Mixed Abstraction Level Hardware Synthesis from SDL for Rapid Prototyping*

Oliver Bringmann
Wolfgang Rosenstiel

Department of Computer Engineering
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Annette Muth
Georg Färber

Laboratory for Process Control and
Real-Time Systems
Technische Universität München

Frank Slomka
Richard Hofmann

Department of Computer Architecture and
Performance Evaluation
Universität Erlangen-Nürnberg

Abstract

SDL is currently gaining interest as a system level specification language for HW/SW codesign. Automated synthesis of SDL in hardware so far had problems with its efficiency. The investigations on the resource usage of SDL-to-VHDL designs presented in this paper identify two key challenges: minimizing the overhead introduced by SDL process infrastructure, and choosing the appropriate synthesis method. This paper presents a framework for SDL hardware synthesis where VHDL code generation, high-level synthesis and RT-level synthesis are combined. A configurable run-time environment implements services like data handling and message passing in efficient, hand-coded library components, which take into account properties of the target architecture. For these components RT-level synthesis was found to be suitable. The behavior of each SDL process on the other hand is freely specified by the system designer. Depending on the type of application, i.e. complex data-oriented or control-oriented, either high-level synthesis, RT-level synthesis, or a combination of both can prove to be optimal.

1 Introduction

SDL has recently received increasing attention as a system level description language for embedded systems design. A system modeled in SDL can be analyzed, verified and simulated at system level. Non-functional requirements for the next synthesis steps – HW/SW partitioning, code generation and synthesis – can be specified using language extensions like the SDL* annotations presented in [10]. In this context, the generation of the software implementation (i.e. in C using RTOS-primitives) is already supported by commercial tools, and is becoming more and more widely used, while interface generation is still object of further research [6]. The focus of this paper is on the automated hardware implementation of SDL system specifications, which is an essential part of an integrated HW/SW codesign environment.An efficient system for the automated generation of hardware from SDL descriptions needs to address the following concerns:

Minimal restrictions on the SDL specification: While not all SDL constructs make sense to be implemented in hardware, it is important that as few constraints as possible are imposed on the system level description by the implementation.

Flexible adaptation to a target architecture: The hardware generation has to take into account and make use of the properties of the target architecture, i.e. RAMs and communication interfaces. In an efficient design process it must be possible to separate this from the application itself.

Minimal use of hardware resources: A straightforward implementation of SDL-processes in hardware, especially the process infrastructure like message queues, leads to prohibitively large designs (see Section 4). The resource usage can be minimized using optimized, reusable components and taking into account the properties of the application and the target architecture.

1.1 Related Work

SDL is used for hardware design in different application areas and with different purposes. The distinguishing mark of the various approaches is the implementation model used. All of them support only a subset of SDL.

A framework for the automated design of communication subsystems is presented in [9]. The target architecture for SDL-processes implemented in hardware is a specialized protocol automaton, where an ALU, ALU-interface and I/O-interface are predefined components. Only the execution and control unit are compiled from the SDL-model. This work is very specialized and efficient for communication protocols, but not easily transferable to other applications.

In contrast to this, there are several approaches implementing a more general server model. Here, each SDL process is realized in one VHDL entity and its behavior is translated directly to VHDL. The abstract communication between the processes has to be mapped to existing interfaces and protocols. The SDL-to-VHDL translator presented in [7] uses a textual implementation description to select functions from a library of channel and protocol descriptions. In [3], SDL-to-VHDL is embedded in the codesign environment COSMOS. An SDL description is translated to an intermediate format. During an interactive refinement process, the abstract channels of this model are replaced by protocols, communication units and interfaces from a library. A large subset of SDL is supported.

In all three approaches, flexible adaptation to the target architecture is only considered for inter-process communication. To our knowledge, results concerning synthesis and resource usage have not been published yet.

1.2 SDL Hardware Synthesis Framework

The SDL hardware synthesis framework presented in this paper addresses two steps in the design process: The automatic generation of a VHDL description from the SDL

* This cooperative work is supported by the DFG research program “Rapid Prototyping of Embedded Systems with Hard Time Constraints” under grant Ro 1030/4, Fa 109/11-2 and He 1408/4-2.

model, and the following RT-level or high-level synthesis of a netlist from the VHDL model. The key concept of our framework is the **configurable run-time environment** for the hardware implementation of the SDL system. The term *run-time system* is borrowed from software-based systems, for like a software run-time system it serves to isolate the application from the underlying target architecture, like i.e. the available communication hardware, availability of RAMs, etc. In the context of an SDL model, it is the behavior of each process that can be separated from the infrastructure of each process and from the inter-process communication. For these infrastructure functionalities, which often cause large overhead and high resource usage, specialized hand-coded library functions, which are optimized for the target architecture, are used. Another class of functions of the run-time system are data handling functions, which offer i.e. a design alternative between using registers or RAM.

In the next step, the synthesis of a netlist from the VHDL-description, different **synthesis methods** can be applied, of course with an impact on the generated VHDL code, and with different advantages. High-level synthesis can synthesize timing-free, algorithmic VHDL descriptions. During high-level synthesis, the sequential timing behavior is determined and resource sharing is performed automatically, which leads to efficient implementations for SDL processes with computation intensive behavior. If the VHDL-description on the other hand contains cycle-fixed timing information, it can be synthesized using RT-level synthesis. RT level synthesis cannot perform resource sharing automatically, but avoids a certain overhead introduced by high-level synthesis. It is very efficient for small, control-oriented or communication intensive SDL processes, and for the architecture dependent functions of the run-time system - all system parts with with little arithmetic computation. We offer the use of both synthesis methods in order to combine the strengths of both.

The different synthesis variants are discussed more deeply in Section 2. Our SDL hardware synthesis environment is presented in Section 3. The paper closes with experimental results and conclusions in Section 4 and 5.

2 Synthesis Methods

As mentioned above, our SDL synthesis approach can be divided into two steps. First, the generation of a VHDL model from the SDL* system specification, and second, a mixed abstraction level synthesis methodology for the generated VHDL model. In this section, the synthesis methodology and the reasons for using the mixed abstraction level synthesis approach are presented in more detail.

The processes of an SDL description are composed of one or more states. The state transition can contain simple inter-process communications, complex arithmetic calculations, or both. In both cases, it can be necessary to introduce multiple states at RT level for a single SDL transition, because functional and I/O resources are limited by the target architecture. E.g., the number of functional resources restricts the data processing complexity of a single RT state and the I/O resources restrict the signal width of external or inter-process data transfers. Hence, one important step within an SDL synthesis framework is the determination of the sequential timing behavior of the design. This can be done explicitly by generating a cycle-fixed VHDL model directly from the SDL description, or implicitly by applying a high-level synthesis system to a superstate VHDL model.

The former method requires the fixing of the sequential timing before synthesis. The generated cycle-fixed VHDL model can directly be synthesized by an RTL synthesis tool. This is recommendable, if each SDL transition can be directly transferred into a single state of the RT VHDL model or the SDL specification contains little arithmetic computation. Once the complexity of the SDL transitions increases, the optimization potential decreases due to the restrictions of the explicitly fixed schedule and the absent resource sharing capability at RT level.

The latter method makes use of a superstate VHDL model. The superstate VHDL model can immediately be generated from the SDL description, because each SDL transition can be directly transferred into one VHDL superstate. Thus, the sequential timing needs not to be fixed before synthesis, and the following high-level synthesis step can take advantage of the preserved optimization potential. Result of high-level synthesis is a VHDL description at RT level, with a fixed sequential timing and an optimized area performance product. Especially, resource sharing is performed automatically. Main problem is the inferior result quality in case of communication intensive specifications as shown in Section 4. This is caused by the quite direct transformation of the control structures of the input description to the RT controller during high-level synthesis, which leads to larger hardware than necessary in cases when a complete control structure could be scheduled within a single clock cycle allowing the generation of simpler hardware without explicit control.

SDL specifications are often composed of several communication intensive parts and several data processing parts. Therefore, it is obvious to take profit when combining the strengths of the previously discussed synthesis methods to a mixed abstraction level synthesis approach. In this approach, the computation intensive parts are passed to high-level synthesis and the communication intensive parts to RT synthesis. An important constraint is that the high-level part as well as the RT part of the SDL description can be described using a single VHDL description in order to allow the simulation of the entire VHDL model. This can be done by encapsulating the communication intensive parts into interface procedures or interface components. An interface procedure represents the interface protocol and can be inline-expanded in order to get a pure high-level description. In contrast to this, an interface component represents the protocol at RT or lower level of abstraction and can be instantiated using simple communication procedures. Because the interface protocol mainly depends on the target architecture, scalable and reusable interface components can be kept in a run-time library and have not to be integrated in the design before synthesis is completed. The advantage is, that the interface components have to be synthesized only once and can be optimized manually, while the synthesis time can be kept low. Thus, not only different processes but also process parts can be synthesized separately using RT level or high-level synthesis tools.

All in all, this approach provides a universal methodology for synthesizing general SDL specifications using solely

high-level synthesis, solely RT synthesis, or both. The possibility to apply exclusively high-level or RTL synthesis allows the handling of pure computation intensive or pure communication intensive designs, as well.

3 The SDL Hardware Synthesis Environment

3.1 Synthesis Flow

The framework shown in Figure 1 supports all three synthesis methods for implementing SDL processes in VHDL, as discussed in the previous section. This allows the designer or an optimization tool to test various implementation techniques for an application.

Depending on the communication structure of the SDL specification, the tool generates a VHDL netlist for the architecture of the entire hardware/software architecture (1). Defining the length of signal queues and the word width of the arithmetical unit of the SDL automaton is possible using language constructs defined in SDL* [10]. Additionally the hardware/software partitioning can be defined in SDL*.

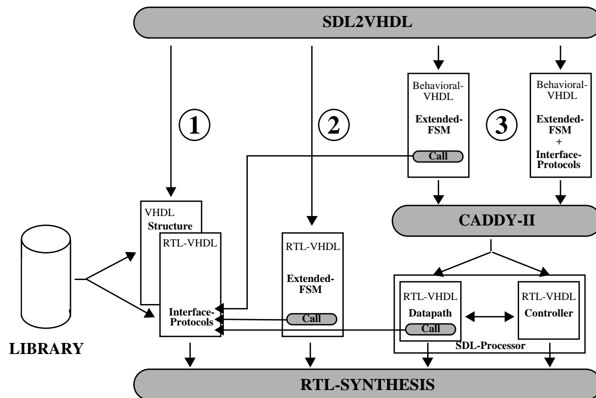

Figure 1: SDL Synthesis flow

To synthesize hardware descriptions of the behavior of the SDL processes, two alternative ways are supported by the SDL2VHDL tool:

- The tool generates VHDL code at register-transfer-level (2). In this case the behavioral description of the SDL process includes the cycle-fixed timing specifications needed for the hardware implementation. Such a VHDL description can be synthesized directly to the gate level by an RTL synthesis tool.
- In contrast to this approach the tool also allows to synthesize a VHDL description without any timing constraints (3). Then a high-level synthesis tool is needed, to calculate the sequential timing behavior of the resulting hardware architecture.

In our framework the high-level synthesis system CADDY-II ([1], [2]) is used. CADDY-II generates an application specific datapath with its own controller. After high-level synthesis, the VHDL description of the datapath and the controller is synthesized to a gate level description. The approach presented in this paper is not limited to CADDY-II, which was used as an example high-level synthesis tool.

To experiment with different approaches, it is possible to generate a VHDL description for CADDY-II together with inline-expanded communication procedures. On the other hand, CADDY-II can synthesize a VHDL description without any communication components and integrate the interface components taken after synthesis from the SDL run-time libraries. In the SDL run-time library, components like timers and communication mechanisms are predefined at RT level (see Section 3.4).

In data intensive applications, like i.e. from the telecommunication area, often very long arrays are used to describe the protocol messages. This leads to many registers in the datapath. The library approach for communication protocols is used additionally to connect dedicated memory components to the data path.

3.2 Structual Synthesis of the SDL System

The SDL2VHDL framework supports the synthesis of communication structures for the system. In SDL, communication means sending and receiving signals between processes. Each SDL process is represented by its own hardware module. The module contains the behavioral description of the SDL process, a hardware interface to the communication structure, the signal queue, and in some cases external memory modules for large data segments.

For the implementation of the communication structures itself, three different approaches are known:

- All processes are connected by a crossbar switch, which connects each process module with each other [9].
- The structure of the SDL specification is analyzed in order to connect only communicating processes [8]. This reduces the resource overhead needed to implement a full crossbar switch.
- The cheapest approach is to connect all processes by a bus system [3].

The SDL2VHDL framework supports a flexible way to select different communication interfaces by using inline-expandable functions to connect the behavioral description of one SDL process with predefined and pre-synthesized protocol interfaces. The selection of different interfaces is supported at the specification level by SDL*. In addition, all hardware components needed by software parts of the system are also integrated in the structural VHDL description of the design using the mapping constructs of SDL*.

3.3 Behavioral Synthesis of SDL Processes

As described in Section 3.1, the designer has two options to generate a VHDL description of the behavioral part of an SDL process. The first approach is to synthesize an RTL description for the hardware implementation. To perform this, we integrated in the tool the technique described by Glunz [7]. In such an RTL description, each state is described explicitly by a *wait until* construct. Thus, for each arithmetic operation a separate component and for each *wait until* a separate register is generated by the synthesis tool.

Figure 2 depictss an example SDL process specification. The states of the EFSM (extended finite state mashine) are *IDLE*, *SETUP* and *CONECT*[*]. The process defines a simple connection setup protocol. After receiving the signal *conReq* the process builds a message (3) to setup a connection and

[*]Because connect is an SDL keyword the name of the state is CONECT

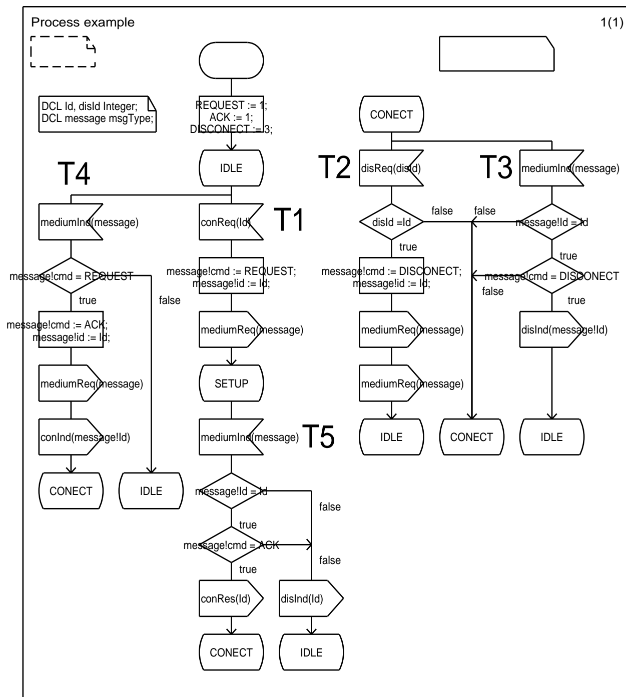

Figure 2: Example SDL specification

sends this message to another SDL process. This transition is labeled with T1. If this process sends a message with the command *ACK* the connection is established (T5). To disconnect the process may receive the signal *disReq (*T2) or a message with the command *DISCONECT* (T3).

Figure 3 shows a VHDL description of this SDL specification for the high-level synthesis system CADDY-II. The synthesis of VHDL code for CADDY-II includes many language features of SDL: *sending* and *receiving signals*, *enabling conditions*, *continuous signals*, *timer* and the *save* construct. It is possible to generate a VHDL model including or excluding communication functions. The SDL runtime components can be declared using a pragma construct, which allows the CADDY-II parser to include all signals needed by the library components to the port definition of the VHDL behavioral description. The components can be instantiated using the corresponding communication functions as illustrated in Figure 3. At the beginning of the process, all local variables are defined (variable definition). After the initialization phase of the system variables, the code enters an infinite *while loop*. In each iteration of the loop first the queue is evaluated. If the queue contains a valid signal the signal is received (reading queue). If the queue contains no signal a *continuous signal* is evaluated. Sending and receiving signals is supported by abstract VHDL communication functions (4) as discussed in Section 3.4. The SDL transition executed by the process is selected dependent on the actual state of the automaton. To support the *service* construct of SDL, all services of a process are evaluated in a round robin scheme.

In order to synthesize the generated VHDL description, the CADDY-II VHDL parser extracts for each communication function (e.g. *send* and *receive*) the sequential timing and selects the corresponding SDL component. This information is used by CADDY-II to integrate the SDL component in the synthesized RT datapath. In contrast to this, also the communication functions or the functional component description can be inline-expanded in order to apply high-level synthesis to the entire VHDL model. The different

```
-- Generated by SDL2VHDL Version 0.2pre
-- from source rsp99.sdl at 18-Mar-99 9:46:27 AM

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE work.sdlRunTime.all;
-- pragma DSL USE work.sdlRunTime.sdlRunTimeLib;

entity fsm is
  port
  (
    clk : in Std_Logic
  );
end fsm;

architecture fsm_process_example of fsm is
begin
  someProcess: process
    type Signed_Vector is array(Integer range <>) of Signed(7 downto 0);      variable
      variable dataIn : Std_Logic_Vector(7 downto 0);                         definition
      variable readyIn : Std_Logic;
      variable initPhase : Boolean;
      variable currentProcessOrServiceId : Integer;
      variable transitionPartId : Integer;
      variable state_1 : Integer;
      variable request_1_i : Signed(7 downto 0);
      variable ack_1_i : Signed(7 downto 0);
      variable disconect_1_i : Signed(7 downto 0);
      variable id_1_i : Signed(7 downto 0);
      variable disid_1_i : Signed(7 downto 0);
      variable message_1_i : Signed_Vector(0 to 1);
  begin
    initPhase := true; currentProcessOrServiceId := 1;                        initialisa-
    transitionPartId := 0; request_1_i := conv_signed(1, 8);                  tion
    ack_1_i := conv_signed(1, 8); disconect_1_i := conv_signed(3, 8);
    state_1 := 1;
    someLoop: while true loop
      case transitionPartId is
        when 0 => receive(id_i);                                              reading
          receive(readyIn, dataIn);                                           queue
          if readyIn = '1' then
            case dataIn (7 downto 6) is
              when „01" =>
                case state_1 is
                  when 1 => receive(id_1_i);                                  T1
                          message_1_i(0) := request_1_i;
                          message_1_i(1) := id_1_i;
                          send(„00000000", „0", message_1_i(0 to 1));
                          state_1 := 3;
                  when 2 | 3 => receive(REMOVE, 2);
                  when others => null;
                end case;
              when „10" =>
                case state_1 is
                  when 2 => receive(disid_1_i);
                          if disid_1_i = id_1_i then                          T2
                            message_1_i(0) := disconect_1_i;
                            message_1_i(1) := id_1_i;
                            send(„00000000", „0", message_1_i(0 to 1));
                            send(„00000000", „0", message_1_i(0 to 1));
                            state_1 := 1;
                          else transitionPartId := 1;     ---> to label
                          end if;
                  when 1 | 3 => receive(REMOVE, 2);
                  when others => null;
                end case;
              when „11" =>
                case state_1 is
                  when 2 => receive(message_1_i(0 to 1));                     T3
                          if message_1_i(1) = id_1_i then
                            if message_1_i(0) = disconect_1_i then
                              send(„10000000", „1", message_1_i);
                              state_1 := 1;
                            else transitionPartId := 2;
                            end if;
                          else transitionPartId := 2;
                          end if;
                  when 1 => receive(message_1_i(0 to 1));                     T4
                          if message_1_i(0) = request_1_i then
                            message_1_i(0) := ack_1_i ;
                            message_1_i(1) := id_1_i;
                            send(„00000000", „0", message_1_i(0 to 1));
                            send(„10000000", „1", message_1_i);
                            state_1 := 2;
                          else state_1 := 1;
                          end if;
                  when 3 => receive(message_1_i(0 to 1));                     T5
                          if message_1_i(1) = id_1_i then
                            if message_1_i(0) = ack_1_i then
                              send(„10000000", „1", id_1_i);
                              state_1 := 2;
                            else transitionPartId := 3;   ---> to label
                            end if;
                          else transitionPartId := 3;     ---> to label
                          end if;
                  when others => null;
                end case;
              when others => null;
            end case;
          end if;
        when 1 => state_1 := 2;                 <--- from jmp
                transitionPartId := 0;
        when 2 => transitionPartId := 1;        <---
        when 3 => send(„10000000", „1", id_1_i); <--- from jmp
                state_1 := 1;
                transitionPartId := 0;          ---> to label
        when others => null;
      end case;
    end loop someLoop;
  end process someProcess;
end fsm_process_example;
```

Figure 3: VHDL Description for CADDY-II (8-Bit)

synthesis methods can be chosen by use of VHDL attributes or command line options of the VHDL parser. So it is possible, to easily combine the advantages of high-level synthesis with the advantages of RTL synthesis.

To support a flexible SDL run-time system, the compound data types of SDL, e.g. structs, are not translated to the corresponding VHDL construct. Each *struct* data type is translated to a VHDL array. This technique leads to only two higher order data types in the VHDL behavioral description: Integer arrays and bit arrays. Using these arrays, it is possible to send or receive any signal and its parameters word by word. Another advantage of this approach is, that it is possible to move large arrays from the register set of the data path generated by CADDY-II to an additional external memory. This reduces the number of gates of the synthesized design.

3.4 Hardware Run-Time Library for SDL

To support the full communication mechanism of SDL we have defined a set of VHDL communication functions. The components accessed by the functions are described in [4]. The realization of the send and receive mechanism is very close to the mechanism described in [3]. Because we have mapped all higher data types to arrays, only a small set of communication procedures is used. The VHDL functions presented in Figure 4 are sending different commands, e.g. save, set timer etc., to the library components. This is a very flexible way because the support of different SDL mechanisms is controlled by allocating parameterized library components [4]. In the same way, it is possible to store large arrays outside the register set in external memories. This leads to efficient implementations on FPGAs, if fast static RAM is supported by the hardware.

To automatically select the VHDL function for the different cases of the send parameters, the overloading mechanism of VHDL is used. This is illustrated in Figure 4 only for several send functions: a function to send just one integer, a function to send an array of integers, a function to send an array of bits (for binary or string parameters). With all these functions it is possible to build three functions for the sending mechanism.

Each SDL signal contains a header with the type of the signal, the *Pid* of the send and the *Pid* of the receive process. Using this approach, it is possible to support SDL constructs like *sender*, *self* and *to*. Such a library also exists for the implementation of the send mechanism, the timer functions (*set, reset, now*), and for the access to external memories.

If the application contains large arrays these arrays are located in external memory. The run-time library connects the algorithmic description of the SDL automaton with this memory. If the standard memory's word length is incompatible with the word length of the synthesized datapath, the memory access function of the run-time library also adapts the memory corrected by using a parameterized shifter.

4 Experimental Results

To evaluate the resource usage of SDL implementations in hardware, first a very simple application ("ping-pong") was translated to VHDL using the SDL-to-VHDL translator described in [8]. This application contains a very simple

```
PROCESS
  TYPE Signed_Vector is array(Integer range <>) OF Signed(regWidth-1 downto 0);

  PROCEDURE sendHeader(header : in std_logic_vector; destOut : in std_logic_vector; cont : in std_logic) IS
  BEGIN
   WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '0';
   rtss.cmd <= SEND;
   rtss.dest <= destOut;
   rtss.snd <= header;
   rtss.changed <= '1';
   rtss.continue <= '1';
   WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '1';
   rtss.changed <= '0';
   rtss.continue <= cont;
  END sendHeader;

  PROCEDURE sendWord (data : in Signed_Vector; cont : in std_logic) IS
  BEGIN
   FOR i IN data'low TO data'high LOOP
    WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '0';
    rtss.snd <=  Std_Logic_Vector(data(i));
    rtss.changed <= '1';
    WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '1';
    rtss.changed <= '0';
   END LOOP;
   rtss.continue <= cont;
  END sendWord;

  PROCEDURE sendBit(data : std_logic_vector) IS
  BEGIN
    FOR i IN data'low TO data'length/regWidth LOOP
    WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '0';
    rtss.snd <=  data(i*regWidth to ((i+1)*regWidth)-1);
    rtss.changed <= '1';
    WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '1';
    rtss.changed <= '1';
   END LOOP;
    IF data'length-(data'length/regWidth) > 0 THEN
     WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '0';
   rtss.snd(data'length-(data'length/regWidth)-1 downto 0) <= data(data'length-(data'length/regWidth)-1 downto 0);
     rtss.changed <= '1';
     WAIT UNTIL clk'event AND clk = '1' AND rtss.busy = '1';
     rtss.changed <= '0';
   END IF;
   rtss.continue <= '0';
  END sendBit;

  PROCEDURE send(header : in std_logic_vector; dest : in std_logic_vector) IS
  BEGIN
   sendHeader(header, dest, '0');
  END send;

  PROCEDURE send(header : in std_logic_vector; dest : in std_logic_vector; wordData : in Signed_Vector) IS
  BEGIN
   sendHeader(header, dest, '1');
   sendWord(wordData, '0');
  END send;

  PROCEDURE send(header : in std_logic_vector; dest : in std_logic_vector; bitData : std_logic_vector) IS
  BEGIN
   sendHeader(header, dest, '1');
   sendBit(bitData);
  END send;

  PROCEDURE send(header : in std_logic_vector; dest : in std_logic_vector; wordData : in Signed_Vector; bitData :
std_logic_vector) IS
   BEGIN
   sendHeader(header, dest, '1');
   sendWord(wordData, '1');
   sendBit(bitData);
  END send;
 BEGIN
 END process;
END rtss;
```

Figure 4: Library functions for sending signals

communication protocol without any arithmetic computation. In the generated VHDL code, the cycle-fixed timing is a priori defined. It was synthesized using a commercial RT synthesis system (RT), fitted to Xilinx FPGA with XACT and tested on the configurable I/O-processor of the rapid prototyping environment REAR presented in [5]. The resulting CLB usages (RT estimation) for increasing message sizes and message queue lengths are shown in Figure 5. In the investigated range of values, the CLB usage increases linearly. For very small message sizes and queue lengths, the CLB count of this example is moderate, but increases greatly with the message size. A much higher impact however has the parameter queue length: The gradient of the CLB usage here is more than doubled compared with the parameter message size.

For the smallest design (message size 0, queue length 0), a comparison was made between a commercial RT synthesis system and a commercial high-level synthesis system (HL). The CLB-usage with high-level synthesis was by factor 9 larger than with RT-level (RT: 34 CLBs, HL: 311 CLBs). This illustrates the effect of synthesizing communication intensive, cycle-fixed RT VHDL descriptions without any arithmetic computation with a high-level synthesis tool.

As a more complex example, parts of a CAN controller

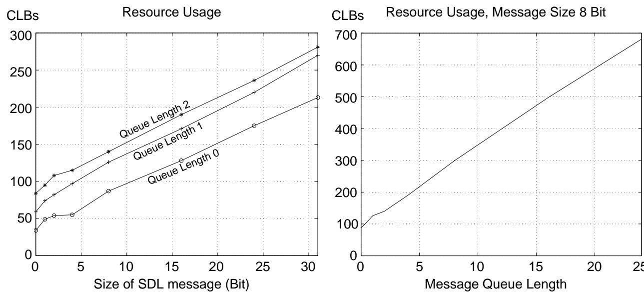

Figure 5: Resource usage of the “ping-pong” example

were implemented on REAR using the same design flow. In this SDL model, a process `serialization` receives a CAN message and outputs the single message-bits to a second process `CRC-generation`. This process computes the CRC checksum of the message using a simple iterative algorithm given in VHDL in a SDL task body. For CAN messages of the max 83 bit length, the design could not be fitted on the X4025E FPGA. The results for the minimum message length of 19 bit are shown in Table 1. For these experiments, a generic hand-coded HW/SW interface with time measuring capabilities was used. The results of both examples show that the message and especially the queue size have a strong impact on the resources needed to implement an SDL process. Even a process with nearly no behavioral part (ping-pong) can cause considerable overhead only due to the message handling implied by the semantics of SDL. The CRC example supports this: process `CRC generation` has a higher computational complexity than `serialization`, but handles only small messages (1 bit). Correspondingly, it has a much smaller CLB usage.

	flip-flops	F+G	CLBs
`serialization` part (pre PPR)	227	359	183
`CRC-generation` part (pre PPR)	53	105	55
incl. HW/SW interface (pre PPR)	347	707	419
after placing/routing	347	442	420

Table 1: Ressource usage CAN controller example

In addition to these experiments we have evaluated the difference of the RT synthesis and the high-level synthesis for SDL processes. To perform this, we added arithmetical data operations to the example SDL process shown in Figure 2. In this experiment, the synthesis was performed for the Altera Flex 10K100 family. The formula given in the SDL specification was only the addition of a few variables. The number of variables increases with the number of operations. The architecture generated for the RT-level synthesis

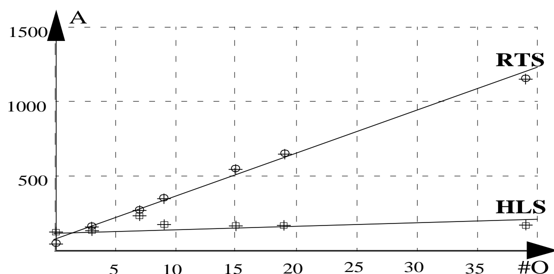

Figure 6: Area in dependency of the number of operations

uses the maximum number of resources to perform parallism. This leads to an implementation with minimal latency but maximal area. Figure 6 shows the comparison between the RT-level architecture and the controller synthesized by CADDY-II. If we only consider an SDL process with no arithmetical operations the RT-level VHDL code leads to a better implementation. After adding four integer variables within three additions, e.g. x = a + b + c + d the area needed by the controller generated by CADDY-II leads to a smaller implementation.

5 Conclusions

This paper presented a new approach for hardware synthesis of SDL system specifications and is embedded into our SDL HW/SW codesign framework. The approach based on a mixed-abstraction level synthesis approach in order to combine the advantages of high-level and RT synthesis. Communication intensive parts are synthesized by RT synthesis and computation intensive parts by high-level synthesis. Furthermore, the possibility to apply exclusively high-level synthesis or RT synthesis is still supported. Future work will consider the difference of RT synthesis and high-level synthesis for the SDL processes in more detail.

6 References

[1] P. Gutberlet, W. Rosenstiel: *Timing Preserving Interface Transformations for the Synthesis of Behavioral VHDL.* EURO-DAC, 1994.

[2] O. Bringmann, W. Rosenstiel: *Cross-Level Hierarchical High-Level Synthesis.* Design, Automation, and Test in Europe (D.A.T.E), 1998.

[3] J.M. Daveau, G.F. Marchioro, et. al.: *VHDL generation from SDL specifications.* XIII IFIP Conference on Computer Hardware Description Languages (CHDL ‘97), Toledo, Spain, 1997.

[4] M. Dörfel, F. Slomka, R. Hofmann. A Scalable Hardware Library for the Rapid Prototyping of SDL Specifications. 10th IEEE International Workshop on Rapid System Prototyping. 1999.

[5] F. Fischer, T. Kolloch, A. Muth, G. Färber: *A Configurable Target Architecture for Rapid Prototyping High Performance Control Systems.* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97), Las Vegas 1997.

[6] F. Fischer, A. Muth, G. Färber: *Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment.* 6th International Workshop on Hardware/Software Co-Design Codes/CASHE ’98, Seattle, USA, 1998.

[7] W. Glunz, T. Kruse, T. Rössel, D. Monjau: *Integrating SDL and VHDL for System-Level Hardware Design.* XI IFIP Conference on Computer Hardware Description Languages (CHDL ‘93), Ottawa, Canada, 1993.

[8] D. Reichelt: *Design and Implementation of a Tool for the Automatic VHDL Generation from a Annotated SDL System Description.* Diploma Thesis at Department of Computer Architecture and Peformance Evaluation, Universität Erlangen-Nürnberg, 1998 (in german).

[9] G. Carle, J. Schiller: S*emi-automated Design of High-Performance Communication Subsystems.* 31st IEEE Hawaii International Conference on System Sciences, HICSS 98, Kona, 1998.

[10] S. Spitz, F. Slomka, M. Dörfel: *SDL* - An Annotated Specification Language for Engineering Multimedia Communication Systems.* 6th Open Workshop On High Speed Networks, Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1997.