# Tools for Automated Implementation of Communication Protocols on Transputer Networks

Torsten Braun, Burkhard Stiller
*Universität Karlsruhe*
*Institut für Telematik*
*E-mail: [ braun | stiller ] @ telematik.informatik.uni-karlsruhe.de*

*July 11, 1994*

## Abstract

*As the number of developed communication protocols and their implementations increase, the efforts on implementing and evaluating these approaches grow significantly. Therefore, the use of protocol-independent tools that allow for the automated or semi-automated implementation and performance evaluation is obvious. The presented approach throughout this paper delineates a number of tools, which interact to provide the non-time-consuming development process. Each step is defined to clarify the treatment of specifically developed tools, their inputs and outputs as well, and of commercially available tools. Every tool is defined and their results are specified. Also, these tools are exemplified step-by-step with a number of short examples that apply to the XTP work carried out including brief performance evaluations.*

## 1. Motivation

In the area of communication protocol development several steps have to be processed achieving an executable and efficient communication protocol starting from a protocol specification. Defining a structured and step-wise clarified development process, this process can be automated in certain steps. Still, keeping the goal of efficient multi-processor implementations in mind, parallelism is one concept to apply for high performance protocols, which can be evaluated applying monitoring and measurement tools to the resulting implementation. This common approach needs a number of well understood and exact steps in the development process, since otherwise the inter-processor communications may increase to an unacceptable high amount of processing time. Careful analysis of the communication concept and basic multi-processor-capable extensions for the process interfaces are necessary preconditions for an efficient implementation. The use of multiprocessor architectures like transputer networks as a platform for high-performance protocol implementations is a promising approach, which is investigated by a considerable number of research projects, *e.g.,* [RUK92] and [

Figure 1 depicts the complete steps of the development process and their execution order. Each node of this graph represents an intermediate or final result. Edges correspond to tools. There exist two kinds of tools. One category involves tools dependent on the hardware platform (transputer network). They are commercially available within the ANSI-C-Toolset [INM90]. Examples are the regular compiler, linker, configurer and code collector. The second category encompasses developed tools dedicated to complete the task of developing communication protocols in an automated manner. Examples are the protocol configurator, an event pre-processor, the monitoring tool NETMON-II, and an evaluation tool.

The common root of the development process is the protocol specification. After the definition of multiprocessor-capable process interfaces, on one hand, the code is refined for the performance

evaluation inserting events. On the other hand, the future dynamic behaviour is derived from the protocol configurator, which results in the process configuration file, including the process location on the processors, and the physical hardware configuration file, including the physical hardware-link interconnections. After these two main independent activities the compiled and linked code is downloaded onto the configured hardware-platform. While executing the code the monitoring tool is applied to collect event-data, which show various dynamic behaviour of the executed code. Finally, these data are evaluated to calculate the current performance of the code.

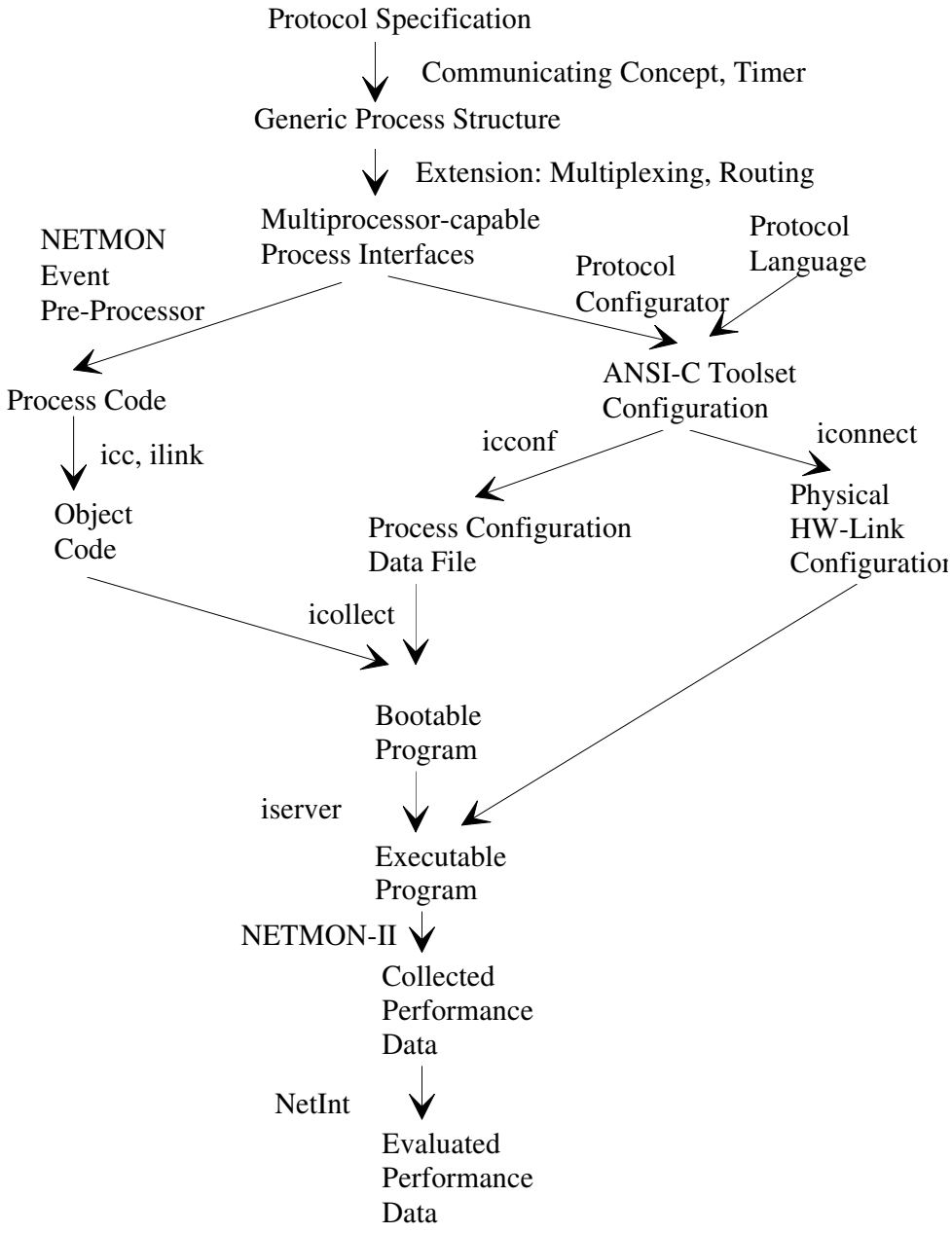


*Figure 1: Flow Chart of Design and Configuration Steps*

This paper is organised as follows. Section 2 presents the steps from the communication protocol specification to the resulting protocol code including the validated communication concepts and necessary extensions for multi-processor programs. Furthermore, section 3 outlines the preparations for achieving executable programs. Next to an event pre-processor the configurator tools are described. The evaluation of executed communication protocols is shown in section 4, including the monitoring system NETMON-II and an interpretation tool for the measurement data. Finally, section 5 summarises and concludes the paper.

## 2. Developing Multi-Processor Capable Process Interfaces

### 2.1. Mapping Protocol Specifications to Transputer Processes

Communication protocols are usually specified as a set of co-operating extended finite state machines (FSMs). In this section we describe how to map a protocol specified as a set of extended finite state machines to a set of parallel transputer tasks. Currently, the FSMs are mapped manually to transputer processes (tasks), but a kind of automatism is being used, which is described in the following.

Different FSMs of a protocol specification co-operate on common data, e.g. context data, which may describe the state of a connection, and by sending signals to other FSMs. FSM state transitions are triggered by received signals and depend on the contents of the current context data. Several types of signals may be distinguished in a communication protocol: network, timer, internal, and external signals. Network signals occur when packets arrive from the network, i.e. the layer below the specified protocol (service provider) [1]. External signals are triggered by the layer above, i.e. the service user. Internal signals are created by the protocol FSMs and are signalled to other protocol FSMs. Timer signals are triggered by expired timers. Timers are used by several FSMs of a protocol specification.

The central issue is how to implement the ***protocol FSM***s. It is impossible to create a process for each FSM because there exists always one FSM for each connection and for each FSM type. Therefore, the number of FSMs could become very high. In consequence, performance would decrease by process switching. Also creation and release of processes are very time consuming. The chosen implementation strategy creates a separate process for each FSM type [SCH90]. This process is responsible for each created connection (context). Signalling between processes is implemented by message passing across channels, which interconnect two processes. For two processes, i.e. for two FSM types, there exists exactly one channel, if the two processes are exchanging signal messages between each other. Various signal types can be exchanged across this channel. A message identifier indicates the signal type and the corresponding context.

The primary goals of the implementation strategy are efficiency and flexibility. For that reason, small FSMs that exchange a lot of signals with other FSMs and that share a considerable amount of common data with other FSMs may be combined to a product FSM. Too fine process granularity causes high communication overhead because of the communication concepts realised on transputers. In these cases, the resulting product FSMs may be implemented by single processes that can run on single processors or that can run concurrently with other processes on common processors.

The structure of a protocol process is depicted in Figure 2. After some initialisations at the beginning of the process, the process waits for new incoming signal messages. A new signal will be analysed to determine the corresponding context and the signal type. Different procedures are performed dependent on the actual state of the context variables and the signal type. These procedures contain all necessary actions triggered by the received signal message including context state modifications, sending messages to other FSMs etc.

Figure 6 shows the task structure of an implementation of XTP (XTP: Xpress Transfer Protocol) [BRZ91]. The XTP protocol FSMs are mapped to seven processes: user interface (UI), which is the interface between send and receive part of XTP and the XTP users, input (IP) to trigger reactions dependent on headers of received packets, output (OP) to format data packets and to perform functions such as flow control, rate control, and retransmission of lost or corrupted packets, control packet (CT) to create control packets such as acknowledgements, context manager process (CM) to manage the connection and the corresponding context state information, and two network interface

[1] we assume a layered communcation architecture such as OSI Reference Model or the TCP/IP protocol family

processes for send (NIS) and receive (NIR) part. Additionally, there are processes to manage data structures such as hash (HT) and routing tables (RT).

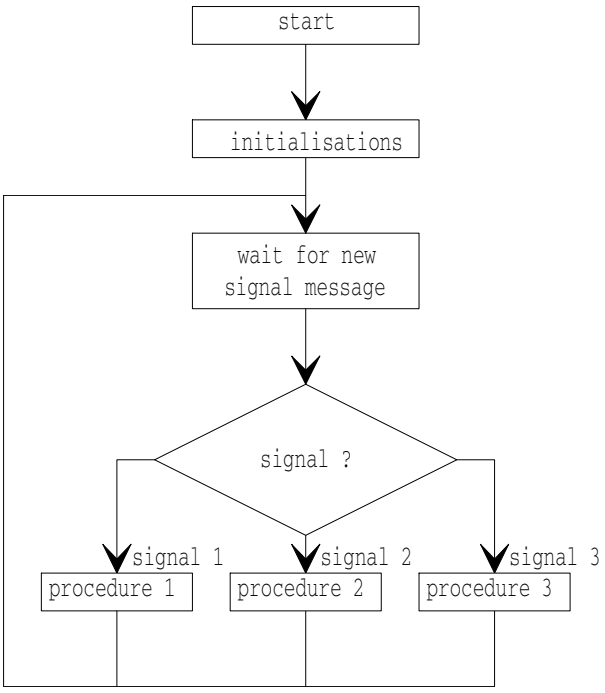


*Figure 2: Structure of a Protocol Process*

If required, the protocol processes are supported by ***timer*** processes for timer management. Protocol and timer processes communicate by message passing across two channels. A protocol process sends command messages (to start, stop, and reset timers) to the timer process. The timer process sends timeout messages to the protocol process (cf. Figure 3).

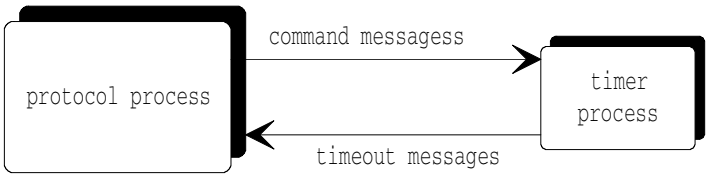


*Figure 3: Co-operation of Protocol and Timer Processes*

The timer process keeps a linked list to manage the timers. Each element of the linked list has an information field, e.g. for the connection key or the type of the timer, and a time value, which indicates the time difference to the predecessor element. In Figure 4 the timer of connection A expires at the time 2, the timer of B expires 4 time units after A, and the timer of C expires 3 time units after B.

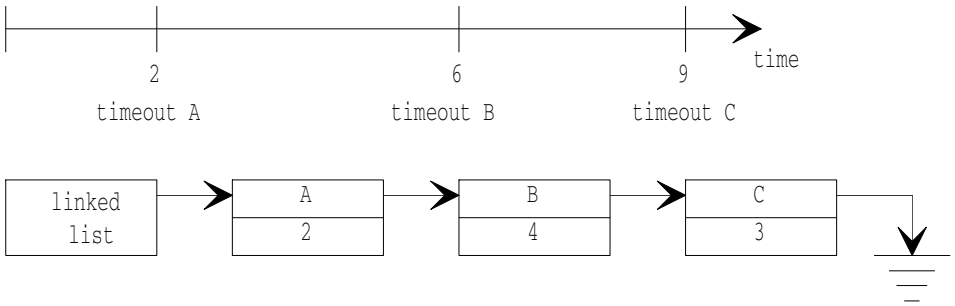


*Figure 4: Timer Management*

This management scheme allows a very efficient timer implementation on transputers. Timer and protocol process may be implemented by separate tasks. The implementation of a timer is depicted in Figure 5. The timer process waits for command messages from the protocol processes or any timeouts. The waiting time is always the time value of the first element in the linked list of time values (cf. Figure 4). Concurrent waiting for command messages and timer expiration is realised very efficiently on transputers. The process will be deactivated until a command message arrives or the specified time expires. Dependent on the reason for reactivation of the process further processing steps to manage the linked list or to send a timeout message to a protocol process are performed.

```
void timer ()
{
      while (TRUE)
      {
            switch (wait_for_message_or_timeout ())
            {
                  case MESSAGE:
                        read_synchronous_message();
                        do_timer_command ();
                        break;
                  case TIME_OUT:
                        send_timeout_message_to_process ();
                        break;
            }
      }
}
```

*Figure 5: Timer Process*

The processors used for the protocol functions have to operate on the common data areas. Those data areas may be context data, user data received, or user data to be sent. The different protocol processes operate on common context data. To allow all processes to run on different transputers, the context data is distributed as well as possible to those processes that mainly operate on it. Context data needed by several processes have to be exchanged whenever any context values change.

To avoid movements and distribution of user data, which may be of considerable size, among the involved processors, global data memories must be provided, to which several processors must have access. Using global data areas, e.g., received data packets remain at a fixed place while being processed by several protocol processors.

To provide a global data memory, ***memory manager*** processes are used. The memory manager processes for send part (MMS) and receive part (MMR) have to manage the user data packets received or to sent. Co-operation between memory manager processes and protocol processes is organised in a client/server principle. Protocol processes (clients) send requests to the memory manager processes (servers) to read or write data packet information and to allocate or release buffer space. Memory manager processes perform the desired actions and return the results to the clients. For example, a protocol process must send a command to MMR, if the process wants to read the received XTP header.

MMR returns the header data. Additionally, for test purposes, there exist so-called test processes, which substitute the underlying network (network simulation) and the XTP user (user processes). The resulting XTP process structure with all protocol, timer, memory management, and test processes is shown in Figure