Towards Interprocess Communication and Interface Synthesis for a Heteogenous Real–Time Rapid Prototyping Environment*

Franz Fischer Annette Muth Georg Färber
Laboratory for Process Control and Real–Time Systems, Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{**Franz.Fischer,Annette.Muth,Georg.Färber**}**@lpr.e-technik.tu-muenchen.de**

Abstract

Rapid Prototyping has been proposed as a means to reduce development time and costs of real–time systems. Our approach uses a heterogeneous, tightly coupled multiprocessor system based on off–the–shelf components as target architecture for an executable prototype, which is generated from the specification in an automated design process. Here, too, we aim to use existing tools and languages. But interface and communication synthesis, while being the key requirement of an automated translation of a abstract specification to a distributed system, is not yet state–of–the–art. The sensitivity of the overall performance of multiprocessor systems to overhead and latency introduced by communication on the other hand calls for an efficient interprocess communication (IPC). This paper presents concept and implementation of IPC functions which, implementing the message queue semantics of the specification language SDL, links the standard components of our multiprocessor system in an efficient manner, while at the same time providing the interface synthesis needed by the automated generation of a rapid prototype. The experiences gained when implementing a non–trivial, real–world CAN controller and monitor application on our rapid prototyping environment, are described as a first proof of concept.

Keywords: communication synthesis, rapid prototyping, SDL, hard real–time, task classification model;

1. Introduction

The aim of rapid prototyping real–time applications is to substantially reduce development times by confirming the functional and timely requirements of the application at a very early stage of development with the help of an executable prototype.

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on “Rapid Prototyping for Embedded Hard Real–Time Systems” under Grant Fa 109/11-1.

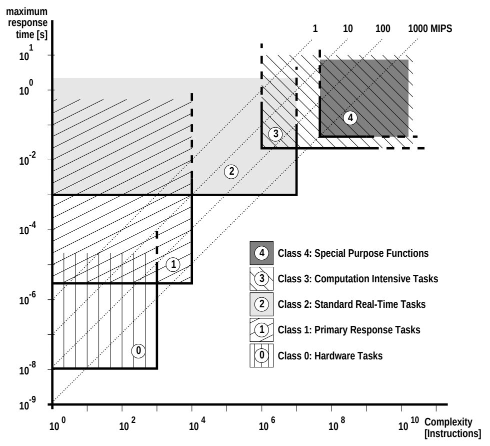

Figure 1. Classification of application tasks

When the real–time application has *hard deadlines*, the focus of the development lies not only on the functional correctness of the system, but also on the proof that it will meet all its deadlines. A real–time analysis that is able to deliver this proof needs the worst case execution times (WCETs) of the implementation.

Our rapid prototyping **target architecture REAR** (Rapid Prototyping Environment for Advanced Real-Time Systems) was designed to support real–time analysis in guaranteeing *realistic, not–too–pessimistic worst–case execution times*. The basis for this is the task classification model presented in [3], where each type of real–time task corresponds to a best suited processor type, in terms of performance and deterministic execution times. It is a configurable and scalable heterogeneous multiprocessor system consisting of standard off–the–shelf components, which are tightly coupled by a global PCI–Bus. The processing units' basic difference is the loss of predictable performance caused by interrupts and context switches:

High Performance Unit HPU (Tasks Class 3): based on standard computer architectures to benefit from technological advances, it is in our case a PCI slot CPU

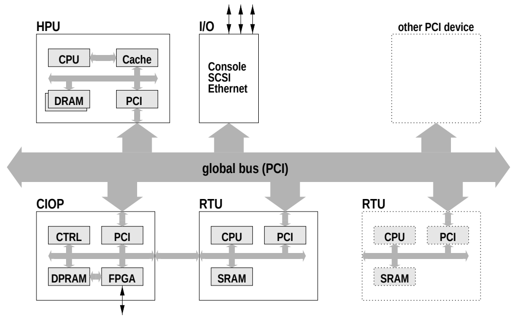

Figure 2. REAR hardware architecture

with Intel Pentium processor, large L2–cache and memory. The impact of interrupts and context switches on predictability is limited by software means.

Real–Time Unit RTU (Tasks Class 2,1): optimized for small tasks with short response times, which do not allow predictions of the cache behaviour. Instead, the much slower RAM–access times have to be used for the computation of WCETs. Our current RTU is a MIPS R4600 based single board computer with PCI–Interface and fast static RAM, running the freely available light–weight kernel RTEMS.

Configurable I/O Processor CIOP (Tasks Class 0): FPGA–based unit for tasks with deadlines too short to be met in software. The implementation in hardware allows a straightforward computation of the WCETs. In our rapid prototyping application, the CIOP consists of one Xilinx FPGA and additional Dual Ported RAM. It serves two purposes: It acts as a “event–filter” for events with short deadlines, and it links the rapid prototyping platform to the embedding process in a configurable manner.

Design Process To allow a schedulability analysis of the resulting generated system in addition to the automated transformation, our rapid prototyping design process requires both, the specification of the systems behaviour as well as a specification of the timing properties and constraints of the system’s embedding environment. The latter is done with the help of event streams, their associated maximum response time and event interdependences [5], while we use SDL for the functional specification. The specification is followed by the allocation of the SDL processes to the available processing units, according to the task classification model and based on the maximum response times and an estimated computational complexity. After that, high level language code is generated for the software and hardware parts and communication and HW/SW interfaces are synthesized, before the software parts are compiled and linked for the respective processing unit and the hardware parts synthesized for the FPGA–based CIOP(s).

Communication synthesis for distributed real–time systems has recently received considerable interest, e. g. [7, 1, 2]. While our approach does not claim the generality of these systems — system architecture, bus topology and protocol are known after all — we need to achieve an efficient solution for this particular environment and aim to reach the higher degree of automation required by a rapid prototyping system.

For code generation we decided to use existing tools: SDT’s “C advanced” code generator for the SW side (mapping one SDL process in our case to an RTEMS task) and the SDL–to–VHDL translator originally intended to support hardware design on a high abstraction level [4]. As both tools currently do not support mixed HW/SW systems, we developed a concept to integrate them to a synthesis system for our heterogeneous rapid prototyping target architecture REAR.

The following section describes this approach and the implementation of the underlying inter unit communication methods. Section 3 summarizes the results of a case study before the paper closes with conclusions and an outlook on future work.

2. An Approach to Interprocess Communication and Interface Synthesis

A SDL specification consists of a set of processes communicating via asynchronous *Signals* which optionally carry arbitrary data and *imported* or *viewed variables*. Processes are specified as extended finite state machines. A state transition is triggered and the according SDL statements executed by the reception of a signal (INPUT) or a boolean expression including imported/viewed variables (so called continuous signals) or both (signal with enabling condition). For the reception of signals, each process has a private FIFO input queue, where signals may be sent by other processes (OUTPUT).

2.1. SDT generated code

SDT supports code generation for the so called tight integration with a real–time operating system: A SDL process and its input queue are implemented as one RTOS thread and one RTOS FIFO message queue. INPUT and OUTPUT statements are translated to macro calls which finally expand to the RTOS’s message queue receive and send calls. Shared (exported/revealed) variables are mapped to (buffered) global variables.

As stated above SDT does not directly support mixed HW/SW systems: The usual way to handle HW parts is to use one or more external tasks to do all the dirty work and communicate with the SDL processes by sending/receiving “hand coded” SDL signal structures. However, this and the

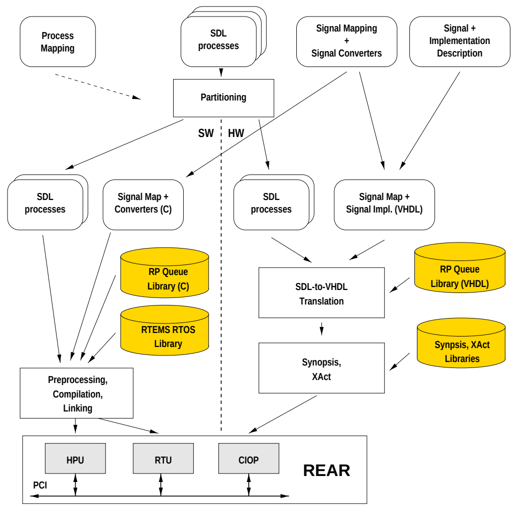

Figure 3. Integrating available code generators to a synthesis system

macro based code forms the path to an integrated HW/SW synthesis system (see below).

2.2. SDL–to–VHDL translator

The tool translates each SDL process to a VHDL entity based on configurable VHDL code fragments. For generating synthesizable VHDL code for asynchronous SDL signals, the translator needs as inputs a set of signal implementation descriptions (VHDL procedures) and a library of protocol descriptions (preprocessed to a VHDL package) in addition to the SDL processes to generate code for. The protocol descriptions can be used to connect external components or — as in our case — to implement the necessary interfaces (signal queues, registers, . . .) to the software side.

2.3. Integration

Figure 3 gives an overview of the integrated synthesis system. In addition to the set of application SDL processes it requires information about the process mapping (HW/SW partitioning), the mapping of signals at the HW/SW boundary to a particular HW/SW queue, an implementation of (optional) signal converter functions and the signal implementation and protocol descriptions required by the SDL–to–VHDL translator. The signal mapping information is necessary to generate for the software side

1. C macros to map local and remote SDL OUTPUT statements to RTOS calls or the library routines (and optionally signal converter) for inter unit communication, respectively,
2. an ISR to handle communication interrupts from remote processing units, and
3. an initialization procedure for all necessary inter unit queues.

For the hardware side the signal mapping information is used to configure the HW/SW queue entities while the other information is passed to the SDL–to–VHDL translator.

2.4. Inter–Unit Queue Implementation

The inter unit queue implementation is based on the tight coupling of the target architecture's processing units, i.e. that communicating units share a common memory region. On REAR this can be e.g. a memory region accessible through the PCI bus by another master capable processing unit or the CIOP's dual ported RAM. This common memory region is used to implement FIFO queues (for SDL signals) or for exported/revealed variables.

2.4.1 Software — Software

For communication between RTEMS threads on the RTU and Linux processes on the HPU, a simple module for message based IPC provides services to establish a communication *queue* and to send messages to and receive messages from the queue.The queue descriptor encapsulates information concerning the sender and receiver threads or processes and a pointer to a FIFO queue for the messages. The queues are located within the RTU's DRAM, which is accessible by the HPU via PCI bus. A queue consists of the *in* and *out* indices and, in the current implementation, a configurable fixed length message buffer array. A more flexible alternative that requires less copy operations, but needs one spinlock or semaphore for each thread, is a buffer pool with linked lists queue, which will replace the fixed arrays in the future.

The basic algorithm for receiving messages is as follows: `queue_getmsg()` checks whether a message is available and if so, copies the message from the queue's to the thread's message buffer and returns `TRUE`. A receiving thread will wait for a message to arrive and then notify the remote side of the receive queue being not full by triggering an interrupt. `queue_putmsg()` performs the same function vice versa. On the RTU, the queue ISR handles the notifications from the HPU side and propagates them as RTEMS *events* to any waiting thread, which in turn is unblocked and re–checks its receive or send queue. On the HPU, access to the queue memory area as well as triggering and handling notification interrupts is supported by a UNIX device driver.

2.4.2 Software — Hardware

A target architecture like REAR allows different implementation alternatives for FIFO queues for SDL signals:

1. for signals with no or only little data, short queues can be realized as FPGA internal FIFO, with the head (or tail) being accessible by a register;
2. larger FIFO queues make use of the CIOP's dual ported RAM and implement the *in* and *out* indices as well as additional control and status flags in a queue control and status register within the FPGA; (this alternative is very similar to the software queue implementation described in Section 2.4.1);
3. implementing only one (set of) SDL signal data register(s) is sufficient if either the interrupt latency on the software side is short enough to avoid loss of a signal, or signal data are transferred to a queue in main memory by bus mastering or direct memory access;

```
PROCEDURE send_canmsg (
    SIGNAL clock, req, abort, done:
                    IN     std_logic;
    SIGNAL data:    IN     std_logic_vector(15 DOWNTO 0);
    SIGNAL ack:     OUT    std_logic;
    SIGNAL base:    IN     std_logic_vector( 6 DOWNTO 0);
    SIGNAL inidx:   BUFFER std_logic_vector( 3 DOWNTO 0);
    SIGNAL offset:  BUFFER std_logic_vector( 2 DOWNTO 0);
) IS

BEGIN
    offset <= "000"; dwrite <= ’0’;
    send: LOOP
        WHILE NOT (req=’1’ OR abort=’1’ OR done=’1’) LOOP
            WAIT UNTIL clock’EVENT AND clock = ’1’;
        END LOOP;
        EXIT send WHEN abort=’1’;
        IF done = ’1’ THEN
            inidx <= inidx + "0001";
        END IF;
        EXIT send WHEN done=’1’;
        daddr(13 DOWNTO 7) <= base;
        daddr( 6 DOWNTO 3) <= inidx;
        daddr( 2 DOWNTO 0) <= offset;
        ddata <= data; dwrite <= ’1’;   -- write data
        WAIT UNTIL clock’EVENT AND clock = ’1’;
        dwrite <= ’0’; ack <= ’1’;      -- data written
        WHILE NOT req=’0’ LOOP
            WAIT UNTIL clock’EVENT AND clock = ’1’;
        END LOOP;
        ack <= ’0’;
        EXIT send WHEN reset=’1’;
        offset <= offset + "001";
    END LOOP;
END send_canmsg;
```

Figure 4. VHDL procedure to "send" a CAN message to a DPRAM queue

For the case study described in Section 3 the second alternative has been chosen to implement a transmit and a receive queue for CAN messages. Central elements on the HW side are the queue control and status register (QCSR) and the `send_canmsg()` and `receive_canmsg()` VHDL procedures (Fig. 4). The QCSR holds the two queue indices, a DPRAM base address, *not–full* and *not–empty* status flags and *clear*, *start* and *interrupt enable* control bits. The VHDL procedures were coded according to the protocol descriptions required later for the SDL–to–VHDL translator.

For the SW side, library routines for enqueuing / dequeuing messages (SDL signals) in / from DPRAM queues have been implemented. The dequeue routine checks the not-empty flag and if it is set, copies the message from the DPRAM address given in the QCSR and increments the *out* index and is typically called from the ISR. In contrast, the enqueue routine is called by the SDL processes sending a signal to the HW side.

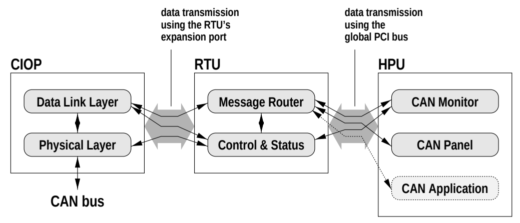

Figure 5. Task allocation and communication of the CAN application

3. Case Study: a CAN Monitor Application

As a test bed for the application to be implemented on REAR, we built a CAN bus environment consisting of two exemplary SLIO–based I/O–cards emulating sensors and actors and two commercial CAN participants with monitor and analyzing software. Details on the realized CAN environment can be found in [6].

From the user's point of view, the application performs two functions: The CAN bus monitor allows the user to send, receive and filter CAN messages, to monitor activity on the CAN bus. The SLIO controller provides an interface to the SLIO cards. The lower levels of the application have to provide the distribution of CAN messages to and from the CAN monitor and the SLIO controller (message routing) and all functions of a fully functional CAN bus participant [8]. The task classification model presented in 1 applied to the identified functions yielded the task allocation shown in Figure 5. The CAN application was modeled in SDL — except for the graphical user interfaces implemented on the HPU. The SDL process diagram is shown in Figure 6. All software processes were implemented as RTEMS–tasks using the described mechanism to include the appropriate IPC–calls respectively RTEMS library functions. As the SDL–to–VHDL translator was not available for this example, the hardware process (`can_controller`) was specified in Statemate and translated to VHDL using Statemate's VHDL–generator. Together with the VHDL–part of the HW–SW–interface, it was fitted to the FPGA.

The implemented system met all the requirements posed by the planned CAN controller and monitor application, in particular met all the deadlines of the CAN protocol up to the maximum 1 MHz without message loss.

For a first evaluation of IPC performance, the queue functions described above were instrumented to write time stamps to a memory buffer. The timing test application for the SW/SW communication included one Linux pro-

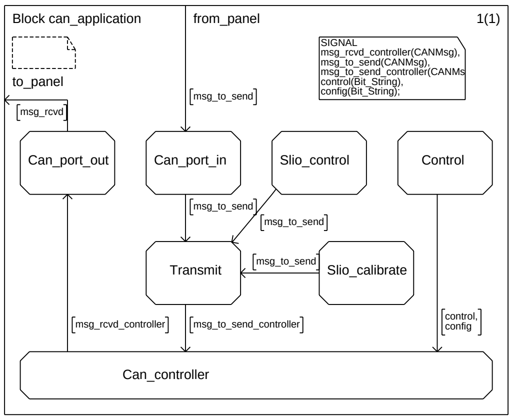

Figure 6. SDL process diagram of the CAN application

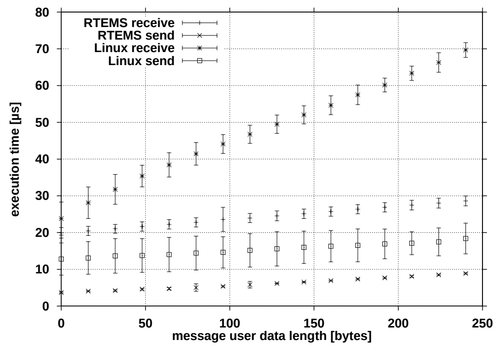

Figure 7. performance of the simple message queue implementation

cess to send a message to a RTEMS thread, which after being unblocked and receiving the message, sent the unmodified message back to the now blocked Linux process. I.e. the receive operations on both sides were blocking and involved processing an interrupt and a context switch, while the send could be performed without the sender being blocked. Fig. 7 shows the measured (average) execution times of the `queue_sndmsg()` and `queue_rcvmsg()` calls on RTEMS and Linux, respectively.

Due to the similar implementation, the send operation from SW to HW is as fast as `queue_sndmsg()` on RTEMS, i.e. below $5\ \mu$s. The opposite direction, however, involves the ISR overhead and the call of `rtems_message_queue_send()` and therefore takes approximately $20\ \mu$s.

4. Conclusions and Future Work

In this contribution we described concept and implementation of efficient IPC functions which are integrated in the automated design process for a rapid prototyping system. It is now possible to map a SDL specification of a real–time system to a heterogeneous multiprocessor platform without regarding the details of the underlying hardware. Latency and overhead caused by the communication are very low: The message queues between HPU and RTU and between RTU and CIOP introduce latencies in the below $20\ \mu$s–range. The next steps include integrating the hardware part of the HW/SW queue interface functions with the SDL–to–VHDL translator, and at the same time implementing the HW/SW interface alternatives outlined in Section 2 (DMA, internal FIFO). The degree of automation can be further increased by an automatic configuration of the message queues on the hardware side, which includes allocation of SDL–signals to message queues and defining their size depending on the data types to be transmitted.

References

[1] I. Bolsens, H. J. De Man, B. Lin, K. v. Rompaey, S. Vercauteren, and D. Verkest. Hardware/software co–design of digital telecommunication systems. *Proceedings of the IEEE*, 85(3):391–418, Mar. 1997. Special Issue on Hardware/Software Co–Design.

[2] J. Daveau, G. Marchioro, C. A. Valderrama, and A. A. Jerraya. Vhdl generation from sdl specifications. In *Proceedings of the XIII IFIP Conference on Computer Hardware Description Languages, CHDL'97*, Toledo, Spain, Apr. 1997.

[3] G. Färber, F. Fischer, T. Kolloch, and A. Muth. Improving processor utilization with a task classification model based application specific hard real–time architecture. In *Proceedings of the 1997 International Workshop on Real–Time Computing Systems and Applications (RTCSA'97)*, Academia Sinica, Taipei, Taiwan, ROC, Oct. 27–29 1997.

[4] W. Glunz. *Hardware–Entwurf auf abstrakten Ebenen unter Verwendung von Methoden aus dem Software–Entwurf*. Dissertation, Fachbereich Mathematik/Informatik, Universität–Gesamthochschule Paderborn, Paderborn, München, Apr. 1994.

[5] K. Gresser. An event model for deadline verification of hard real–time systems. In *Proc. Fifth Euromicro Workshop on Real Time Systems*, pages 118–123, Oulu, Finland, June 1993. IEEE.

[6] A. Muth, F. Fischer, T. Hopfner, T. Kolloch, S. Petters, and S. Rudolph. Implementation of a CAN controller and monitor application on the rapid prototyping platform REAR. Technical report, Lehrstuhl für Prozessrechner, Technische Universität München, Oct. 1997.

[7] R. B. Ortega and G. Borriello. Communication synthesis for embedded systems with global considerations. In *Fifth International Workshop on Hardware/Software Codesign — Codes/Cashe '97*, pages 69–73, Braunschweig, Germany, 24–26 Mar. 1997. IEEE, IEEE Computer Society Press.

[8] Philips Semiconductors, Eindhoven, The Netherlands. *PCA82C200, Stand–alone CAN Controller, Product Specification*, 1992.