OpParIm
A Method and Tool for Optimized Parallel Protocol Implementation*

Stefan Leue
Department of Electrical
& Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
Tel.: +1 (519) 888 4567 ext. 5313
Fax: +1 (519) 746 3077
E-mail: sleue@swen.uwaterloo.ca

Philippe Oechslin
Computer Networking Lab
Swiss Federal Institute of Technology
DI-LTI EPFL
CH-1015 Lausanne, Switzerland
Tel.: +41 21 693 47 08
Fax: +41 21 693 66 00
Email: oechslin@di.epfl.ch

Keywords: Protocol Implementation, TCP/IP/FTP, Specification and Description Language (SDL); Data- and Control Flow Analysis; High Speed Protocols; Protocol Optimization.

Abstract

We are introducing and discussing a method for the *op*timized and *par*allel *im*plementation of protocols as well as a tool OpParIm to apply the method automatically to the specification of a protocol. We present a study case representing an IP/TCP/FTP protocol stack specified in SDL. We show how OpParIm generates dependence graphs from the specification and how it manipulates these graphs to allow for an optimised and possibly parallelised implementation. We then present a hardware architecture on which the protocol stack could be implemented and show the effects of our optimizations on the processing time of an incoming packet. Using two processing elements the optimised implementation executes in less than half the time of what we call a 'faithful' implementation.

*The author names appear in alphabetical order. Part of the work of both authors was carried out in the course of the Δ^2 project of the University of Berne and the EPF Lausanne, funded by the Swiss National Science Foundation. The first author received support from the National Science and Engineering Research Council of Canada (NSERC).

1 Introduction

Motivation. Efficient protocol implementation is a cornerstone in the development of advanced telecommunications systems. These systems rely on high bandwidth communications infrastructures, typically based on optical transmission technology. This has led to a situation in which - as opposed to the situation in the past - not the communication link but instead the protocol processing machine is a throughput bottleneck.

Different approaches have been chosen in order to obtain fast communication infrastructures. Some development groups have chosen to develop new protocol and service classes reflecting the characteristics of the underlying broadband medium capabilities as well as the evolving upper layer application requirements. Such examples include the light-weight protocols that emerged in the late eighties [5]. Others have proposed exploring the potential for a speed-up of existing protocol architectures, as for example the IP/TCP protocol stack in [3] or [16], and we will follow and support this second direction with our method and tool. Because of the prominence of the example and to ensure comparability with other approaches we chose a simplified IP/TCP/FTP protocol stack as running example in our paper.

A few of the latter optimization approaches have been formulated in a rather intuitive fashion, without or only with little formal justification. We think that it is advisable, for obvious reasons, to embed a protocol implementation method into a more comprehensive telecommunications systems engineering methodology, based on a formal foundation. Therefore, we suggest linking our approach to formal protocol specifications. Furthermore, our algorithms are defined in a rigorous formal fashion in [12], however, space does not permit for a reproduction of these algorithms here. We will focus in this paper on a demonstration of the applicability of our method to the chosen example, i.e. the IP/TCP/FTP protocol stack.

In preceding work we observed that a 'faithful' implementation of the SDL specification, where 'faithful' means '*implementing it so that every SDL process maps uniquely to a process in the implementation, and the inter-process communication mechanism is implemented as suggested by the SDL specification*', will not be efficient. As we argue in [12], this can mainly be attributed to the fact that a) there is no explicit parallelism in SDL specification, b) the structuring of the specification into asynchronously communicating concurrent processes does not allow for the combined execution of operations belonging to different protocol layers (sometimes also referred to as *integrated layer processing* [4]), and c) the queue-based inter-process communication mechanism of SDL is not a useful mechanism inside protocol stack implementations. We will provide some more discussion of these points later when discussing the SDL specification.

Overview. In Section 2 we will introduce our study case which is an SDL specification of a protocol stack representing the IP/TCP/FTP protocols. It is not a complete specification of these protocols, but provides a framework for the protocol functionality. Section 3 demonstrates how OpParIm can be used to generate dependence graphs from the SDL specification. Here we also show how these graphs can be manipulated using the algorithms defined in [12]. Section 4 describes how an implementation can be derived from, amongst other things, the manipulated dependence graphs. The benefits of our method are exemplified by a simulated implementation of the protocol stack on a given network adapter board. Section 5 presents the conclusions of this paper.

Related Work. The literature about optimisations of protocol implementations is vast. An overview of the concepts of the domain is given in [6]. Dependence analysis of SDL specifications is also used in the domain of protocol test generation as for example in [17]. Path oriented implementation of protocol stacks, as advocated in this paper, is also central to work reported in [13]. Methods supporting the automatic combination of data manipulation operations have been published in [14] and [1].

Precursors. We have published a precursor of our method in [10]. In [11] we presented the method as it is used now. Exact definitions of the algorithms we use can be found in [12].

2 An SDL Specification of the IP/TCP/FTP Protocol Stack

2.1 Example

We will consider the SDL specification of an IP/TCP/FTP protocol stack as the starting point for the exemplification of our method and tool in this paper. The IP/TCP/FTP protocols are defined in [8], [9] and [7], however in an informal way. For the purposes of our paper we therefore developed a somewhat abstract SDL specification, see Figures 1, 2 and 3.

The specification is logically structured into three blocks, where each block contains one SDL process. The SDL processes comprise the functionality of exactly one of the protocol layers, hence there is a process `IP_process` representing the IP layer functionality, a process `TCP_process` representing the TCP layer, and a process `FTP_process` capturing the FTP layer functions. We allow and use a small number of simplifications compared to a syntactically complete SDL specification. We omit the definition of signal routes and signal lists in order to obtain a mapping from output signals to input signals, instead we use name-identity. Hence, the data unit which the TCP unit creates (called `IP_TCP_SDU` in our example) is sent by the process `IP` and received by process `TCP` when executing the `INPUT(IP_TCP_SDU)` statement. We do not explicitly represent an environment, but we assume that the protocol stack will accept an `IP_packet` data unit from the environment or, more clearly, from the underlying medium service, and that it delivers an `FTP_data` data unit to the environment at the upper boundary of the protocol stack.

The fact that we structure the specification into three processes corresponds to the intuitive approach, that a protocol machine for one layer corresponds to one SDL process. This is not binding, however, many of the examples of SDL protocol specifications in [2] will show a similar structure.

We will now briefly walk through the specification and explain its basic ideas. When referring to individual statements we will use a numbering which corresponds to the numbering at the left margin of the specifications. This numbering is an identification of statements which the OpParIm tool generates automatically and it will also be used to label the nodes in the dependence graphs of later sections.

IP. The `IP` process will input an IP packet (`S1`) and then perform a *checksum* validation of the packet header (`D1`). Depending on the outcome of this check it will either do an error handling (not specified in more detail) and return to a waiting state, or it will perform a check on the the different fields of IP header (`D2`). If this check was successful, the IP process will continue to test whether any options have been set in the packet header (`D3`), and if so the appropriate operations will be carried out (`S4`). In the next step of processing the IP process will determine whether the upper layer protocol addressed by the packet which is currently processed is TCP, UDP or another protocol. In case it is TCP the IP process will compile[1] the data unit to be handed over to the TCP process (S5) and output the intermediate data unit IP_TCP_SDU (S6). For the sake of conciseness we will only consider the case where the upper layer protocol is TCP and not pursue alternatives.

```
            SYSTEM STACK
              BLOCK IP;
                PROCESS IP_process;
                STATE waiting;
S1:             INPUT(IP_packet);
D1:             DECISION IP_check_sum(IP_packet.header);
                  (FALSE) :
S2:                 CALL error_handling();
                    NEXTSTATE waiting;
                  (TRUE)  :
                ENDDECISION;
D2:             DECISION IP_header_check(IP_packet.header);
                  (FALSE):
S3:                 CALL error_handling();
                    NEXTSTATE waiting;
                  (TRUE): ;
                ENDDECISION;
D3:             DECISION IP_test_options(IP_packet.header);
                  (TRUE):
S4:                 CALL options_processing();
                  (FALSE): ;
                ENDDECISION;
D4:             DECISION IP_test_upper_protocol(IP_packet.header);
                  ('TCP') :
S5:                 TASK IP_TCP_SDU := TCP_SDU_compile(IP_packet);
S6:                 OUTPUT(IP_TCP_SDU);
                    NEXTSTATE waiting;
                  ('UDP') :
S7:                 CALL udp();
                    NEXTSTATE waiting;
                  (ELSE ) :
S8:                 CALL other();
                    NEXTSTATE waiting;
                ENDDECISION;
                ENDPROCESS IP_process;
              ENDBLOCK IP;
```

Figure 1: Annotated SDL-PR specification of the IP/TCP/FTP protocol stack

TCP. The TCP layer offers a connection oriented data transfer service between end users over the underlying network. Hence, the TCP protocol has a connection establishment as well as a data transfer phase. We will only specify the operations executed for TCP packets (IP_TCP_SDU) belonging to established connections. The TCP process receives IP_TCP_SDU in S9. It will then perform a checksum calculation (S10), and, if this has been successful, check whether in fact the connection, to which the incoming IP-originating data unit belongs, is established (D6). If this is actually the case, then it will be tested whether any of the different control flags in the packet header (e.g. PUSH, FIN, RST) have been set. If the header is "normal", then a normal operations routine will be initiated (S12). These include adjusting the sliding window of the receiver and preparing or sending of an acknowledgment.

Next, the result of the normal operations is tested (D8) to see if the packet can be delivered. If it

[1] This operation may just consist of setting a pointer to the address of the data part of the IP packet.

is, then the upper layer protocol is identified (`D9`). For reasons of simplicity we will restrict ourselves to FTP or TELNET. If the upper layer application is FTP, then the TCP process assigns the value of the `TCP_FTP_SDU` data unit to be handed over to the FTP process (`S13`) and sends it (`S14`).

```
             BLOCK TCP;
               PROCESS TCP_process;
               STATE waiting ;
 S9:           INPUT(IP_TCP_SDU);
S10:           TASK cksm := DMO_TCP_checksum(IP_TCP_SDU);
 D5:           DECISION (cksm);
                 (FALSE) :
                   NEXTSTATE waiting;
                 (TRUE)  : ;
               ENDDECISION;
 D6:           DECISION
                 connection_state(IP_TCP_SDU.TCP_packet.header.TCP_destination);
                 ('established') :
 D7:               DECISION TCP_test_flags(IP_TCP_SDU.TCP_packet.header);
                     (ELSE) :
S11:                   CALL TCP_exception_handling();
                       NEXTSTATE waiting;
                     ('normal')  :
S12:                   CALL TCP_normal_ops(IP_TCP_SDU.TCP_packet.header);
                   ENDDECISION;
 D8:               DECISION TCP_seqno_ok(IP_TCP_SDU.TCP_packet.header);
                     (TRUE)  :
 D9:                   DECISION TCP_test_application(IP_TCP_SDU.TCP_packet.header);
                         ('FTP') :
S13:                       TASK TCP_FTP_SDU := IP_TCP_SDU.TCP_packet!data;
S14:                       OUTPUT(TCP_FTP_SDU);
                           NEXTSTATE waiting;
                         ('OTHER) :
S15:                       CALL to_other();
                           NEXTSTATE waiting;
                       ENDDECISION;
                     (FALSE) :
S16:                   CALL seqno_error_handling();
                       NEXTSTATE waiting;
                   ENDDECISION;
                 (ELSE) : ;
               ENDDECISION;
               ENDPROCESS TCP_process;
             ENDBLOCK(TCP);
```

Figure 2: Annotated SDL-PR specification of the IP/TCP/FTP protocol stack, continued

FTP. Again we restrict our specification to the part where the FTP process has established an ASCII data transfer. Upon receipt of the `TCP_FTP_SDU` data unit (`S17`) it will immediately translate the external ASCII coding of the TCP originating data unit into internal ASCII usable by the user of the FTP process (`S18`). The end of file condition will then be checked (`D10`) and the translated data will be sent to the environment, where it is expected to be consumed by the FTP service user (`S19`, `S20`).

2.2 Potential for Optimization

We are aiming at four different types of optimization:

- *The anticipation of common cases:* We separate the implementation of uncommon cases from the implementation of the common cases. The denotation 'common cases' and 'uncommon cases' refers

```
            BLOCK FTP;
              PROCESS FTP_process;
              STATE ascii_transfer;
S17:          INPUT(TCP_FTP_SDU);
S18:          TASK FTP_data := DMO_translate(TCP_FTP_SDU);
D10:          DECISION test_eof(FTP_data);
                (TRUE)  :
S19:              OUTPUT(FTP_data);
                  NEXTSTATE closing;
                (FALSE) :
S20:              OUTPUT(FTP_data);
                  NEXTSTATE ascii_transfer;
              ENDDECISION;
              ENDPROCESS FTP_process;
            ENDBLOCK FTP;
          ENDSYSTEM STACK;
```

Figure 3: Annotated SDL-PR specification of the IP/TCP/FTP protocol stack, continued

to the distinction whether an operation will be executed when processing the majority of the packets (i.e., a 'normal' operation) or not (see also [12] for a discussion). When implementing the common cases we perform systematic anticipations which allow for more efficient processing.

- *A path oriented implementation of the common cases:* We identify all operations that need to be carried out throughout a protocol stack when processing one packet (called the common path, CP). The grouped execution of these operations avoids extra message passing and context switching.
- *Integrated Layer Processing:* Having identified the operation of the common path we can apply integrated layer processing in which we combine the execution of data manipulation operations.
- *Exploitation of inherent parallelism:* The dependence analysis carried out in the previous steps also allows to parallelise the implementation of the common case operations provided the target hardware architecture provides parallel processing resources.

A 'faithful' implementation of the SDL specification could for example mean a translation of each SDL process to C code that represents its extended finite state machine, or an interpretation of the SDL specification based on an interpretation of the 'Abstract SDL Machine' (c.f. [2]). The resulting state machines can then be run as independent processes in a multitasking operating system, as independent threads in a multithreaded process or finally as independent modules in a single threaded process controled by a custom scheduler. Such an implementation would be state machine oriented rather than path oriented. Optimisations like avoidance of message passing and context switching in the CP or combined execution of data manipulations can thus not be implemented in a faithful implementation. Also, a faithful implementation would lead to implement each state machine on a different processing element. This is an arbitrary way of structuring an implementation since it reflects design choices (partitioning into SDL processes) that have not been made in respect to an efficient implementation.

2.3 The application of OpParIm to the SDL specification of the IP/TCP/FTP protocol stack

Our tool consists mainly of the following components:

1. an SDL parser which generates dependence graphs from SDL specifications,
2. a set of algorithms to manipulate these dependence graphs,
3. and a public domain back-end which transforms graph descriptions to postscript code that can be displayed or printed out.

In the next Section we demonstrate the stepwise application of our tool and method to the IP/TCP/FTP study case. Figure 4 shows a screen capture of OPPARIM working on the study case. All graphs in the subsequent Figures have been generated using OPPARIM .

3 Graph Manipulations

3.1 Dependence Analysis

As we discussed above, the exploitation of the optimisation and parallelisation potential requires a change of the order of execution for certain operations. However, in order to ensure the consistency of operation of the protocol stack it is necessary to ensure that data dependences between statements are respected. This means in particular, that an operation using some data is not executed before this data has been generated by another operation.

Another kind of dependence between operations are control flow dependences. They describe the ordering of events in the context of the SDL specification, and are determined by the (maybe partly arbitrary) order in which the specifier has written them down, as well as by the definitions in the SDL semantics (which will in most cases prescribe sequential order of execution for directly neighboring statements in the code).

Our dependence analysis is a syntactic operation on the code. We will analyse each statement in the specification with regard to the reference to data variables. If a statement assigns an initial or a new value to a variable or if a particular variable is an output parameter of a procedure call[2], then this statement is a *define* statement with respect to this variable. If a statement references a variable such that the variable occurs on the right hand side of an assignment operator or if the variable occurs as an input parameter of a function or a procedure call, then we say that this statement is a *use* statement with respect to this variable. The communication statements `INPUT` and `OUTPUT` require some additional thought. `INPUT(X)` means that the receiving SDL process will consume a message of type `X` from its input queue, and we assume that afterwards a variable named `X` is defined in the process' data space. Hence, an `INPUT(X)` statement will be treated as a *define* statement with respect to a variable `X`. Analogously, an `OUTPUT(X)` statement is *use* statement with respect to variable `X`.

Control Flow Dependences. The analysis of the code with regard to *control flow* dependences (*cfd*) is quite straightforward. One statement is *cfd*-dependent on another if the latter is a directly preceding statement in the control flow of the SDL process. Subsequent statements are *cfd* dependent, and the possible successors of a `DECISION` statements are as well *cfd*-dependent on the `DECISION` statement.

[2] Our tool currently assumes that procedure calls only have input parameters. Handling of output parameters could be implemented by looking up the procedure definition to identify the input and output parameters.

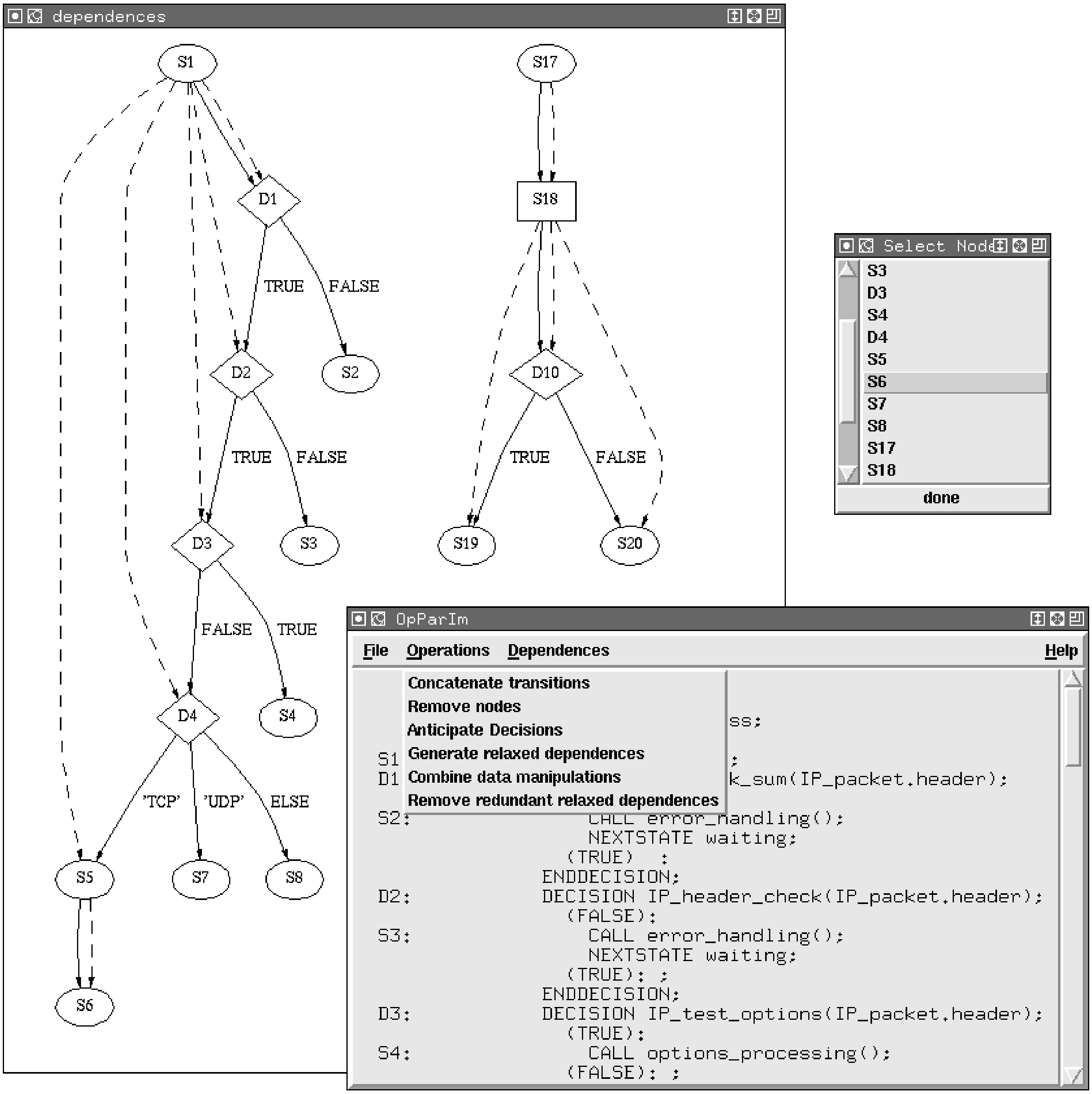

Figure 4: Screen capture of OpParIm

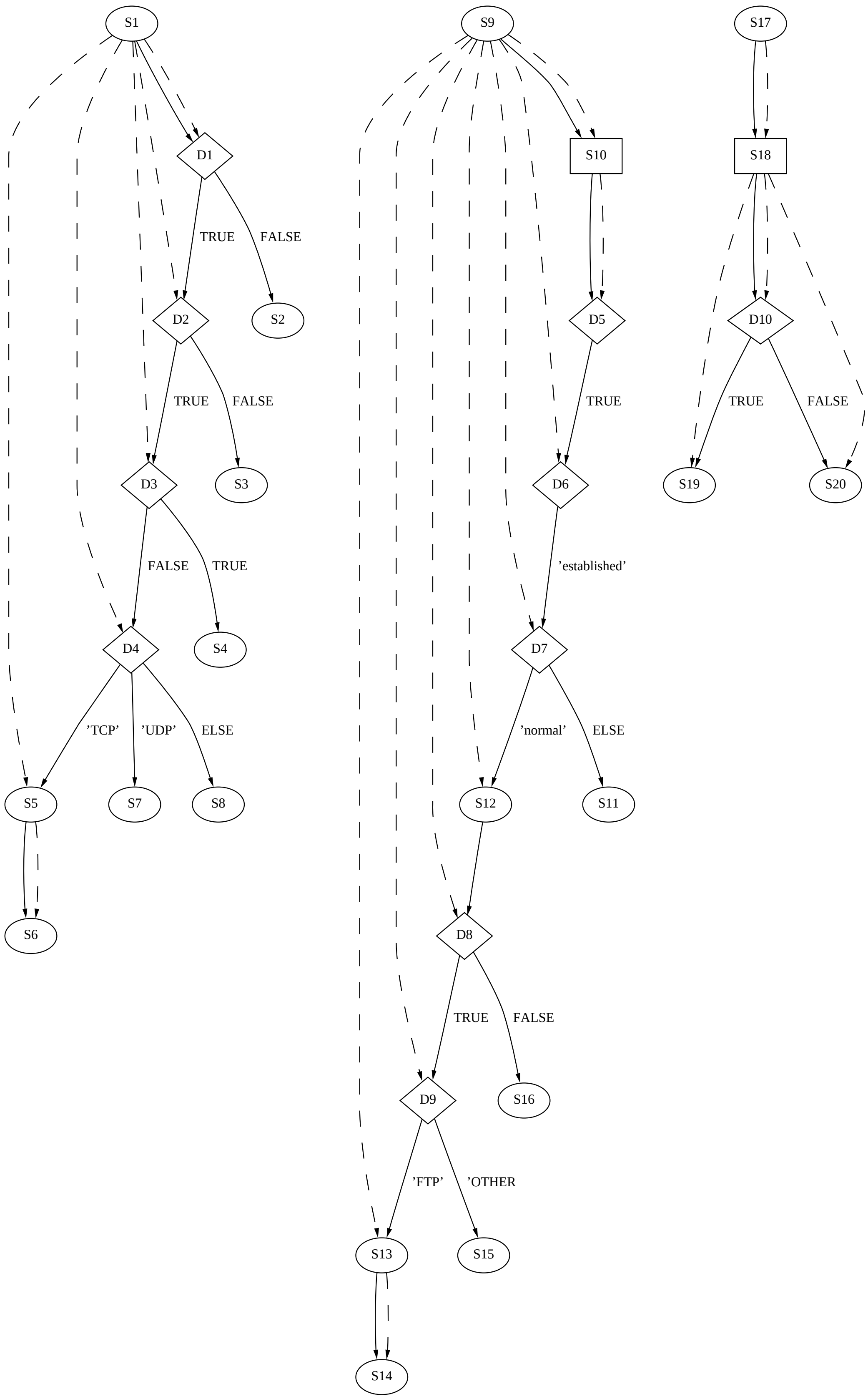

Figure 5: Dependence graphs for IP (left), TCP (middle), and FTP (right).

Data Flow Dependences. Data flow dependences are only considered between pairs of nodes which are in the transitive closure of the *cfd* relation. The *cfd* relation is a tree, hence free of cycles. Thus, an 'earlier' statement in the *cfd* relation can not be data dependent on an 'later' statement in *cfd*. Given two different statements S and T, such that (S, T) is in the transitive closure of *cfd*, the following rules govern the determination of *data flow* dependences (*dfd*)(see [15]):

- *Direct dependence:* T is *dfd*-dependent of S iff it uses a variable which S defines.
- *Output dependence:* T is *dfd*-dependent of S iff it defines a variable which S also defines.
- *Antidependence:* T is *dfd*-dependent of S iff it defines a variable which S uses.

Discussion of Example. Figure 5 shows the dependence graphs generated by our tool after parsing the SDL specification in Figures 1, 2 and 3. The solid lines indicate control flow dependences and the dashed lines data dependences. Decisions are represented by diamonds and data manipulation operations by boxes. All other instructions are represented by ovals.

3.2 Preparing Integrated Layer Processing

The term *Integrated Layer Processing* (ILP) has been coined by Clark and Tennenhouse in [4], although integrated processing was already reported earlier for example in [3]. It was observed that when executing data manipulation operations (DMOs), which are operations that manipulate the complete data part of a packet, more time is usually spent for fetching and storing the data than for performing the actual operations. DMOs include operations like checksum calculation, encryption, translation of data representation or simple copy operations. A considerable gain in efficiency can be obtained by combining the execution of two DMOs since the data which they need to access has only to be fetched and stored once. ILP refers to the fact that DMOs located in different layers of a protocol stack can only be combined if these layers are processed in an 'integrated' way, i.e. by executing them jointly.

In our IP/TCP/FTP example we identify two DMOs, namely the checksum calculation of TCP and the translation from external to internal ASCII in FTP. As the DMOs are located in different layers, a faithful, layer-oriented implementation can not possibly combine both operations. We therefore propose a path oriented implementation of the operations. As we will see, the path oriented implementation also allows for the elimination of asynchronous message passing and context switching in the implementation of the common cases. We know that in the common case the processing of a packet will follow a path through all three layers of our protocol stack. We thus concatenate the dependence graphs of the three processes and by-pass the message-passing asynchronous queues. When concatenating the dependence graphs we remove the nodes corresponding to the output and input operations and add a control dependence between the predecessor in *cfd* of the output operation and the successor in *cfd* of the input operation. Likewise the data dependences between nodes that were data dependent on the input operation and those of which the output operation was data dependent are re-computed. The exact algorithm for the concatenation of the dependence graphs is given in [12]. The result of applying this operation is a multi-layer dependence graph (MLDG) as shown in the output of OpParIm in Figure 6.

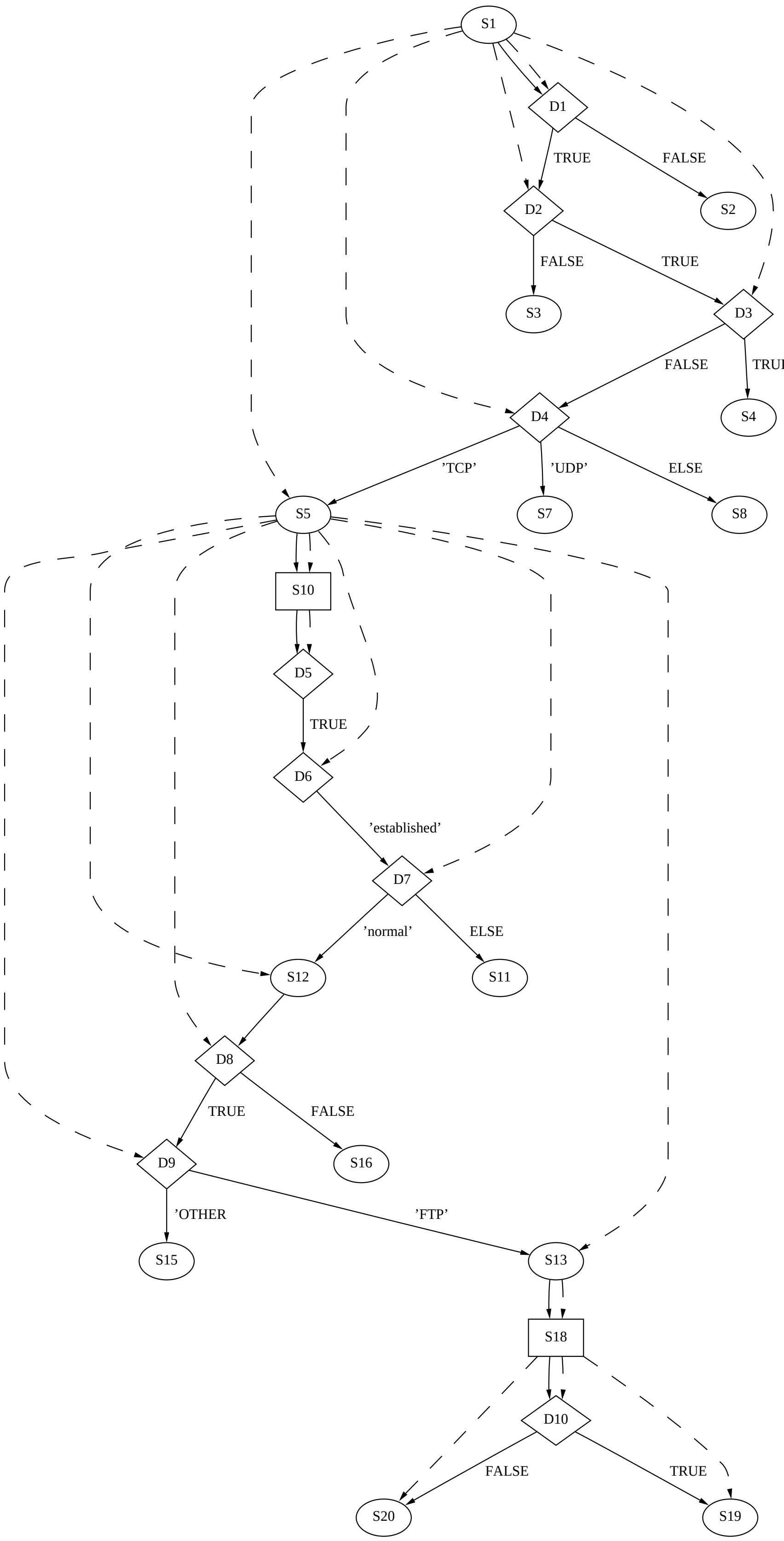

Figure 6: Multi-layer dependence graphs

3.3 Determination of the Common Path

In typical protocol implementations many tests have to be carried out to detect different kinds of errors and exceptions. These can be due to transmission errors in the data or the header of packets, or to connection closing or tear down. Error recovery and exception handling may well make up a larger part of the implementation than the part dealing with the most common operations. In our method we identify the path that packets follow within the protocol in the common case. This will be what we call the *common path* (CP). Separating the CP from the rest of the implementation allows to optimise the implementation where it is most effective. Moreover, it allows to apply optimisations (mainly anticipations) which will speed up the execution of common cases although making uncommon cases more complicated to handle. Similar concepts are being used in the current development of the communication oriented operating system SCOUT [13].

In our example we have already omitted the specification of most uncommon parts of the protocols. These include the details of error handling in both IP and TCP and the whole connection establishment and release phases in TCP and FTP. To obtain only the CP in the graphs generated in Figure 5 we have to identify which nodes correspond to operations which will only be executed in very few cases. These are `S4` (wrong IP checksum), `S11` (TCP flags) and `S16` (wrong sequence number). For the purpose of this paper we assume the the common case in our example to be determined by the following assumptions:

- IP receives error free packets for any upper layer protocol,
- TCP receives error free packets in sequence for any upper layer, and
- FTP processes are in a state in which an ASCII transmission is established.

3.4 Anticipation of the Common Path Decision Outcome

The determination of the common path graph leaves us with a number of nodes which have only one successor, but which represent nodes corresponding to `DECISION` statements (called *decision* nodes, for short). An example of such a node is node `D5` which represents the decision whether the TCP checksum calculation indicates an error or not. Without making further assumptions we cannot change the order of decision nodes and nodes which are *cfd* dependent on these since this might cause inconsistent behavior of the protocol.

However, these sorts of decision nodes impose a sequential ordering which is obstructive when parallelising and reordering other operations. For example, we might be interested in inverting the ordering of nodes `D5` and `D6` (determination of the TCP connection state) but this is impossible as long as we assume that `D6` will only be executed in case `D5` is evaluated to `TRUE`.

What we do at this point is to anticipate the outcome of these decision nodes, following the common-path assumption that a packet will be processed in most cases as described by the common path. Technically, in the tool this simply means that we remove the tag 'decision node' from the anticipated nodes. The result of this operation is shown in Figure 7. Henceforth, we allow an arbitrary ordering of these anticipated decision nodes with other, non-*dfd*-dependent nodes. In the above example we might for example allow for an arbitrary execution order of nodes `D5` and `D6` after having anticipated their outcome which converts them into regular operations.

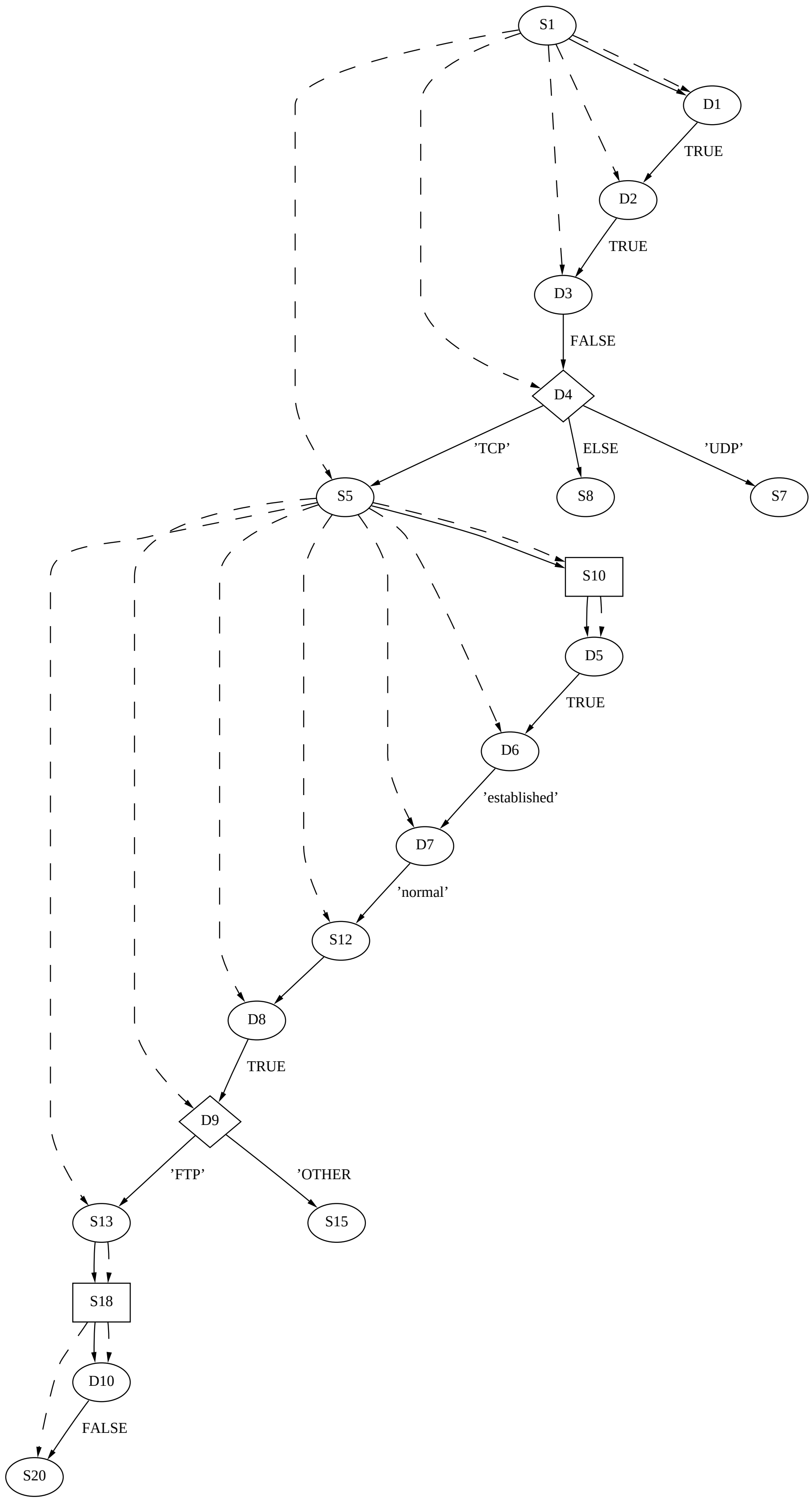

Figure 7: Common path graph, after the anticipation of decisions `D1`,`D2`, `D3`, `D5`, `D6`, `D7`, `D8` and `D10`

However, this anticipation requires later consistency checks, in particular checks on whether a given packet actually complies with the common path assumption, e.g. whether the assumption that the TCP checksum was correct, holds for this particular packet. This will be done when executing *exit* nodes which are nodes when the packet we consider is handed over to the environment. If, at this point, the falseness of an anticipation is detected we have to re-establish the state of the whole system as it was when the packet entered the protocol stack. For a discussion of means to ensure this we refer the reader to [12].

3.5 Relaxing Dependences

Control flow dependences induce a total order on the operations of a protocol stack. However, these dependences need not to be respected to achieve a consistent behavior of the protocol. In fact, a correct execution can be enforced with a more relaxed set of dependences. Relaxed dependences allow to speed up execution in two ways, namely by enabling parallel execution and by allowing the execution of operations in a different order. We thus create a new dependence relation called relaxed dependences *rxd* which consists of the original *dfd* dependences plus some additional dependences that ensure a consistent execution. The additional dependences are:

1. A dependence between each operation and the nearest decision they transitively depend on in *cfd*. This ensures that operations are not executed before it has been decided that they need to be executed.

2. A dependence from each output operation to all of the nodes that it transitively depends on in *cfd*. Output operations represent an interaction of the protocol with its environment. When the protocol interacts with its environment it does two things: it conveys some data to the environment, and it implicitly tells the environment that it has reached a certain point in the execution of the protocol. Hence, at that point, the environment may assume that all operations that have been specified to be executed before the interaction have actually been carried out, and the additional dependences ensure this. They also ensure that all anticipated decisions have been carried out and that the common case assumption can thus be verified before the protocol interacts with the environment.

Figure 8 represents the relaxed dependence graph of our example. Looking at the common path graph in Figure 7 we observe that interaction with the environment, namely the release of the FTP data to the user (`S20`), is not *dfd*-dependent on the checks of the IP header fields (`D2`). However, when executing `S20` we need to check whether our common case anticipation, hence we need to ensure that `D2` is executed before `S20`. Therefore we introduce the consistency ensuring *rxd* dependence in Figure 8 between both nodes. Furthermore, `D7`, `D8` and `S13` do not depend on each other and could thus be executed in parallel. It does not seem to be efficient to implement each independent node on a parallel processor. However, many modern hardware architectures allow some parallelism, for example between bus transfers, CPU calculations and checksum calculation on dedicated hardware. As opposed to respecting all ordering constraints as specified by the *cfd* relation, respecting the orderings of the *rxd* relation allows to take advantage of possible parallelism. The partial order defined by the *rxd* relation furthermore gives more freedom in the ordering of operations, even in a sequential implementation. This allows for a more efficient utilization of resources like instruction and data cache, because, for example, the execution of operations can be combined on the processor level.

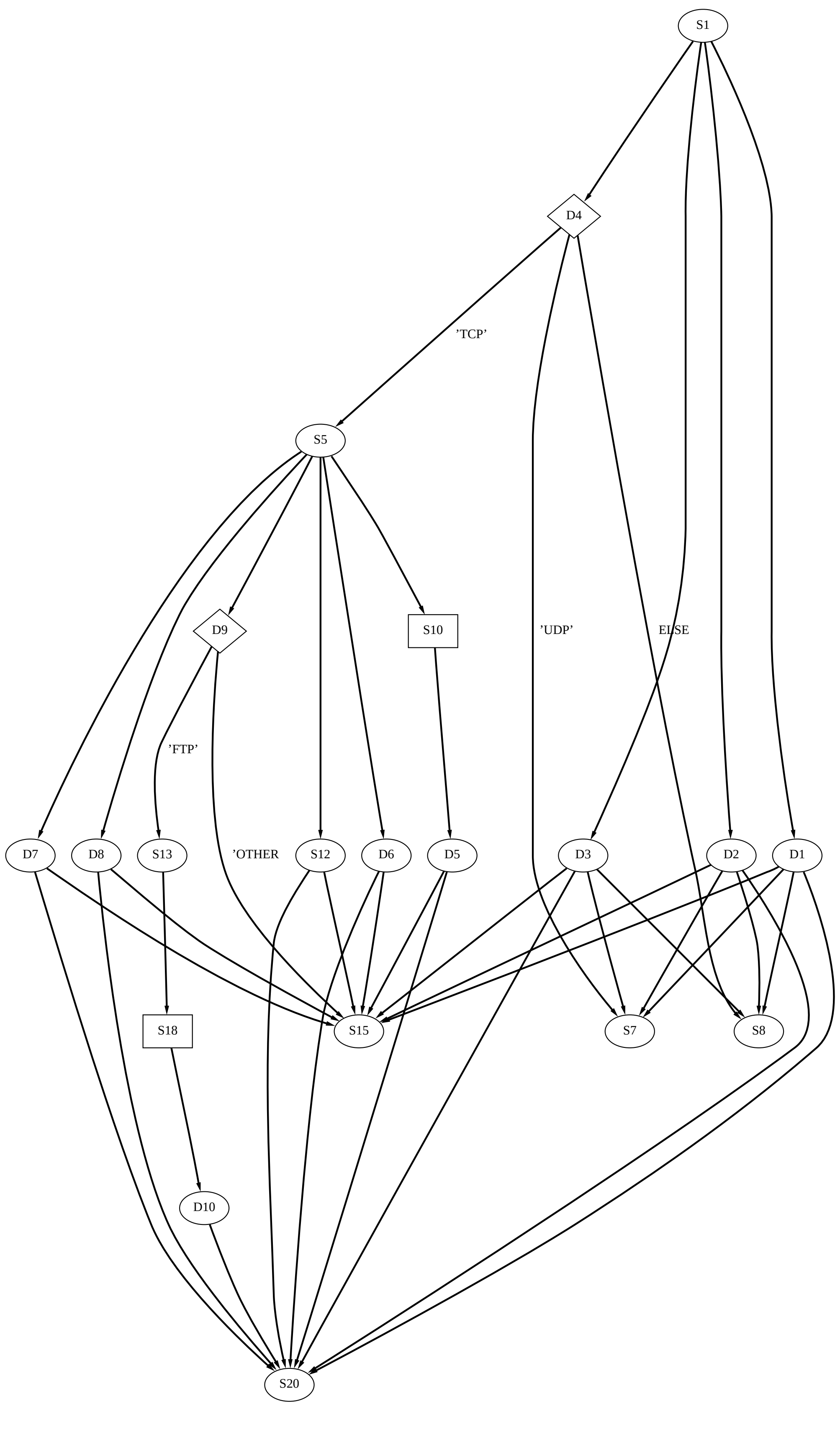

Figure 8: Relaxed common path graph

3.6 Applying Integrated Layer Processing

We call *data manipulation operations* (DMOs) those operations that manipulate the entire data part of a protocol data unit. Examples are checksum calculation and encryption of data. Combining two such operations into one that does two manipulations at the same time saves an extra storing and fetching of all the data. Therefore, we conjecture, these combined operations execute much faster than the non-combined execution of both operations. DMOs are currently identified by OpParIm as functions with names prefixed by the letters "`dmo`". A more advanced analysis of the specification might be capable of determining automatically which operations are DMOs.

In our example we identify two DMOs, namely operations `S10` and `S18`. Operation `S10` corresponds to the calculation of the TCP checksum, and operation `S18` represents the translation from external to internal ASCII representation. To know if we can execute these operations in a combined fashion we first have to know whether the second DMO needs to be executed at all. Therefore we have to wait with `S10` until decision `D9` has been executed. Then we can either execute a combined DMO `S10&S18` or `S10` alone. We have devised an algorithm that transforms dependence graphs to explicitly allow the combined execution of DMOs. It delays DMOs and duplicates them over different outcomes of decisions if this allows to combine them with DMOs that can only be executed after these decisions. The algorithm is described in more detail in [12]. The algorithm is implemented in our tool and can automatically be applied to relaxed dependence graphs. Figure 9 shows the relaxed dependence graph after application of the algorithm. We see that there are two copies of S10, `S10'1` and `S10'2`, where the latter is combined with `S18`. Both copies depend on `D9` which ensures that both copies are executed only after it is known whether the combined execution is possible. Note that `D5`, as it depends on S10, has also been duplicated to make sure it is executed after `S10'1` or `S10'2` but for any evaluation of `D9`.

4 Implementation

The methods used in our tool could be applied by an optimizing compiler to automatically generate efficient implementations. To demonstrate the effect of the optimizations we will carry out the steps of implementation manually for the common path of our example. We will then compare the result of an optimized and an unoptimized implementation.

4.1 The Hardware

The first step of the implementation consists in defining the hardware on which the protocols will be executed. For this example we will use a simple hardware architecture (see Figure 10). It represents a network adapter board with two interfaces: the medium access interface (MAC) to the network and the host interface (HI) to the host machine. Only the receive path is covered by our example. There are two dual-port RAMs acting as FIFO's between the interfaces and the processing part. The processing part is made of a general purpose RISC microprocessor containing a cache, its program RAM (which also contains the state information for all open connections) and a Data Manipulation Unit (DMU), typically a Digital Signal Processor (DSP) or a Field Programmable Gate Array (FPGA). The DMU reads packets from FIFO1, performs the necessary translations and calculations and stores the resulting packet in FIFO2. The microprocessor processes the headers and updates the state of the connections.

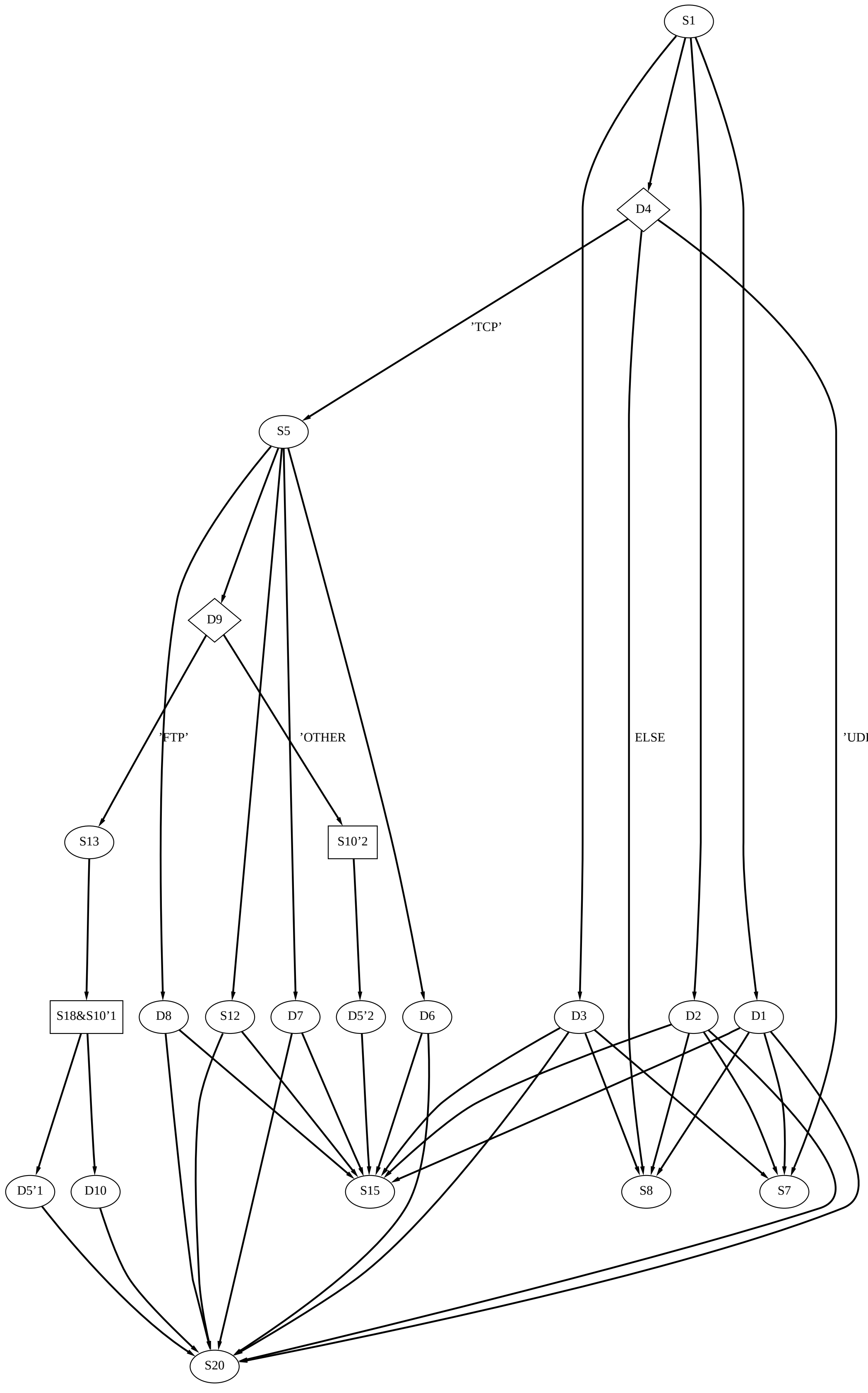

Figure 9: Dependence graph after automatic combination of DMOs

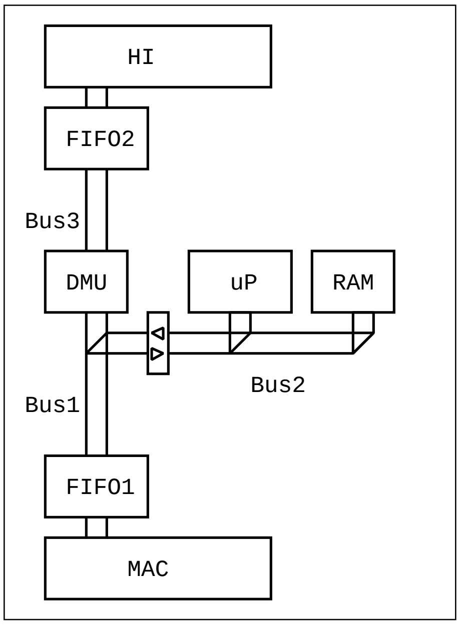

Figure 10: A hardware architecture for the receive path

Its bus (Bus2) can be isolated from the bus that connects FIFO1 to the DMU (Bus1) to allow parallel processing. This is achieved using a bidirectional three-state buffer.

4.2 Compiling the operations

Once the hardware is defined we can compile the operations of the common path. To do so we map the operations identified by the nodes of the graphs to instructions executed by some resources of the hardware. For the combined DMOs we have to generate the instructions for their combined execution. A method for doing this automatically has been given in [1]. Also we will have to add some additional instructions in the output operation `S20` to verify that all the anticipations have been verified. If an anticipation turns out to be wrong then `S20` will have to call an exception handler which will undo all processing done for this packet and will hand it to an unoptimized implementation of the protocol. This unoptimized implementation, together with the implementation of the uncommon parts, could be executed by the main processor of the machine to which the network adaptor board is attached.

For our example we assume that the packets have a length of 1500 bytes, that the busses are 32 bits wide and that they can transfer one word per clock cycle. Table 1 gives the list of resources and indicates which resources will be used for each operation as well as how many cycles are needed for execution[3]. Note that the processor will access the packet header only once in FIFO1. Afterwards the header is retained in the processor cache. The time for transferring the header from FIFO1 to the cache has been attributed to operation D1 as it will be the first access to the header. The numbers in the table are estimations with the sole purpose of illustrating the effects of the optimizations. We argue that for a real implementation the effects of the optimizations would be similar.

[3] Operations `S7`, `S8` and `S15` are not represented in the tables as we only illustrate TCP packets for FTP connections.

	FIFO1	BUS1	BUS2	UP	RAM	DMU	BUS3	FIFO2	cycles
D1	*	*	*	*					20
D2				*					5
D3				*					1
D4				*					1
D5				*					1
D6			*	*	*				20
D7				*					1
D8				*					1
D9				*					2
D10				*					1
S1	*								1
S5		*	*	*		*			2
S10'2	*	*				*	*	*	375
S12				*					200
S13		*	*	*					2
S10'1&S18	*	*				*	*	*	375
S20		*	*	*		*	*		10

Table 1: Required resources and execution time for each operation

operation	starting time	end time	operation	starting time	end time
S1	0	1	S10&S18	28	403
D4	1	2			
S5	2	4			
D9	4	6			
S13	6	8			
D1	8	28			
D2	28	33			
D3	33	34			
D6	34	54			
D7	54	55			
S12	55	255			
D8	255	256			
D10	403	404			
S20	404	414			

Table 2: An optimal schedule on the proposed hardware architecture

4.3 Scheduling

Given the resource allocation table (Table 1) and our last dependence graph (Figure 9) an optimal schedule can be found for the execution of the operations of the common path. This schedule can be found at compile-time, for example by using a branch and bound algorithm. It can then be coded into the operations, such that no dynamic resource allocation, locking, or synchronization is needed at run-time. An optimal solution for our example requires 414 cycles to reach exit node `S20` from root point `S1`. One optimal schedule is given in Table 4.2.

4.4 Results

The sequential execution of the original common path suite of operations on the same hardware would take 1018 cycles. This is more than twice as long as the execution time of the optimised graph. Hence, we achieved a gain of 604 cycles. 229 cycles are gained by executing `D2`, `D3`, `D6`, `D7`, `S12` and `D8` in parallel with the combined DMOs. Further 375 cycles are gained by combining the DMOs. Both of these gains are due to the fact that we have analyzed all the operations needed for the treatment of a packet through the complete protocol stack, and that we have relaxed the dependences between these operations. Which of parallel execution or combined DMOs offers the most potential of optimization depends on the underlying hardware, on the complexity of a given protocol, and on the number of DMOs it contains.

5 Conclusion

We have presented a method and a tool to enable the automatic generation of optimised protocol implementations based on their specification. It leads to a path oriented implementation of the common cases with automatic anticipations, parallelisations and combinations of data manipulation operations. We have introduced a study case representing an IP/TCP/FTP protocol stack and have shown the effects of our method with a simulated implementation of the study case. Using two processing elements our optimised implementation is more than twice as fast as a ‘faithful’ implementation. The implementation gains in efficiency through the combined execution of data manipulations and through the exploitation of limited parallelism as it is available in many modern hardware designs. These optimisations have been reached through a dependence analysis of the original SDL specification and through the application of different transformations to the dependence graphs.

The fact that we have been able to implement our method as a tool indicates that it could be included in an ‘intelligent’ specification compiler. A standard way of compiling specifications consists of first translating them into programming language code and then to use an optimising compiler for this programming language to obtain executable code. The intelligent compiler we envisage, however, will take protocol specific optimisations derived from the specification into account before translating the specification into programming language code. This ensures that the optimization potential at the abstract protocol specification level is exploited. The optimized dependence graph our method and tool generates constitutes the input to the intelligent compiler in order to exploit this high level optimization potential.

References

[1] M. Abbot and L. Peterson. Increasing network throughput by integerating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.

[2] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

[3] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, Jun. 1989.

[4] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM '90 conference*, Computer Communication Review, pages 200–208, 1990.

[5] W. A. Doeringer et al. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions On Communication*, 38(11):2025–2039, Nov 1990.

[6] D. Feldmeier. A framework of architecutral concepts for high-speed communication systems. *IEEE Journal on Selected Areas in Communications*, 11(4):480–488, 1993.

[7] Information Sciences Institute. File transfer protocol. NIC-RFC 959, 1981.

[8] Information Sciences Institute. Internet protocol. NIC-RFC 791, Sep. 1981.

[9] Information Sciences Institute. Transmission control protocol. NIC-RFC 793, 1981.

[10] S. Leue and Ph. Oechslin. Optimization techniques for parallel protocol implementation. In *Proceedings of the Fourth IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 387–393. IEEE Computer Society Press, 1993.

[11] S. Leue and Ph. Oechslin. Formalizations and algorithms for optimized parallel protocol implementation. In *Proceedings of the 1994 International Conference on Network Protocols*, pages 178–185. IEEE Computer Society Press, 1994.

[12] S. Leue and Ph. Oechslin. A formal approach to optimized parallel protocol implementation. Technical Report IAM-94-003, revised version, University of Berne, Institute for Informatics, Berne, Switzerland, Apr. 1995. Also submitted for publication, entitled *On Parallelising and Optimising the Implementation of Communication Protocols.*

[13] A. Montz, D. Mosberger, S. O'Mealley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, Department of Computer Science, The University of Arizona, 1994.

[14] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In M. J. Johnson, editor, *Protocols for High Speed Networks II*, pages 141–156. Elsevier Science Publishers (North-Holland), 1991.

[15] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, 29(9):763–776, September 1980.

[16] C. Partridge and S. Pink. A faster UDP. *IEEE Transactions on Networking*, 1(4):429–440, Aug. 1993.

[17] H. Ural and A. Williams. Test generation by exposing control and data dependencies within system specifications in SDL. In R.L. Tenney, P.D. Amer, and M.Ü Uyar, editors, *Formal Description Techniques, VI*, IFIP Transactions C-22, pages 335–350. North-Holland, 1993.