-Bit Microprocessor for Educational Use
WSEAS TRANSACTIONS on ADVANCES in ENGINEERING EDUCATION, Issue 5, Volume 5, May 2008

[12] 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

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

[14] Greg Andrews
Multithreaded, Parallel, and Distributed Programming
Addison Wesley, 2000

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

[16] Laboratoire d'Informatique de Paris 6
Alliance 5.0 VLSI CAD System
http://www-asim.lip6.fr/recherche/alliance, 2002