Hardware Synthesis of Complex System-on-Chip Designs for Embedded Systems Using a Behavioural Programming and Multi-Process Model

Stefan Bosse(1,2)

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

Abstract

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. Application-specific System-On-Chip (SoC) designs providing the execution platform have advantages compared with traditionally used program-controlled multiprocessor architectures.

SoC designs can be modelled on structural and behavioural level. The behavioural level is 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.

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) 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 efficiently mapped to FPGA and ASIC technologies. Concurrency is modelled explicitly on control- and datapath level. Additionally, concurrency on datapath level can be explored and optimized automatically by different schedulers.

The CSP programming model can be synthesized to different levels, not only used for hardware circuit synthesis: software models (C, ML), intermediate μCode, RTL state level, and finally VHDL. A common source for both hardware and software implementation with identical functional behaviour is used.

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

Keywords

Cyber Physical Systems, System-on-Chip design, Synthesis, Digital Logic, Highlevel Synthesis, ASIC and FPGA technology, Communication, Network Protocols, Parallel systems, Parallel computing

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) [5] 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 datapath level. Additionally, concurrency on data path level can be explored and optimized automatically 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-synthesis are wide spread, but are not fully suitable for RTL synthesis due to strong dependency on memory model (pointers) and the missing concurrency

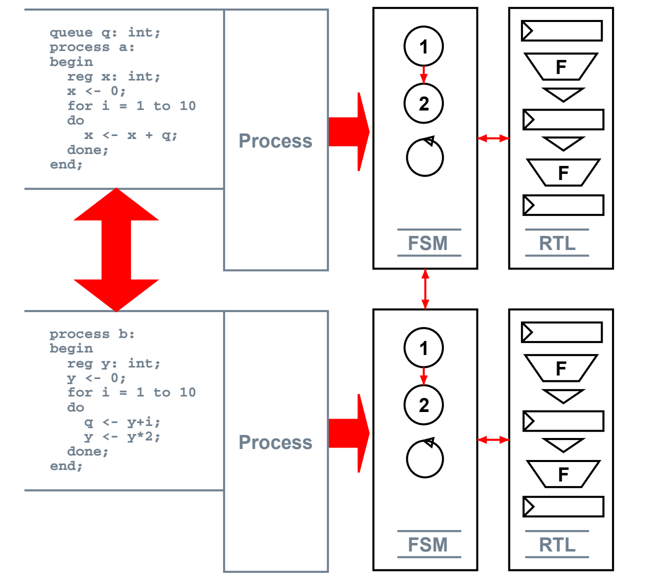
model.
What is novel compared with other high-level-synthesis approaches? One language targets both concurrent software and hardware programming, the hardware synthesis process can be fine-grained controlled on programming level using parameterized blocks. A traditional compiler approach with µCode intermeadiate 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.

2. System-On-Chip Design 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 multiprocessor 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, respective. Concurrent access of global resources is automatically guarded by a mutex scheduler, serializing access, and providing atomic access 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 basic-

block 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 FSM-RTL architecture using high-level synthesis.

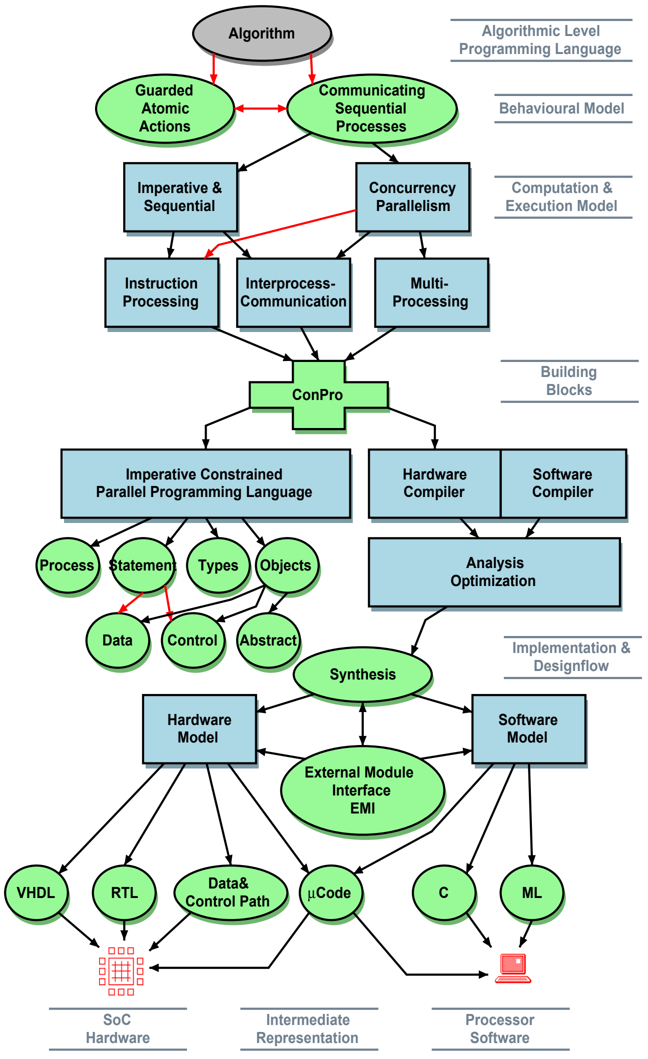


```
queue q: int;
process a:
begin
  reg x: int;
  x <- 0;
  for i = 1 to 10
  do
    x <- x + q;
  done;
end;
```

```
process b:
begin
  reg y: int;
  y <- 0;
  for i = 1 to 10
  do
    q <- y+i;
    y <- y*2;
  done;
end;
```

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**.

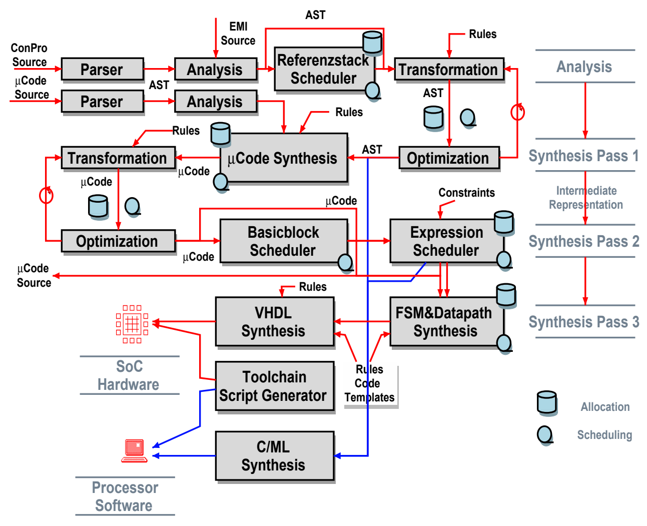
Figure 2. Building blocks: from the programming model to hardware using high-level synthesis.



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 **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. SoC design flow using the high-level synthesis framework ConPro providing mapping of a parallel programming model to RTL hardware and alternatively to software.



The synthesis flow

Equation 1.

$$\chi \; : \; \mathrm{CP} \rightarrow \mathrm{AST} \rightarrow \mu\mathrm{CODE} \rightarrow \mathrm{RTL} \rightarrow \mathrm{VHDL}$$

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 concurrent computation system performing data modification by an array of four processes `sum[0..3]`. They access the global register `x`. Though the access of `x` is atomic and guarded, the expression in line 9 is it not, thus a mutual exclusion lock `m` is required. A master process `someother` controls the system and waits for completion of all `sum` processes using a semaphore. A timer `t` performs group synchronization (here just for fun). The synthesis is controlled on block level with different settings (loop unrolling in line 10, scheduling in line 11, object constraints in lines 2 & 3). Line 14 creates a bounded block for data assignments to registers `a` and `b` (using a colon instead of a semicolon).

Example 1. Parts of a ConPro source code example.

```
1  open Mutex; open Timer; open Process; ...
2  object m: mutex with scheduler="fifo";
3  object t: timer; t.time(1 millisec);
4  object s: semaphore;
5  reg x: int[12];
6  array sum: process[4] of begin
7    for i = 1 to 10 do begin
8      t.await ();
9      m.lock(); x <- x + 1; m.unlock ()
10   end with unroll=true; s.up ();
11 end with schedule="basicblock";
12 process someother: begin
13   reg a,b: int[10];
14   a <- x+1, b <- x-1; x <- a;
15   t.init (); t.start (); s.init(0);
16   for i = 0 to 3 do sum.[i].start();
```

```
  for i = 1 to 4 do s.down();
end;
```

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 level).

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

The Simple Local Intranet Protocol (SLIP) [6] 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 informations are 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 meshgrid), 2.) a packet descriptor (PDT) with routing and path informations, 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. 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 untill 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].

Tables **1** to **3** show synthesis and simulation results, of both hardware (HW) and software (SW) implementation. They show low resource demands and latency. Different checkpoints Lxx indicate the progress of packet processing. 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 1. HW implementation of routing part of SLIP [packet pool: variable array, ASIC leonardo+SXLIB]

Ressources	Checkpoint	Clock Cycles
Registers: 767 FF	L01	104
Area: 12475 gates	L03	113 (+9)
Path delay: 18 ns	L04	187 (+74)
Source: 1109 lines CP → 9200 lines VHDL	LA0	235 (+48)

Table 2. HW implementation of routing part of SLIP [packet pool: register array, ASIC leonardo+SXLIB]

Ressources	Checkpoint	Clock Cycles
Registers: 587 FF	L01	102
Area: 10758 gates	L03	107 (+5)
Path delay: 16 ns	L04	148 (+41)
Source: 1109 lines CP → 7900 lines VHDL	LA0	184 (+36)

Table 3. SW implementation of routing part of SLIP [packet pool: variable array, SunPro CC, SunOS, USIII, CS:Code-, DS/BSS:Data segments]

Ressources	Checkpoint	Machine Operations
BSS: 40980 bytes	L01	60000
DS: 4352 bytes	L03	60019 (+19)
CS: 49288 bytes	L04	60796 (+777)
Source: 1109 lines CP → 2667 lines C	LA0	62305 (+1509)

4. Summary

The ConPro programming language uses a concurrent multi-process model with inter-process-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-orientated programming model.

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

[6] 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