Hardware-Software-Co-Design of Parallel and Distributed Systems Using a Behavioural Programming and Multi-Process Model with High-Level Synthesis

Stefan Bosse[1,2]

University of Bremen, Department Computer Science, Workgroup Robotics, Germany[1], ISIS Sensorial Materials Scientific Centre, Germany[2]

Abstract

A new design methodology for parallel and distributed embedded systems is presented using the behavioural hardware compiler ConPro providing an imperative programming model based on concurrently communicating sequential processes (CSP) with an extensive set of interprocess-communication primitives and guarded atomic actions. The programming language and the compiler-based synthesis process enables the design of constrained power- and resource-aware embedded systems with pure Register-Transfer-Logic (RTL) efficiently mapped to FPGA and ASIC technologies. Concurrency is modelled explicitly on control- and datapath level. Additionally, concurrency on data-path level can be automatically explored and optimized by different schedulers.

The CSP programming model can be synthesized to hardware (SoC) and software (C,ML) models and targets. A common source for both hardware and software implementation with identical functional behaviour is used.

Processes and objects of the entire design can be distributed on different hardware and software platforms, for example, several FPGA components and software executed on several microprocessors, providing a parallel and distributed system. Intersystem-, interprocess-, and object communication is automatically implemented with serial links, not visible on programming level.

The presented design methodology has the benefit of high modularity, freedom of choice of target technologies, and system architecture. Algorithms can be well matched to and distributed on different suitable execution platforms and implementation technologies, using a unique programming model, providing a balance of concurrency and resource complexity.

An extended case study of a communication protocol used in high-density sensor-actuator networks should demonstrate and compare the design of a hardware and software target. The communication protocol is suited for high-density intra- and interchip networks.

Keywords

Cyber Physical Systems, System-on-Chip design, Synthesis, Digital Logic, High-Level Synthesis, ASIC and FPGA technology, Communication, Network Protocols,

Parallel systems, Parallel computing, Distributed Systems

1. Introduction and Overview

Embedded systems used for control, for example in Cyber-Physical-Systems (CPS), perform the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource-constrained manner. System-On-Chip designs are preferred for high miniaturization and low-power applications. Traditionally, program-controlled multi-processor architectures are used to provide the execution platform, but application-specific digital logic gains more importance.

There are two different ways to model and implement System-on-Chip-Designs (SoC) used in those embedded systems: using 1. a structural and/or 2. a behavioural level. The structural level decomposes a SoC into independent submodules - processor cores (or data processing units in general), memories, and peripherials - 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 without detailed assumptions about system architecture, generally a more sophisticated modelling level. In the context of CPS, these are mainly reactive systems with dominant and complex control paths. The major contribution to concurrency appears on control path level which can be explicitly modelled on algorithmic level.

A new SoC-design methodology is presented using the behavioural hardware compiler ConPro providing an imperative programming model based on concurrently communicating sequential processes (CSP) [7] and guarded atomic actions [4] with an extensive set of interprocess-communication primitives. The programming language and the compiler-based synthesis flow enables the design of application-specific constrained power- and resource-aware embedded systems on Register-Transfer-Level efficiently mapped to FPGA and ASIC technologies. Concurrency is modelled explicitly on control- and data-path level. Additionally, concurrency on data- path level can be automatically explored and optimized by different schedulers. Hardware blocks (including IPC and externally modelled) can be accessed transparently from programming level with a generic object-orientated approach.

The CSP programming model can be synthesized to different other levels, not only used for hardware circuit synthesis: software models (C, ML), intermediate μCode, RT state level, and finally to hardware behaviour level, e.g., VHDL. A common source for both hardware and software implementation with identical functional behaviour matches different embedded architecture levels and enables code reuse. The metalanguage ML (OCaML) is well suited for simulation and test-pattern based functional model checking.

Why a new language? Traditional programming languages like C are designed for sequential programming only, and concurrency is present to some extent through the use of libraries [1]. Concurrency should be controlled by first-class language constructs [3] to enable optimized design of massive parallel systems and hardware synthesis. There are several examples of new designed languages for concurrent programming, like SystemJ [1] or X10 [3]. C-like languages used for hardware-syn-

thesis are wide spread, but are not fully suitable for RTL synthesis due to strong dependency on memory model (pointers) and the missing concurrency model.

What is novel compared with other high-level-synthesis approaches? One language targets both concurrent software and hardware programming, the hardware synthesis process can be fine grained controlled on programming level using parameterized blocks. A traditional compiler approach with μCode intermediate representation (without loss of concurrency) enables fast and optimized synthesis. Object-orientated access of hardware blocks using the External Module Interface (EMI) - part of the programming model - provides a modern and transparent interface for both software and hardware designers, closing the gap between software and hardware models. The extended set of IPC primitives enables concurrent programming of complex control and data processing systems.

Processes and objects of the entire design can be distributed on different hardware and software platforms, for example, several FPGA components and software executed on several microprocessors, providing a parallel and distributed system. Inter-system-, interprocess-, and object communication is automatically implemented with serial links, not visible on programming level.

2. Design of Parallel Systems Using a Behavioural Model Approach and High-Level Synthesis

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

The **structural level** decomposes a SoC into independent submodules interacting with each other using centralized or distributed networks and communication protocols, mainly program-controlled multi-processor architectures.

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

The ConPro high-level synthesis of SoC designs uses a behavioural imperative programming language with a compiler-based synthesis approach from algorithmic programming level to register-transfer level mappable directly to digital logic **[2]**.

Concurrency is modelled explicitly on control path level with processes executing a set of instructions sequentially, initially independent of any other process. Interprocess-communication (IPC) provides synchronization with different objects (mutex, semaphore, event, timer) and data exchange between processes using queues or channels, based on the **Communicating Sequential Processes** (CSP, Hoare 1985) model.

There are local and global resources (storage, IPC) , accessed by one process and several processes, respectively. Concurrent access of global resources is automatically guarded by a mutex scheduler, serializing access, and providing atomic ac-

cess without conflicts.

There are process and top-level instructions. Top-level instructions are evaluated during synthesis (configuration). Process instructions are transformed and mapped to states of a clock-synchronous finite-state-machine (FSM) controlling the process RTL data path temporally and spatially, shown in figure **1**.

More fine-grained concurrency is provided on data path level using bounded blocks executing several instructions (only data path, e.g., data assignments) in one time unit. Block level parallelism can be enabled explicitly or implicitly explored by a basicblock scheduler **[2]**.

The complete synthesis process can be fine-grained parameterized on programming block level, for example selection of different expression models (allocation) or activation of specific schedulers and optimizers.

Figure 1. Mapping of the proposed multi-process model to the multi-FSM RTL architecture using high-level synthesis.
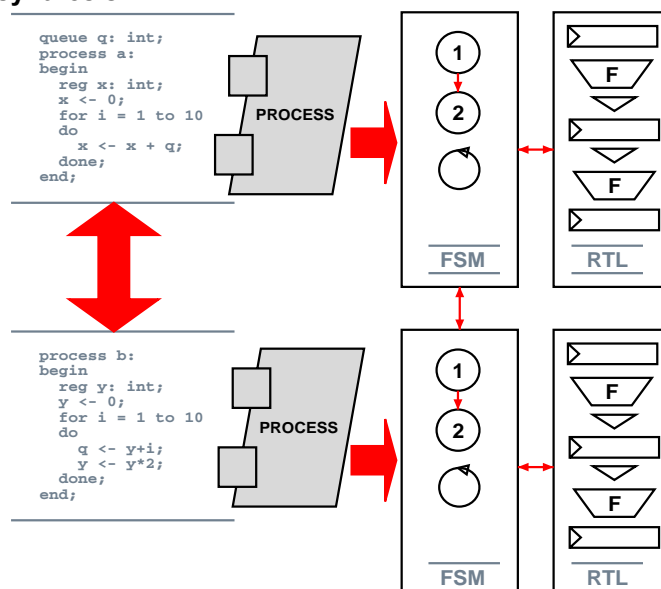
Hardware blocks, modelled on hardware level (VHDL), can be accessed from the programming level using an object-orientated programming approach with methods. All hardware blocks, including IPC, are treated like abstract data type objects (ADTO) with a defined set of methods accessible on process level and top level (only applicable with configuration methods, for example setting the time interval of a timer). The bridge between the hardware and software model is provided by the External Module Interface (EMI).

The relationship of the proposed programming and execution model and the required building blocks of Conpro (programming language and synthesis) are illustrated in figure **2**.

The programming language supports different types of storage objects (single registers and variables in shared RAM blocks, true bit-scaled), different aggregation types (array, structure), and abstract objects. Programming statements can modify data (expressions, assignments) or have impact on the control flow (conditional and counting loops, conditional branches, concurrent multi-value selection).

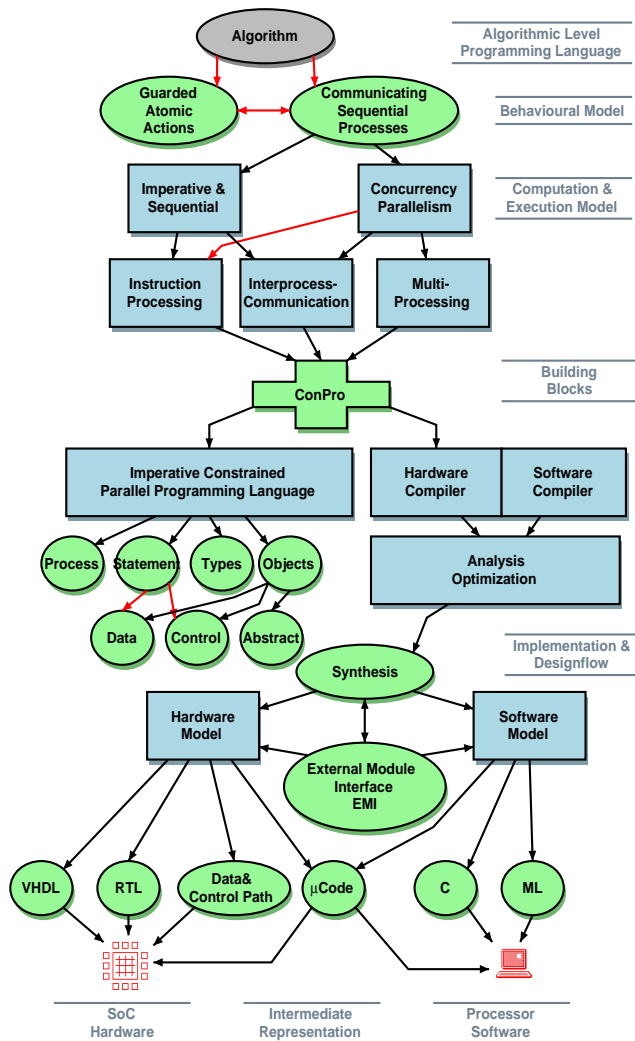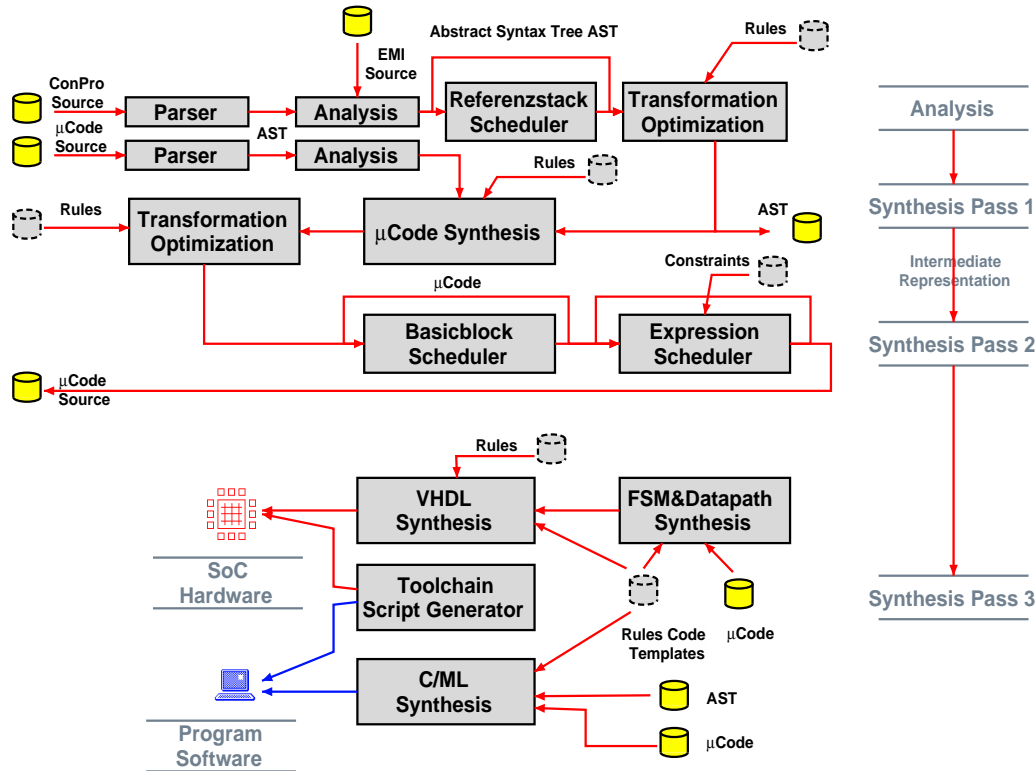**Figure 2. Building blocks: from the programming model to hardware using high-level synthesis.**

Figure **3** gives an overview of the design flow guiding through different levels provided by the ConPro framework. After the source code is parsed and transformed into an abstract syntax tree (AST), there are different allocation, scheduling, and optimization stages. The reference stack scheduler performs symbolic analysis on AST level and resolves constant and storage propagation, conditional assignments and multiple assignments. This ALAP scheduler has impact on scheduling and allocation done by optimization. The intermediate μCode representation was choosen for simplified RTL synthesis and optimization (synthesis pass I).

The basicblock scheduler partitions the program code into blocks without control side entries containing only data assignments (basicblocks). For each basicblock, a data-dependency analysis is performed. Independent data assignments can be bound to the same time unit. These optimizing schedulers can be activated or deactivated on block level. Finally, in synthesis pass III the RTL is synthesized and mapped to VHDL. Alternatively, after pass I (AST) or II (μCode), software output with

same functional and simulated/scheduled concurrency behaviour can be compiled.

Figure 3. Design flow using the high-level synthesis framework ConPro provides mapping of a parallel programming model to SoC-RTL hardware and software targets.

The synthesis flow

$$\chi \; : \; CP \rightarrow AST \rightarrow \begin{cases} \mu CODE \rightarrow \begin{cases} RTL \rightarrow VHDL \\ C/ML \end{cases} \\ C/ML \end{cases} \qquad \textbf{(1)}$$

is defined by a set of rules χ. Each set consists of subsets which can be selected by parameter settings (for example scheduling like loop unrolling, or different allocation rules) on block level.

Example **1** shows a part of the program code for the implementation of the dining philosopher problem. This system consists of five processes concurrently executing and implementing the action of the philosophers. They are defined using an array construct (line 13). The instructions of each process are processed sequentially, indicated by a semicolon after each instruction statement. The resource management of the forks is done with semaphores (abstract object type). Each `philosopher` process tries to allocate two forks, the left- and right-hand side forks, by calling the `down` method for each semaphore. If a philosopher process succeeds, it calls the

(inlined) function `eat`, and sets global registers (`eating`, `thinking`) simultaneously, indicated in the program code by using a colon instead of a semicolon (bounded instruction block). An event object `ev` is used to synchronize the startup of the group. The philosopher proceesses waiting for the event by calling the `await` method (line 15). The event is woken up by the main process calling the `wakeup` method (line 41). All processes are started from the `main` process (line 40). Processes are treated as abstract objects, too, providing a set of methods controlling the process state.

Example 1. Parts of a ConPro source code example: the dining philosopher problem implementation mapped to processes using semaphores for resource management.

```
1   open Core; open Process; open Semaphore; open System; open Event;
2   object ev: event;
3   array eating,thinking: reg[5] of logic;
4   export eating,thinking;
5   array fork: object semaphore[5] with
6         Semaphore.depth=8 and Semaphore.scheduler="fifo";
7   function eat(n):
8   begin
9     eating.[n] <- 1,thinking.[n] <- 0;
10    wait for 5;
11    eating.[n] <- 0,thinking.[n] <- 1;
12  end with inline;
13  array philosopher: process[5] of
14  begin
15    ev.await ();
16    if # < 4 then  -- all processes with array index lower 4
17    begin
18     always do
19     begin
20       -- get left fork then right
21       fork.[#].down (); fork.[#+1].down ();
22       eat (#);
23       fork.[#].up (); fork.[#+1].up ();
24     end;
25    end
26    else
27    begin
28     always do
29     begin
30       -- get right fork then left
31       fork.[4].down (); fork.[0].down ();
32       eat (#);
33       fork.[4].up (); fork.[0].up ();
34     end;
35    end;
36  end;
37  process main:
38  begin
39    for i = 0 to 4 do
```

```
40    philosopher.[i].start ();
41    ev.wakeup ();
42 end with schedule="basicblock";
```

Objects (like IPC) belong to a module, which have to be opened first (line 1). Each module is defined by a set of EMI implementation files providing all necessary informations about objects of this module (like method declarations, object access, and implementation on hardware and software level).

Figure 4. Process and interprocess-communication architecture of dining philosopher problem implementation from example source code 1.
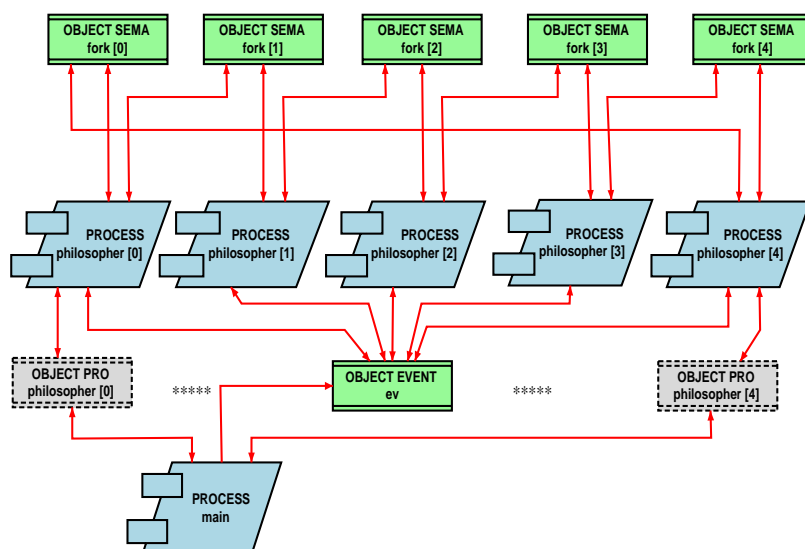
3. Transition from Parallel to Distributed System Design

Initially, there was the demand and the requierement to synthesize complex hardware designs from algorithmic level using high-level synthesis. This was supported by the Conpro[1] compiler. To reuse the same programming source for hardware-optimized and software designs, the high-level synthesis approach was extended to hardware-software synthesis in the ConPro[2] compiler using the External Module Interface hiding hardware dependencies and additonal software generators. Finally there is a demand for the design of distributed systems to enable the implementation of very complex systems.

In the proposed architecture and synthesis framework, objects and processes can be assigned to domains. Each domain is independently synthesized by the ConPro compiler either producing a hardware or software implementation. Domain management is performed separately by a partitioner module part of the Silicium-Compiler-and-Analyzer (SiCA) framework, shown in figure **5** (under development). There is only one programming source containing all objects and processes distributed over domains. The partitioner is responsible to create an appropiate communication architecture around the design, explained in the next section. The partitioner creates for each domain a separate design passed to the ConPro compiler, which synthe-

sizes either a hardware or software system target.

Figure **6** shows a possible partitioning of the dining philospher problem. Each domain contains one process and one semaphore object. The domain nodes are arranged in a two-dimensional mesh-network.

Figure 5. Design of parallel and distributed systems using the ConPro and the partitioner from the SiCA framework (under development). Domains are managed by the partitioner.
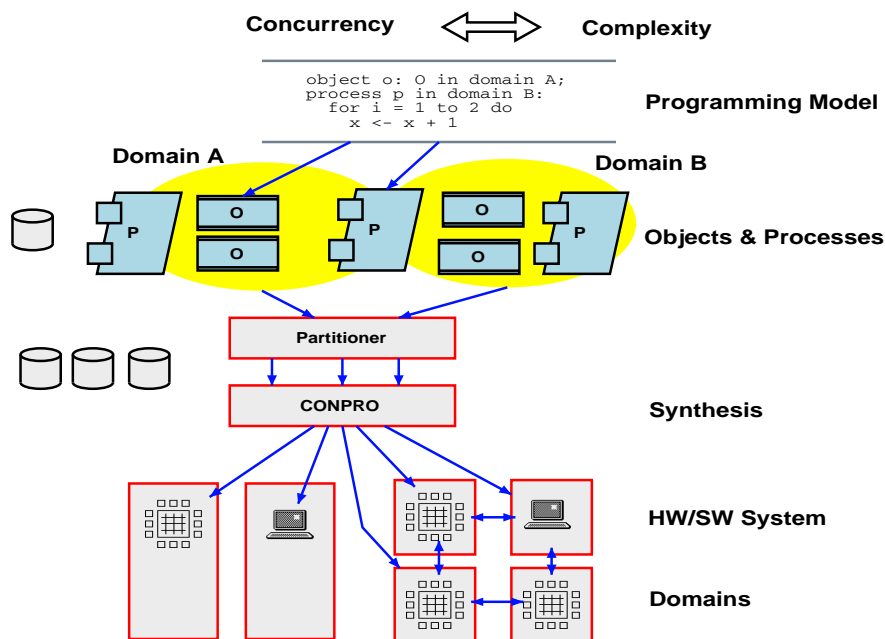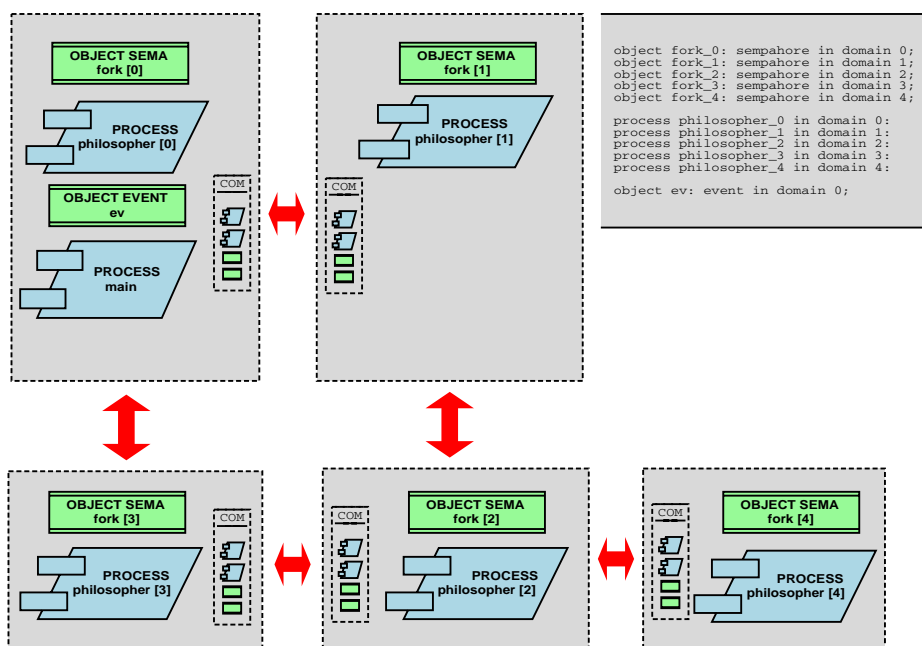

Figure 6. Domain partitioning of the dining philosopher problem implementation from example source code 1. Each domain is an independent system with additional communication blocks (COM).
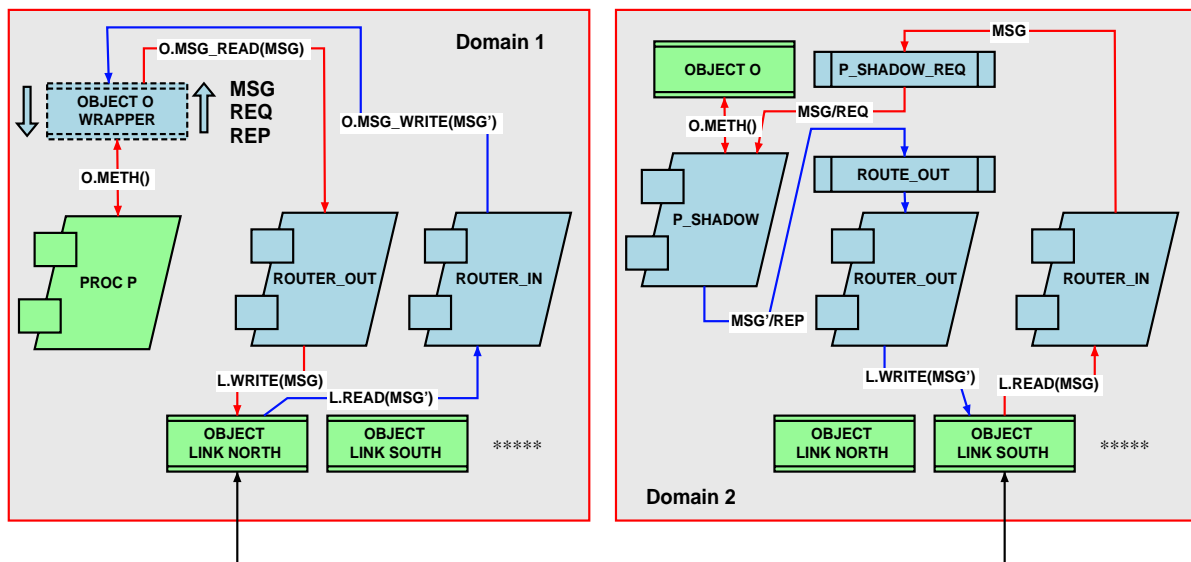
4. Communication Architecture and Links

Intra-chip/design communication is implemented using unidirectional signals for the hardware design target, and with functions and variables for the software target. Inter-chip/design communication is implemented with bidirectional asynchronous links using two-rail encoded serial data and a four-phase protocol for the hardware target. Message-based communication is used to access remote objects in different domains. A design domain is a network node, too. Network nodes are arranged in a two-dimensional mesh-grid. Each node has four different links: {North, South, East, West}.

Static routing based on process and object message identifiers is used to transmit a request or reply message from a source to a destination domain.

A method-based access of a process P in domain X of an object O implemented in domain Y is transformed into a method access of a local wrapper object O^W. This wrapper object (which can be accessed by several processes in the local domain) transforms the object access to a message request, read and processed by a local router (router_out), shown in figure **7**.

This router process selects an appropiate route to the destination domain Y, and sends the message to a serial asynchronous link connected to the next node in the network. If this node is the destination domain where the object O is implemented, the local router input processer (process router_in) passes the request message to a shadow process P_shadow performing the real method-based object access. After the method operations has finished, this shadow process creates a reply message passed to the router_out process, and is sent back from domain Y to X, finally received by the object wrapper.

Figure 7. Distributed system architecture: Method access of objects in foreign domains are translated into message-based communication with static routing.


An important part of the communication architecture proposed for distributed sys-

tems is the link used to connect different domains. Each domain has a local clock (assuming a locally synchronous system). The complete system consists therefore of different clock domains, too.

A Globally Asynchronous Locally Synchronous (GALS) system interconnect link architecture is required **[5]**. A serial asynchronous link is used for the proposed communication architecture, providing serialization and deserialization of data messages of arbitrary data size (bits). This link uses dual-rail encoding and a four-phase acknowledged protocol, based on the proposed architecture in **[6]**, shown in figure **8**. The advantage of dual-rail encoding is an asynchronous delay-insensitve interconnect.

The transceiver and sender were initially implemented with asynchonous logic using Muller-C gates. Those circuits can be well mapped to ASICs, but difficult to FPGA architectures (due to their synchronous system architecture). Instead the asynchronous circuit from **[6]** was reimplemented with synchronous logic using state machines and input signal oversampling methods, shown in figure **8**. The data stream contains empty and data packets. The empty packet is required for the last phase of the data transfer (return to zero completion).

Figure 8. Serial asynchronous communication link.
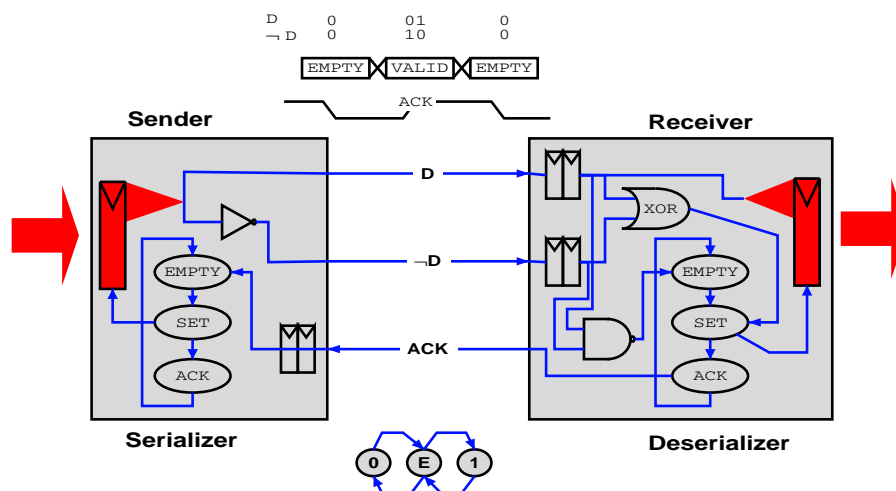
The message format used by the communication framework is shown in table **1**. Each message starts with a field indicating a request or reply type, followed by calling process, accessed object, and applied method identifiers. The data field is only required for methods transferring data (read or write operations). The set of domain-shared objects and processes accessing these objects is known by each node and

required for routing and delivering of messages.

Table 1. Communication protocol message format: upper rows: description, lower row: bit length.

Message type	Processes identifier	Object identifier	Method identifier	Optional method arguments
TYP	**PID**	**OID**	**MID**	**DATA**
1	$\log_2(P_N)$	$\log_2(P_O)$	$\log_2(M_N)$	MAX(argsizes)

The bit lengths of each field depends on the system partitionining and the number of objects accessed across domain boundaries O_N, the number of accessing processes P_N, and the number of methods used M_N.

There is a significant difference in object access latency time for the cases of local and remote access. Local object access requires (at least) two time units (clock cycles) if the resource is available and if the access can be immediately served. Remote access requires the passing of two messages (request and reply). Actually, the link module is implemented with a synchronous state machine and input signal oversampling. Assuming a transmission of one bit requires about ten time units (achieving 1 MBit/s with 10 MHz system clock), and a distributed system with 8 processes, 16 objects, 8 different methods, and a data size of 12 bits, then the overall message length is 23 bits, requiring 230 time units for each message transmission, with about 30 time units for message processing, in total 260 time units. Routing actually is performed with store-and-foreward buffering. Each routing attempt requires thus about 300 time units. Inter-domain communication is very expensive, and this is the reason why the domain partitioning is made by the programmer/designer.

5. An Extended Example: Implementation of a Protocol Stack for Robust Communication in Sensor Networks

The Simple Local Intranet Protocol (SLIP) **[8]** is used for communication in wired high-density sensor- and actuator networks. It implements smart routing of messages with Δ-addressing of nodes arranged in a n-dimensional network space (line, mesh, cube). The network can be heterogeneous regarding node size, computation power, and memory. The communication protocol is scalable regarding network topology and size.

A node is a network service endpoint and a router, too. The routing information is always kept in the packet, consisting of: 1.) a header descriptor (HDT) specifying the address size class ASC, the address dimension class ADC (for example 2 is a two-dimensional mesh-grid), 2.) a packet descriptor (PDT) with routing and path information, and finally the data part. SLIP was designed for low-resource System-On-Chip implementations using ASIC/FPGA target technologies, but a software version was required, too.

A node should handle several serial link connections and incoming packets concurrently, thus the protocol stack is a massiv parallel system, and was implemented

with the ConPro behavioural multi-process model.

The programming model implementation with partitioning of the protocol stack in multiple processes exeecuting concurrently and communicating using queues is shown in figure **9**.

Each link is serviced by two processes: a message decoder for incoming and an encoder for outgoing messages. A packet processor `pkt_process` applies a set of smart routing computation functions (`route_normal`, `route_opposite`, `route_backward`, applied in the given order until routing is possible), finding the best routing direction. Communication between processes is implemented with queues. There are three packet pools holding HDT, PDT and data parts of packets. They are implemented with arrays. The packet processor can be replicated to speed up processing of packets.

A test setup consisting of the routing processor part of SLIP was implemented A. in hardware (RTL-SoC, gate-level synthesis with mentor graphics leonardo spectrum and SXLIB standard cell library), and B. in software (SunOS, SunPro C compiler). A packet with ADC=2, Δ=(2,3) and a link setup of the node L=(-y,-x) is received on the second link (-x) [L01] and is processed first by the `route_normal` rule (would require connected +x /+y links) [L03], and finally by the `route_opposite` rule [L04] forwarding the modified packet to the `link_0` process [LA0].

Figure 9. Process and interprocess-communication architecture of the SLIP protocol stack.
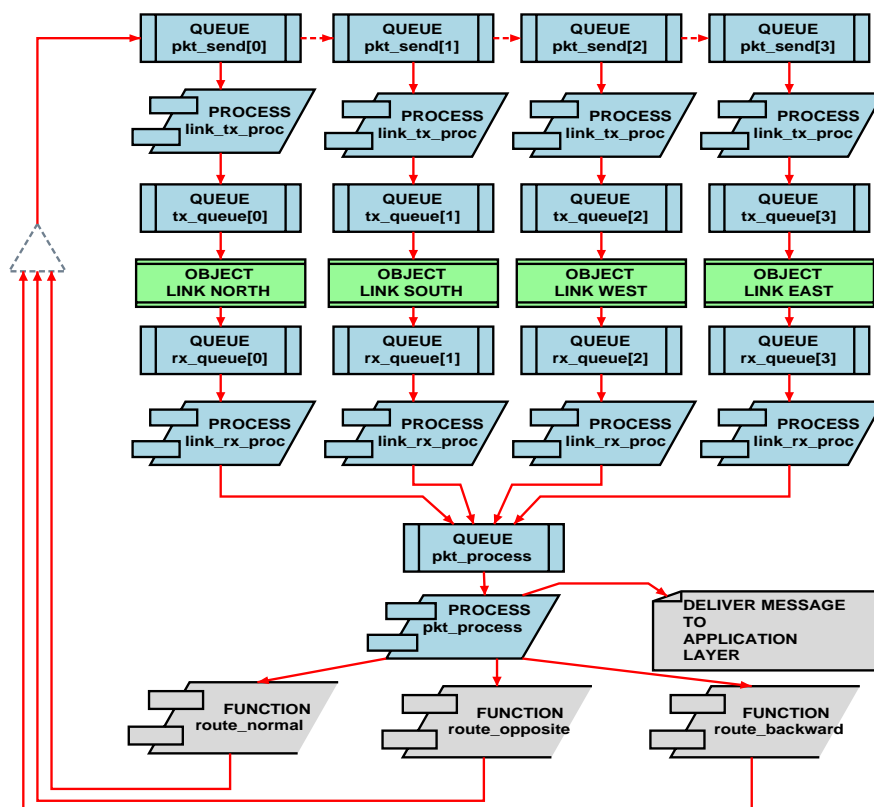
Tables **2** and **3** show synthesis and simulation results, of both hardware (HW) and software (SW) implementation. They show low resource demands and latency. Dif-

ferent checkpoints Lxx indicate the progress of packet processing. Figures in brackets give the latency progress relative to the previous checkpoint.

Table 2. Comparison of resources required for the HW implementation of routing part of SLIP implemented with a packet pool: (1) variable array, (2) register array. ASIC synthesis was performed with leonardo spectrum software and SXLIB standard cell library.

Ressource	Variable[1]	Register[2]
Registers [FF]	767	587
Area [gates]	12475	10758
Path delay [ns]	18	16
Synthesized Source CP $\rightarrow$ VHDL	1109 $\rightarrow$ 9200 lines	1109 $\rightarrow$ 7900 lines

From gate-level simulation, required clock cycles are obtained, and from software simulation with a debugger, required machine operations are obtained. The two HW implementations differ in packet pool architecture: 1. variable array in RAM blocks with EREW-access, and 2. register array with CREW-access, resulting in lower latency. The SW implementation contains built-in multi-processing, and requires up to 30 times more operations (time units) than the HW implementation.

Table 3. Simulation results of the HW and SW implementation of routing part of SLIP. HW: packet pool: (1) variable array, (2) register array, clock cycles. SW: SunPro CC, SunOS, USIII, CPU machine operations

Checkpoint	Clock Cycles[1]	Clock Cycles[2]	Machine Operations
L01	104	102	60000
L03	113 (δ=9)	107 (δ=5)	60019 (δ=19)
L04	187 (δ=74)	148 (δ=41)	60796 (δ=777)
LA0	235 (δ=48)	184 (δ=36)	62305 (δ=1509)

6. Summary

The ConPro programming language uses a concurrent multi-process model with interprocess-communication and guarded atomic actions, well suited to implement parallel control and data processing systems. Algorithms can be reused from traditional sequential programming. The ConPro synthesis tool is capable to implement complex algorithms, like communication protocols requiring concurrency on control path level, efficiently in hardware (below and beyond 1M gates), and software with same functional behaviour. Hardware blocks are accessed using a method-based object-oriented programming model.

Processes and objects of the entire design can be distributed on different hardware

and software platforms, for example, several FPGA components and software executed on several microprocessors, providing a parallel and distributed system. Inter-system-, interprocess-, and object communication is automatically implemented with serial links, not visible on programming level.

7. Bibliography

[1]   Malik, Avinash and Salcic, Zoran and Roop, Partha S., *SystemJ compilation using the tandem virtual machine approach*, ACM Trans. Des. Autom. Electron. Syst., Vol 14, (2009)

[2]   S. 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

[3]   Charles, Philippe et al., *X10: an object-oriented approach to non-uniform cluster computing*, OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (2005)

[4]   Daniel L. Rosenband and Arvind, *Modular Scheduling of Guarded Atomic Actions*, Proceedings of the 41st annual conference on Design automation (2004)

[5]   Paul Teehan, Mark Greenstreet, Guy Lemiex, *A Survey and Taxonomy of GALS Design Styles*, IEEE Design & Test of Computers, 2007

[6]   John Bainbridge, *CHAIN: A Delay-Insensitive Chip Area Interconnect*, IEEE Micro, 2002

[7]   C. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

[8]   S. Bosse, D. Lehmhus, *Smart Communication in a Wired Sensor- and Actuator-Network of a Modular Robot Actuator System Using a Hop-Protocol with Δ-Routing*, Smart Systems Integration, Como, Italy, 23-24.3.2010