On Parallelising and Optimising the Implementation of Communication Protocols

Stefan Leue, *Member, IEEE*, and Philippe Oechslin, *Member, IEEE*

***Abstract*— We present a method for the automatic derivation of efficient protocol implementations from a formal specification. Optimised efficient protocol implementation has become an important issue in telecommunications systems engineering as recently network throughput has increased much faster than computer processing power. Efficiency will be attained by two measures. First, the inherent parallelism in protocol specifications will be exploited. Second, the order of execution of the operations involved in the processing of the protocol data will be allowed to differ from the order prescribed in the specification, thus allowing operations to be executed jointly and more efficiently. The method will be defined formally which is useful when implementing it as a tool.**

1 Introduction

A consequence of the evolution of telecommunications systems and in particular of the underlying optical transmission technology is that, as opposed to conventional communications systems, the performance bottleneck is no longer the transmission link, but instead the protocol processing machine. It is consequently imperative to have efficient protocol implementations available. We are therefore looking at the following problem: given the design of a protocol, how can we derive an implementation from this design which exploits its inherent optimising potential? To answer this question we need a) to define which kind of *design* information we should base our method on and b) to clarify which *optimising steps* we are looking at.

Design. We follow a trend in telecommunications systems engineering to base the software development on formal methods, in particular on formal protocol specifications [25] [33]. The implementation method we propose is based on formal specifications using the ITU-TS standardised *Specification and Description Language* SDL [4]. The formality requirement is useful when implementing the method as a tool, and we will consequently define all our transformation steps formally.

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

S. Leue is with the Department of Electrical & Computer Engineering, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

Ph. Oechslin is with the Communication Networks Laboratory, Department of Computer Science, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland.

Optimising steps. The optimising steps we envisage are a) the exploitation of the *inherent parallelism* in a protocol specification (or, in case of multi-layered protocols, of the entire protocol stack), b) the systematic anticipation of what we call the "*common case*", and c) the combined execution of operations belonging to different protocol layers, sometimes also referred to as *integrated layer processing*, allowing particular operations to be executed jointly and more efficiently. All these optimising steps require the ordering of the operations in the protocol to be changed, compared to the original specification. This requires a formal analysis under which conditions these changes of ordering can be carried out, and a description of the algorithms which allow the changes to be made.

Sketch of the method. We propose a method to transform the sequential structure of operations inside the processes of an SDL specification into optimised relaxed dependence graphs which may later serve as a basis for an efficient, potentially parallel implementation of the specified protocol stack. We first derive a data- and control flow dependence graph from each SDL process. Then, in order to perform cross-layer optimisations we combine the dependence graphs of different SDL processes. Next, we determine the common path through the multi-layer dependence graph. We then parallelise this graph wherever possible which yields a relaxed dependence graph. Based on this relaxed dependence graph we interpret different optimisation concepts that have been suggested in the literature, in particular the combination of data manipulation operations. The resulting graph can then be used in order to implement the protocol stack on either a sequential or a parallel machine architecture, which leads to the necessity to solve a scheduling problem of operations on the different hardware resources. The parallelism we envisage is a low-degree parallelism, reflecting the possibilities of modern microcomputer architectures[1].

Overview. It is the objective of this paper to provide a formalised description of the above sketched implementation method. In Figures 2 and 3 we present a (partial) view of the SDL specification of a two layer protocol stack which will serve as a running example. The example is given in

[1]Although our method is independent of any particular system architecture considerations we envisage a hardware architecture in which multiple operations can be executed in parallel even in a single processor environment, like modern workstation or network adaptor board architectures.

both the graphical (GR) and the equivalent textual (PR) SDL syntax. It is the purpose of our optimisation and implementation method to transform SDL specifications similar to into parallelised and optimised implementations.

In Section 2 we discuss the sort of layered SDL specifications we consider. Here, we also argue why a direct and faithful implementation of SDL specifications would lead to inefficient implementations. Then we turn to a description of our analysis and optimisation method:

- First, we construct a dependence graph representing control-flow and data dependences among statements in an SDL specification. This leads us to so-called *Transition Dependence Graphs*. Their construction is explained in Section 3.1.
- Second, we optimise and parallelise operations related to processing a packet along the way the packet takes from the point where it enters the protocol stack to where it exits. Therefore we combine transition dependence graphs belonging to different SDL processes and eliminate the inter-layer communication mechanism. The result is a Multi-Layer Dependence graph (see Section 4).
- Third, we identify the path a packet takes through the protocol stack in the so-called common case, from the root node representing the point where a packet is accepted from the environment to the exit node, where the packet is conveyed to the environment. The resulting graph is called *common path graph* (CPG), for its construction see Section 5. We will apply our later optimisations only to the common case part of the specification.
- Fourth, we relax dependences on the common path graph in the following steps.
  - *Anticipation of the common case:* In this step we ignore that certain statements depend on a decision, namely for those decisions for which we assumed a common outcome. Henceforth we treat these decision nodes as if no other node depends on their execution.
  - *Parallelising:* We construct a relaxed dependence graph by only retaining the data flow dependence relation of the CPG and by adding additional dependences which ensure that a node is never executed before the last decision node on which it depends in the control flow dependence relation has been executed (see Section 6.2).
- Finally, in Section 7 we show how suggestions that have been made in the literature to optimise the implementation of communication protocols can be interpreted based on the relaxed dependence graph. We refer to the concepts of *Lazy Messages* (see [28]), and, in particular, *Grouping of Data Manipulation Operations* or *Integrated Layer Processing* (see [7], [8] and [1]).

The optimised and parallelised graph now serves as a foundation for an implementation on either a sequential or a parallel machine architecture. We discuss some issues concerning an implementation of the optimised graph in Section 8.

Related Work. Efficiency of implementation has become an imperative requirement in the context of high speed protocols. Aspects of hardware and software architecture that increase an implementation's efficiency are discussed in [7], [8], [28], [10] and [32]. Hardware implementations for high speed protocols have been proposed in [18]. In the literature on optimised protocol implementation special attention has been paid to parallelising protocol implementations, so for example in [5] and [34]. However, the guidelines for parallelising proposed in these papers depend mainly on the intuition of the designer and thus its efficiency may be non-optimal. Therefore, automated support when parallelising is desirable. [13] describes parallelising methods which have the per-packet approach with ours in common, but lack a formal justification of the parallelising steps. [16] suggests a one object per protocol layer implementation of protocol stacks. This approach exploits some of the protocols inherent parallelism but does not take advantage of integrated layer processing. [17] suggests parallel protocol implementation based on a highly parallel architecture, applied to higher OSI layers, without providing formal justification for the approach taken. The authors observe that a lot of processing time is consumed by representation transforming operations. We address this point formally when suggesting integrated layer processing. The approach taken in [9] resembles our approach in that it points at shortcomings of a horizontal implementation of protocols and suggests a vertical approach. [14] observes that, although classical multiprocessor implementations perform poorly, fine-grain parallelism in dedicated hardware offers the most promising solution for high-speed protocol processing. An approach based on the scheduling of parallel tasks generated by an Estelle compiler is presented in [12]. [26] describes the determination of data-flow dependence graphs for parallel implementations of stream processing programs on transputers.

The dependence graph construction is an application of methods known from the domain of compiler optimisation and parallel compilation as they are for example described in [11] and [3]. Work presented in [30] analyses data flows in networks of Communicating Finite State Machines for the purpose of the detection of so-called non-progress properties. [31] suggest a method for the analysis of data flows in distributed communicating processes in order to enable the detection of unreachable program statements and the compile-time determination of values of program expressions. Closely related work is included in [19] which analyzes the data- and message flow dependences between communicating processes for static analysis purposes (e.g. compile-time deadlock detection). The algorithms given are highly complex. Our later assumption that there is a one-to-one mapping of send and receive primitives in the code greatly facilitate the message flow analysis in our model and, in fact, makes it trivial.

Precursors. An earlier version of our method has been applied to an IP/TCP/FTP protocol stack SDL specification [21]. Our method is briefly described in [23], and in more detail in [22]. Extended descriptions of our method also appear in [20] and [27].

The Role of SDL. The formal specification technique we consider is the ITU-TS standardized *Specification and Description Language* SDL [6]. We consider this language because it enjoys wide acceptance in the protocol engineering community. For an overview of SDL see [4] and [33]. The choice of a formal description technique as a starting point connects our method to existing techniques and methods in the domain of telecommunications systems and protocol engineering [25]. We may for example assume that as result of a previous verification step the specifications on which we base our optimisation are verified with respect to certain correctness criteria, e.g. dead- and live-lock freeness.

Part of our method (the dependence analysis and the construction of multi-layer dependence graphs) are specific to features of SDL. However, we claim that for many other procedural specification methods and even for most procedural concurrent programming languages an easy adaptation of our method is possible. The later steps (starting with the CPG construction and down to the optimisation steps we describe) are independent of the specification method on which the dependence graph is based.

2 A Discussion of SDL Specifications

In this Section we argue why 'faithful' implementations of SDL protocol stack specifications are inefficient which gives rise to our 'non-faithful' implementation method.

2.1 SDL Specifications of Protocol Stacks

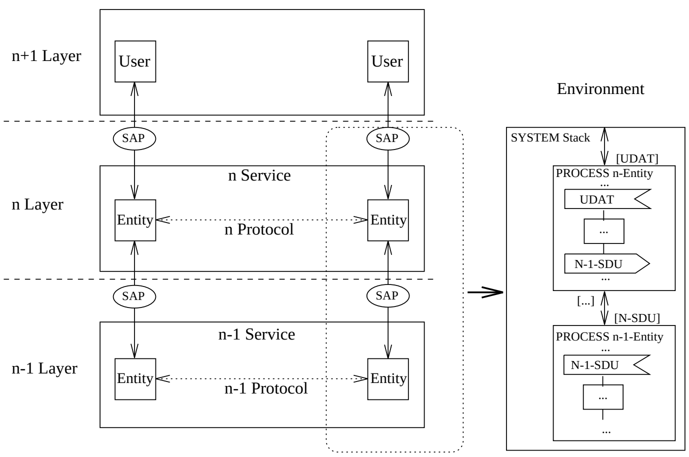

Figure 1: Layered protocol architecture and schematic SDL specification of a two-layered protocol stack.

Communication and Concurrency. SDL is a Formal Description Technique frequently used for the layered specification of communication protocols. Figure 1 shows a schematic model of the representation of a protocol stack by an SDL specification where each layer consists of a number of interacting protocol entities. In SDL processes communicate with the environment as well as with other processes via *asynchronous communication* through *process-unique input queues of unbounded capacity.* In the example in Figure 1 the process `n-Entity`, which represents the layer n protocol machine, communicates with the adjacent layer process `n-1-Entity` via the exchange of `N-1-SDU` messages, and with the user located in the environment by exchange of `UDAT` messages.

The processing inside an SDL process is sequential. However, at run-time all processes belonging to an SDL specification run concurrently, so an SDL specification can be seen as a collection of sequential processes that run in parallel. Each process can be structured into a set of transitions, each transition leading from a symbolic state to another or the same symbolic state, triggered by an input signal (see for example Figure 2). A transition may lead to many successor states, the choices are either made by logical decision predicates, or by checking the different `INPUT` events by which a transition can be triggered. For many examples of protocol and service specifications based on SDL see [4] and [33].

Asynchronous message exchange using the SDL primitives `INPUT` and `OUTPUT` seems to be the mechanism most frequently used for inter-layer communication in protocol specifications. However, the SDL standard introduces further mechanisms. Communication between processes can also be through SDL *remote procedure calls*, through a so-called *viewing* mechanism allowing processes to share variables, and finally an *import/export* mechanism which, however, only hides an asynchronous message exchange. Finally, an extension of SDL by a synchronous communication primitive has been suggested in [15]. In the next Sections we will assume that inter-layer communication is only through asynchronous message exchange. A discussion of the adaptation of our method to the alternative communication mechanisms can be found in [20].

The Two-Layer Protocol Stack Example

The *Two Layer Protocol Stack* example of two protocol processes `N` and `N+1`, which we assume to belong to adjacent layers of some protocol stack, are presented in Figures 2 and 3[2]. Both processes are only partially specified: Process `N` accepts either a message of type `X` from a non-specified lower layer service, which is then processed and sent out as a message of either type `Y` or type `Z`, or it accepts a message of type `U` which after processing is being sent out as a message of type `V`. Hereafter, we shall sometimes abbreviate the terminology by saying *a message* `X` instead of *a message of type* `X`. We will also facilitate the SDL sender and receiver

[2]Note that this will be the running example throughout the subsequent development.

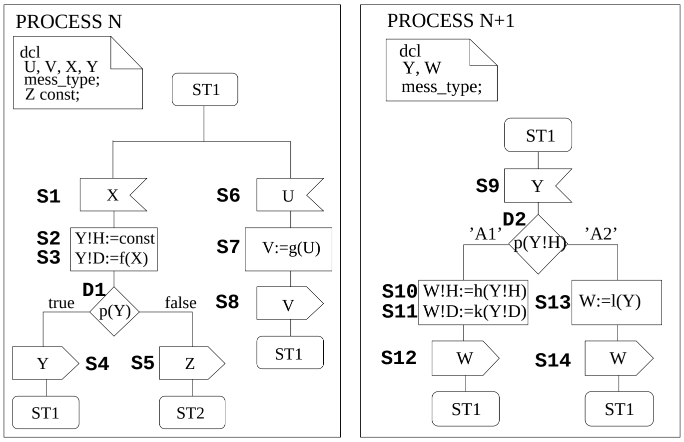

Figure 2: SDL-GR representation

```
   PROCESS N;
   ...
   STATE ST1;
S1 INPUT(X);
S2 TASK Y!H:=const;
S3 TASK Y!D:=f(X);
D1 DECISION P(Y);
     (true):
S4     OUTPUT(Y);
       NEXTSTATE ST1;
     (false):
S5     OUTPUT(Z);
       NEXTSTATE ST2;
   ENDDECISION;
S6 INPUT(U);
S7 TASK V:=g(U);
S8 OUTPUT(V);
   NEXTSTATE ST1;
   ...
   ENDPROCESS N;
```

```
    PROCESS N+1;
    ...
    STATE ST1;
S9  INPUT(Y);
D2  DECISION p(Y!H);
      ('A1'):
S10     TASK W!H:=h(Y!H);
S11     TASK W!D:=k(Y!D);
S12     OUTPUT(W)
        NEXTSTATE ST1;
      ('A2'):
S13     TASK W:=l(Y);
S14     OUTPUT(W);
        NEXTSTATE ST1;
    ENDDECISION;
    ...
    ENDPROCESS N+1;
```

Figure 3: SDL-PR representation

mapping mechanism and assume that sender and receiver are matched by the identity of the message type name.

2.2 Inadequacy of 'Faithful' Implementations

By the term *faithful implementation* we refer to an implementation which follows in its structure and in the sequence of operations exactly the original SDL specification from which it is derived. This may for example mean (a) that the SDL specification is directly compiled so that every statement in the SDL specification is mapped to a (sequence of) statement(s) in the implementation, (b) that every SDL process corresponds to a process in the implementation, and (c) that the processes in the implementation communicate using the SDL asynchronous communication mechanism via infinite queues. However, as we argue in the following such a faithful implementation is not efficient.

No Explicit Parallelism. Although SDL processes run concurrently the processing inside an SDL process is strictly sequential. This means that the structuring of the specification into processes, which in many cases is influenced by more or less arbitrary design decisions, determines the degree of parallelism of a specification. It also means that without optimisations the sequential processing of operations inside a process may be inefficient compared to a parallel execution.

Structuring of the Specification into Processes. The structure of the specification often means that there is one process per protocol layer peer entity of the protocol (see for example the specifications presented in [4] and [33]). The design of communication protocols is often governed by the principle that *'a good specification is a highly modular and layered specification'*. We stipulate that in order to derive efficient parallel protocol implementations such a layered design is obstructive. This can mainly be attributed to the fact that the parallel scheduling and combined execution of operations belonging to different protocol layers, which can lead to a considerable gain in efficiency, are inhibited by the layer-wise structuring of the specification. Similar arguments can be found in [10].

Asynchronous Inter-Layer Communication via Infinite Queues. An efficient implementation of a protocol stack for one peer entity will usually be a non-distributed system. Apparently, it can be very inefficient to implement the exchange of data in such a non-distributed system via asynchronous queues.

The objectives of our method are therefore to remove the boundaries between processes, to remove the asynchronous communication between processes, and to analyze dependences between statements so that parallel and combined execution of statements belonging to different processes is enabled.

3 Dependence Analysis for SDL Processes

In this Section we explain how a data- and control-flow dependence graph can be obtained by syntactic analysis from an SDL specification. For a definition of the mathematical notation we use here and in later Sections we refer the reader to the Appendix. First, we will explain how transitions as basic building blocks of SDL process specifications can be formalised and then, how entire protocol stacks can be represented as graphs, built up from the graphs representing single transitions.

3.1 Transitions in SDL Specifications

Syntactic structure. A *transition* in an SDL specification is a construct which describes the transition of an SDL process from one *symbolic state* into a successor symbolic state. The body of a transition consists of a collection of statements which we group in the set of statements S. We only consider a limited subset of SDL-statements, namely `INPUT`, `TASK`, `DECISION` and `OUTPUT` statements, and we identify one of these four statement types with every element of S.

Justification. For the sake of conciseness we have limited our considerations to the above described SDL language subset but we conjecture that an adequate treatment of other language constructs is a straightforward extension. Furthermore, we conjecture that the language subset chosen here allows for an analysis of most of the standard protocol specifications as presented in [4][3].

3.2 Control Flow and Data Flow Dependences

The syntactical analysis of the SDL specifications that we describe in this Section yields a graph structure over the set of statements S of an SDL specification. This so-called *dependence graph* identifies the two types of dependences between members of S, namely *control flow* and *data flow* dependences.

Dependences.

- Statements, which according to the syntactical and semantical rules of SDL are direct successors, are part of the *control flow dependence* relation *cfd* over the set S. A statement of type `DECISION` has two or more directly succeeding statements, all pairs of a `DECISION` statement and its successor statements are part of the *cfd* relation. The execution of a statement directly succeeding a `DECISION` statement depends on the run-time evaluation of the decision predicate. This is represented by a branching of the *cfd* graph. We will in later optimisation steps, in particular when parallelising the dependence graph, have to ensure that statements will only be executed when the decisions on which they depend have been taken.
- Statements usually reference process variables in two different ways: First, we say that a statement S_n *uses* a variable x iff it references the variables current value without modifying it. Note that in one statement more than one variable may be used. A typical use of a variable would be to reference its value in the expression on the right hand side of an assignment statement. Second, we say that a statement S_n *defines* a variable x iff it assigns an initial or new value to the variable. A typical example is the definition of a variable on the left hand side of an assignment statement. It should be noted that all assignment statements are *single assignment* statements.
- A pair of statements (s_1, s_2) is in the *data flow dependence* relation *dfd* if and only if (s_1, s_2) is in the transitive closure of the *cfd*-relation[4] and s_2 *uses* a variable which is *defined* in s_1. For simplicity we assume that no re-definition of variable names inside transitions occurs[5]. Also, we assume that every variable name used in a transition is defined inside of the transition, therefore no data dependences from statements in other transitions exist. Function calls are assumed to have no side-effects and to return a single value. Assignments to structured variables are decomposed into component-wise assignments. An `INPUT(X)` statement is a *define* statement with respect to a variable named X, an `OUTPUT(Y)` statement is a *use* statement with respect to variable named Y[6].

3.3 Transition Dependence Graphs (TDG)

Definition Transition Dependence Graph. Let S, STT and X denote pairwise disjoint sets, the elements of which we call *statements*, *statement types* and *variables*. Formally, we define a Transition Dependence Graph (TDG) as a tuple

$$T = (S, STT, X, sttype, use, define, cfd, dfd)$$

where

- $cfd \subseteq S \times S$,
- $dfd \subseteq cfd^+$,
- $STT = \{input, decision, task, output\}$,
- $sttype \subseteq S \times STT$ is a functional relation (relating a statement to a statement type),
- $use \subseteq S \times \mathcal{P}(X)$ is a functional relation (relating a statement to the set of variable names which are being *used* in it), and
- $define \subseteq S \times X$ is a partial functional relation (relating a statement to the variable name which is being *defined* in it),

satisfying the following conditions:

1. (S, cfd) is a tree.
2. $\forall s \in S$ the following conditions hold:
   - $(sttype(s) = \{input\}) \leftrightarrow (\mid \{s\} \lhd cfd \mid = 1 \wedge root(S, cfd) = \{s\})$ (an `INPUT` statement has exactly one successor, and it is the root of the tree),
   - $sttype(s) = \{decision\} \rightarrow \mid \{s\} \lhd cfd \mid \geq 2$ (every `DECISION` node has at least two successors),
   - $sttype(s) = \{task\} \rightarrow \mid \{s\} \lhd cfd \mid \leq 1$ (every `TASK` node has at most one successor), and
   - $sttype(s) = \{output\} \rightarrow s \in leaves(S, cfd)$ (an `OUTPUT` statement is a leaf of the tree).
3. $(\forall (v, w) \in dfd)(define(v) \in use(w))$.

[3] However, we do not explicitly treat timers. Their treatment needs to be handled in the implementation of the exit nodes (see Section 8). Furthermore, we do not treat two `OUTPUT` statements within one transition, this can be handled by two exit nodes in many situations. Also, we assume that the `SAVE` construct is not being used along what we later call the *common path* (see Section 5.2).

[4] Thus our definition of the data dependence implies that an 'earlier' statement in the control flow cannot be data dependent on a 'later' one.

[5] This avoids additional *output* dependences, see [29].

[6] The data dependences we consider are purely local to the processes, we do not consider data dependences between processes caused by message flows.

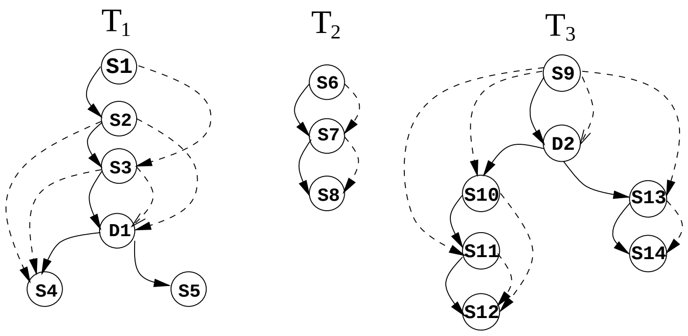

Figure 4: Data and control-flow dependence graphs for processes of our example

3.4 Example SDL Processes and TDGs

In the specification of our example (Figures 2 and 3) we have added labels `Sn` and `Dn` to help us to identify regular and decision statements, respectively. The same labels are used to identify the corresponding nodes of the dependence graphs resulting from our syntactic analysis. These labels, however, are not part of the specification.

The example in Figure 4 shows a partial view of the specification of a process `N` of which we show only two transitions. The transition on the left hand side leads from state `ST1` via statements `S1`, `S2`, `S3`, `D1` and either via `S4` to a successor state `ST1` or via `S5` to successor state `ST2`, depending on the evaluation of the decision predicate `p(Y)`. This transition is triggered by the input of an `X` signal. In statements `S2` and `S3` the variable `Y` is defined. We assume that a variable of type `mess_type` is defined as a record, and that for example the expression `Y!H` refers to the first component of the record `Y` and `Y!D` to its second component[7]. The evaluation of the decision predicate `p(Y)` determines whether a message `Y` or a message `Z` will be issued, and hence whether the successor state will be `ST1` or `ST2`.

The dependences are as follows. The *control flow* dependence follows the linear sequence of the statements `S1`, `S2`, `S3` and `D1` and then branches to either `S4` or `S5`. The `DECISION` statement `D1` has possible successor statements `S4` and `S5`, the respective control flow dependence edges are labeled for illustrative purposes by *true* and *false*. The data flow dependences are so that `S3` depends on `S1` because of variable `X`, whereas `D1` and `S4` both depend on `S2` and `S3` because of the use of variable `Y`. Figure 4 presents a graphical representation of this TDG which we call T_1. *Solid* line arrows represent control flow dependencies, thus elements of *cfd*, and *dashed* line arrows represent elements of *dfd*. When in state `ST1` process `N` may execute two different transitions. If the head of the input queue contains a message of type `U` the corresponding second transition leads from `ST1` via statements `S6`, `S7` and `S8` to state `ST1`. The resulting transition dependence graph is T_2. We will use the process named `N+1` first presented in Figure 2 and its dependence graph T_3 as a further example in our development. Note that the parallel execution of statements is allowed if they are neither directly nor indirectly data flow dependent on each other. In TDG T_1 statement `S3` is control flow, but not data dependent on statement `S2`, hence these statements can be executed in parallel.

[7]Think of `Y!H` to stand for the header and `Y!D` for the data part of a protocol data unit or a packet.

4 Dependence Graphs for Protocol Stacks

As we argued in Section 2.2, it is advantageous to remove the boundaries between layers of SDL processes and to eliminate the inter-layer communication via infinite queues. In this Section we describe the necessary steps to combine the transition dependence graphs of different SDL processes and to remove the communication between them. Technically, we perform this in two steps: First, we label all TDGs of all processes by so-called *input/output labels*. These labels are the names of the signals exchanged by the `INPUT` and `OUTPUT` statements at the beginning and at the end of each transition. Second, we combine all TDGs with matching input/output labels, eliminate the `OUTPUT(X)` / `INPUT(X)` statement pairs, and perform a cross-layer data dependence analysis. We may do this because we assume that every `OUTPUT` statement can be mapped to a unique `INPUT` statement of another process. The result is a graph which we call *Multi-Layer Dependence Graph.*

4.1 Input/Output labeled Transition Dependence Graphs (IOTDGs)

We assume that all transitions we consider for the combination process start with an `INPUT` statement accepting a data packet from an adjacent layer process, and end with an `OUTPUT` statement which delivers the processed packet to the next adjacent layer process. Hence, we assume that all the processing for a packet in a layer process is carried out in the course of *one* transition, and that no looping and branching due to `JOIN` statements inside a transition occurs. Thus, our dependence graphs are always trees. Different transitions starting in different states in one process may exist, but they only represent the process to be in different states (e. g. state *waiting* and state *transmission*). Furthermore, we assume that the packet passing is unidirectional, either from the medium towards the user or vice versa.

Formal Definition of Input/Output labeled TDGs. Based on the above stated assumptions on the structure of the SDL transitions we formalise the concept of labeling of root and leaf nodes of TDGs by the appropriate signal names as follows. Let $T = (S, STT, X, \mathit{sttype}, \mathit{use}, \mathit{define}, \mathit{cfd}, \mathit{dfd})$ denote a TDG and let SIG denote a set disjoint from any other set in sight, the elements of which we call *signal names.* Furthermore, let $\mathit{insig} \subseteq ((S \cap \mathit{root}(T)) \times SIG)$ and $\mathit{outsig} \subseteq ((S \cap \mathit{leaves}(T)) \times SIG)$ denote functional relations. We define an Input-Output labeled Transition Dependence Graph (IOTDG) as a tuple

$$I = (S, STT, X, SIG, \mathit{sttype}, \mathit{cfd}, \mathit{dfd}, \mathit{insig}, \mathit{outsig})$$

for which the following conditions hold[8]:

- $\mathit{sttype}(\mathit{root}(I)) = \mathit{input}$, and
- $(\forall x \in \mathit{leaves}(I))(\mathit{sttype}(x) = \mathit{output})$.

These two conditions imply that all transitions we consider start with an `INPUT` statement and end with an `OUTPUT` statement. In other words, we exclude all those transitions that do not end with an `OUTPUT` statement.

4.2 Multi-layer Dependence Graph (MLDG)

What we have obtained so far is a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of IOTDGs. $\mathcal{T}$ represents the dependences of all transitions of the specification that we analyze. In this Section we describe an algorithm that transforms $\mathcal{T}$ into a set $\mathcal{M}$ of Multi-Layer Dependence Graphs (MLDG). Each MLDG represents the dependences of the processing of one packet or protocol data unit in adjacent layers of the protocol stack. We are interested in following the processing of one packet from the code location where it enters into the protocol stack to the location where it exits. In our example this means that we will derive a connected control flow dependence graph from statement `S1`, where the packet `X` enters the processing in process `N`, to the statements `S12` and `S14`, where it exits the stream of processing in process `N+1` as a message of type `W`. Thus we have to compose the individual IOTDGs in $\mathcal{T}$. The criterion for composing two IOTDGs will be that they exchange a message with identical names, e. g. one IOTDG ends with an `OUTPUT(Y)` statement and another IOTDG begins with an `INPUT(Y)` statement. We assume that the names of the types of the messages exchanged are unique at the interfaces between two processes, and that the direction of the message flow is uniquely determined by the message type names. Also, we assume that every `OUTPUT` statement can be mapped to a unique `INPUT` statement. Note that SDL transitions are deterministic on `INPUT` signals, i. e. in one state the future behavior is uniquely determined by the type of the message that is consumed next.

MLDG Construction Algorithm. The functioning of Algorithm 1 is as follows. First, a set $\mathcal{T}'$ of initial IOTDGs is selected (step I.). This set contains all those IOTDGs that do not input a message that is output-ed by another IOTDG. The algorithm then loops over all these IOTDGs (III.). The set $\mathcal{Z}$ (V.) contains all those IOTDGs that can be appended to a leaf node of an IOTDG from $\mathcal{T}$. The next loop (VI.) performs the merging of two IOTDGs (VII. to XVI.) for all elements of $\mathcal{Z}$. The merging of two IOTDGs comprises the elimination of the two nodes x and $\mathit{root}(Z)$ by which the two graphs are merged (IX.), this corresponds to the elimination of the `OUTPUT/INPUT` statements. XIII. describes the construction of the new *cfd* relation. Every node which depended on $\mathit{root}(Z)$ is made dependent on every node from which x depended. The construction of the new *dfd* relation (XIV.) is very similar, but we additionally check whether a node on which x depended defines a variable which is used in a node that depended on $\mathit{root}(Z)$. XVII. constructs the result, a set $\mathcal{M}$ of MLDGs.

[8] We omit mentioning the relations *use* and *define* in this and later definitions of modified dependence graphs.

Algorithm 1

I. SELECT $\mathcal{T}' = \{T'_1, \ldots, T'_m\} \subseteq \mathcal{T}$ SO THAT
$(\forall T'_i)(\forall T_j)(\mathit{insig}(\mathit{root}(T'_i)) \cap \bigcup_{j \neq i} \mathit{outsig}(\mathit{leaves}(T_j)) = \emptyset)$;
II. $\mathcal{M} := \emptyset$;
III. FOR ALL $T'_i \in \mathcal{T}'$
  IV. $M := T'_i$;
  V. $\mathcal{Z} := \{T \in \mathcal{T} \mid \mathit{outsig}(\mathit{leaves}(M)) \cap (\mathit{insig}(\mathit{root}(T))) \neq \emptyset\}$;
  VI. WHILE $\mathcal{Z} \neq \emptyset$
    VII. FOR ALL $Z \in \mathcal{Z}$
      VIII. SELECT $x \in \mathit{leaves}(M)$ SO THAT
      $(\mathit{outsig}(x) \in \mathit{insig}(\mathit{root}(Z)))$;
      IX. $S'_M := S_M \cup S_Z - \{x\} - \mathit{root}(Z)$;
      X. $X'_M := X_M \cup X_Z$;
      XII. $\mathit{sttype}'_M := S'_M \lhd (\mathit{sttype}_M \cup \mathit{sttype}_Z)$;
      XIII. $\mathit{cfd}'_M := \mathit{cfd}_M \cup \mathit{cfd}_Z - (\mathit{cfd}_M \rhd \{x\}) - (\mathit{root}(Z) \lhd \mathit{cfd}_Z)$
      $\cup \{\mathit{domain}(\mathit{cfd}_M \rhd \{x\}) \times \mathit{range}(\mathit{root}(Z) \lhd \mathit{cfd}_Z)\}$;
      XIV. $\mathit{dfd}'_M := \mathit{dfd}_M \cup \mathit{dfd}_Z - (\mathit{dfd}_M \rhd \{x\}) - (\mathit{root}(Z) \lhd \mathit{dfd}_Z)$
      $\cup \{(v, w) \in \{\mathit{domain}(\mathit{dfd}_M \rhd \{x\}) \times \mathit{range}(\mathit{root}(Z) \lhd \mathit{dfd}_Z)\}$
      $\mid \mathit{define}(v) \subseteq \mathit{use}(w)\}$;
      XV. $M := (S'_M, STT_M, X'_M, \mathit{sttype}'_M, \mathit{cfd}'_M, \mathit{dfd}'_M)$
    XVI. $\mathcal{Z} := \{T \in \mathcal{T} \mid \mathit{outsig}(\mathit{leaves}(M)) \cap (\mathit{insig}(\mathit{root}(T))) \neq \emptyset\}$;
  XVII. $\mathcal{M} := \mathcal{M} \cup M$

As a result we obtain a set of MLDGs $\mathcal{M}$. Each $M \in \mathcal{M}$ is a multi-edged labeled tree $(S, STT, X, SIG, \mathit{sttype}, \mathit{cfd}, \mathit{dfd})$. Note, however, that not all of the conditions we required for IOTDGs still hold. For example it is not true any more that a node of type *input* has no predecessor in the *cfd* relation.

For a MLDG M we say that a node in $\mathit{root}(M)$ is an *entry* node, that a node in $\mathit{branchnodes}(M)$ is a *branching* node, and that a node in $\mathit{leaves}(M)$ is an *exit* node. An entry node represents a statement where a message (in most cases a packet or protocol data unit) is accepted from the environment, and an exit node refers to a statement in the code where a message is delivered to the environment.

Figure 5 shows the set $\mathcal{M}$ which we obtain by applying our algorithm to the IOTDGs of our example. It contains two MLDGs, one with root `S1` and one with root `S6`. Note that the *cfd*-relation forms the skeleton of the MLDGs. The nodes `S4` and `S9` have been eliminated, reflecting the elimination of the `OUTPUT(Y) / INPUT(Y)` statement pair. The additional *cfd* pair $(D1, D2)$ has been added. Furthermore, data dependences between statements of the two merged graphs have been added, so for example $(S2, S13)$.

Justification for the MLDG construction. When building the MLDG we modified the original SDL specification in two ways. First, we ignored the asynchronous queue communication mechanism, and second, we eliminated the corresponding `OUTPUT / INPUT` statement pair. The justified question arises whether these modifications preserve the correctness of the original specification. We argue that ignoring the queue can be justified because this is a refinement step which preserves two essential queue properties, namely 1. the *safety* property that it is always true that if

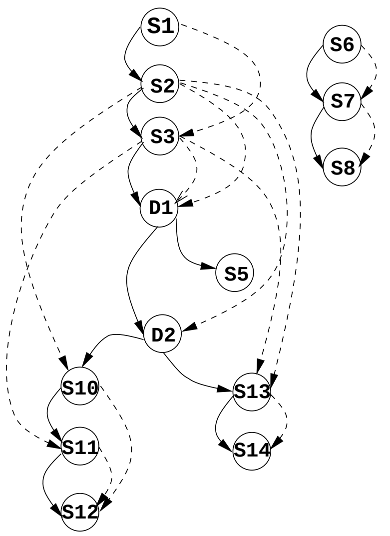

Figure 5: MLDGs for our example

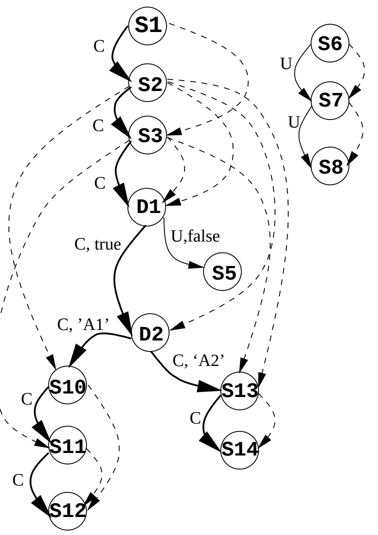

Figure 6: Labeled MLDGs for our example

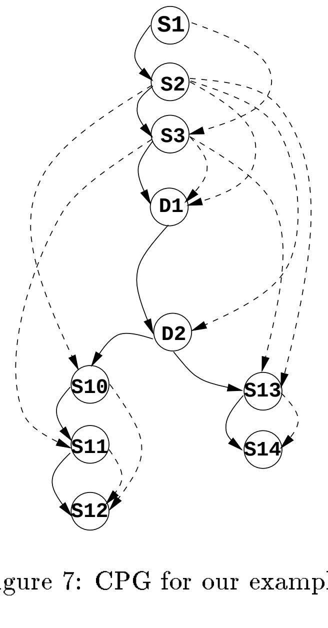

Figure 7: CPG for our example

something is received it must have been sent before, and 2. the *liveness* property that it is always true that if something is sent it will eventually be received.

The safety property is trivially satisfied because the order of the `OUTPUT(X)` and `INPUT(X)` statements is preserved. The liveness property is satisfied if we assume our implementation to be live, namely that every transition which is continuously enabled will eventually be taken.

Another way of looking at it is to consider the traces generated by each of the alternatives. Let $!X$ stand for an `OUTPUT(X)` and let $?X$ stand for an `INPUT(X)` event and let the system be in an infinite loop. Then the language of events that can be observed in the case of asynchronous queue communication for the original SDL specification can be described by the language expression

$$(!X^{n_i}?X^{m_i})^\omega \text{ with } (\forall i)(\sum_i n_i \geq \sum_i m_i)$$

whereas our implementation generates the expression $(!X?X)^\omega$. Hence, the traces generated by our implementation are a subset of the traces allowed by the original SDL specification. We can say that out of the many interleavings of events which are possible according to the original specification we only implement one possible representative, namely the interleaving where a packet is accepted at one end of the protocol stack, entirely processed, and finally handed over at the other end before the next packet is accepted for processing[9].

5 Determination of the Common Path Graph

The later steps of our optimisation method rely on the assumption that we optimise the processing of a packet only for the 'common case' (we will come to a clearer understanding of this expression in this Section). Restricting the optimisation to the common case has the advantage of reducing the complexity of the code that needs to be optimised and therefore leads to more compact optimised code modules. Furthermore, in Section 6 we introduce optimisation steps that anticipate certain common decision results according to a common case assumption. These optimisation steps, which rely on relaxing the dependences of statements before and after certain decisions, would be impossible without the common case assumption. We consider our common path determination a generalization of the *Common Path* optimisation as advocated in [7].

Protocols usually have the task of hiding imperfect behavior of lower layer services from upper layer users. This means that a major part of their functionality aims at de-

[9] It should be noted that we will later require that we have both an optimised as well as a conventional implementation available (see Section 8). We need to assume that when the processing of a stream of packets is taken over by the optimised implementation there are no messages left in the inter-layer queues of the conventional implementation. Furthermore, we need to assume that the SDL processes are always in suitable states so that they can always accept a packet originating from an adjacent layer process.

tection and treatment of many kinds of exceptions and errors. Exceptions and errors, however, are usually uncommon, in particular in typical high speed communication environments. On the other hand, optimising the common case implies that we need to take care of uncommon cases using alternate non-optimised error-case implementations. But, as we argued above, because of the low probability of these error handling cases we can tolerate the non-optimised processing of these error cases without risking a considerable degradation of the performance of the protocol. However, not all branching in the control flow can be classified so that *one* branch is common and *all others* are uncommon. It may as well be the case that more than one alternative is a common choice, namely when the branching does not aim at handling exception cases.

Now, what does the term *common case* mean technically? We distinguish the decision edges (outgoing *cfd*-edges of a node with outdegree > 1) of the *cfd* relation of an MLDG M disjointly into those which are taken with a probability above a certain fixed threshold value (the *common* ones, labeled with 'C') and those for which the probability is below this threshold value (the *uncommon* ones, labeled with 'U', see Section 5.1). The labeling defines a *common path graph* which is a subgraph of the *cfd* graph. Hence, our further optimisation will only address the common way a packet takes through the protocol stack, along a common path, and not the uncommon cases. In order to obtain what we call the *Common Path Graph* (CPG) we drop those subgraphs of M which start with an edge labeled as uncommon from every decision node (see Section 5.2).

5.1 Labeling of MLDGs

Common/Uncommon Labeling of MLDGs. Let M denote an MLDG and let $C = \{C, U\}$ a set disjoint from any other set in sight. Furthermore let $cul \subseteq (branchedges(S, cfd) \times C)$ a functional relation. We say that *cul* is a *common/uncommon labeling* of the MLDG M. Figure 6 shows a common / uncommon labeling for the example. Note that the labeling of the branching edges yields a tree which represents the normal way the packet takes through the protocol stack from an entry to an exit point. This normal path is common to many packets, therefore the name. The tree is identified in the Figure by bold solid line arrows.

Discussion. Whether a decision edge is common or uncommon depends in part on the environment in which a protocol is running. The common / uncommon attributes can thus not be automatically derived from the protocol specification. The attribution has to be provided by the implementor as an input for our method, based on common sense understanding of the protocol behaviour, or alternatively based on automated code analysis (see for example [2]).

5.2 Common Path Graph (CPG)

Given an MLDG M we now describe an algorithm to remove those subgraphs that depend on an uncommon decision in M. Technically, this means that we drop those subgraphs of M which start with an edge labeled as uncommon from every decision node.

Algorithm for the Construction of the CPG. Let M an MLDG and let cul_M the corresponding common / uncommon labeling. The algorithm for the construction of the common path graph C_M is as follows:

Algorithm 2

I. $C_M := M$
II. FOR ALL $x \in domain(domain(cul_M \rhd \{U\}))$
  III. $C_M := mlprune(C_M, x)$

Example CPG. In Figure 7 we present the CPG derived from the common / uncommon labeled MLDG in Figure 6. The subgraph starting with the edge $(D1, S5)$ has been removed. The subgraphs starting in node $D2$ have both been retained as they both represent common branches of a decision. Also, the TDG starting in node $S6$ has been removed as it has no edge belonging to the common path.

Remark. However, the result of the dependence analysis in Section 3 has been a set $\mathcal{M}$ of MLDGs, whereas this Section only addresses the determination of a CPG based on a single MLDG. We expect that the user decides which elements of $\mathcal{M}$ he wishes to be optimised by the later optimisation steps, based on a similar common/uncommon decision as we discussed in the context of the labeling of edges.

6 Construction of the Relaxed Dependence Graph

In the previous Sections we have shown how a common path graph (CPG) can be derived from an SDL specification based on a control and data flow dependence analysis. In this Section we will construct a relaxed dependence graph (RDG) which will be the starting point for later optimisation and implementation steps. The relaxation will be mainly a relaxation of the sequentiality constraints imposed by the sequential control flow dependence relations in the CPG. The implementations of the CPG will be, as we argue later, correct implementations of the original specification, but they will execute more efficiently than 'faithful' implementations. The relaxation will consist in the following steps:

Anticipation of the Common Case. Most nodes in the CPG depend on[10] a decision node, and normally nodes depending on a decision node can only be executed when

[10] For a node x to *depend on* a node y here means that x is in the transitive closure of the *cfd* relation restricted to y on its first component.

the last decision node on which they depend has been executed. Hence, decision nodes limit potential parallelism because they enforce an execution order. However, we have identified some nodes in the CPG which are of type *decision* but only have one outgoing *cfd* edge (cf. node `D1` in Figure 7), so they do not represent a decision along the common path. We therefore *anticipate* the outcome of this decision to be always the way which we predicted when determining the common path. We henceforth treat these decision nodes as nodes representing 'normal' statements, and as if no other node depended on their execution. Thereby we reduce the amount of linear sequential execution conditions. We call the resulting graph an *anticipated CPG*.

Parallelising. We relax the anticipated CPG such that we first strip away the *cfd* relation and only retain the *dfd* relation. However, we need to add some additional dependences which ensure that a node is never executed before the last decision node on which it depends in the *cfd* relation has been executed. The result is a *Relaxed Dependence Graph* (RDG). In the later implementation two statements can be executed in parallel iff they do not depend on each other in the RDG.

6.1 Anticipation of the Common Case

To enhance potential parallelism we anticipate the outcome of decisions that have only one outcome in the CPG, which means that we treat such decisions as if they represented nodes of type *task* instead of nodes of type *decision*. A successor of an anticipated decision can then be executed before the outcome of the decision is known. If the outcome corresponds to the anticipation we have a potential gain in parallelism. However, in the very few cases where our anticipation of the common outcome of a decision was wrong, e.g. an erroneous packet has been processed and the error was detected, then statements which have already been executed in anticipation of the common outcome of the decision may need to be undone (see Section 8 for a discussion).

Anticipation of the common case can be applied to a CPG using Algorithm 3. Given a CPG C, the algorithm selects all decision nodes from the set S_C that have only one successor in *cfd* (I.) and changes the type of these nodes to *task* (III.).

Algorithm 3

I. SELECT $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ SO THAT
$(\forall D_i)((sttype(D_i) = decision) \wedge (|\,\{D_i\} \lhd cfd\,| = 1))$
II. FOR ALL $D_i \in \mathcal{D}$ DO
III. $sttype(D_i) := task$

The result of the algorithm is a graph in which all nodes of type *decision* have more than one successor in *cfd*. All decision nodes are thus branching nodes as defined in 4.1.

Example. Anticipating the common case in our example results in changing the statement type of D1 from *decision* to *task*. When the sequential *cfd* dependences have been removed this will allow us to execute node `D1` in Figure 7 *after* node `S11`.

6.2 Relaxation of Dependences

In this transformation we remove the *cfd* dependences from the CPG in order to increase the potential for parallel execution[11]. More precisely, we remove all *cfd* edges, retain the *dfd* edges, and add some auxiliary dependences. We obtain a graph, called *relaxed dependence graph* (RDG), which has the same set of nodes as the anticipated CPG, but only one dependence relation on its nodes. We call this relation the *relaxed dependence* relation (*rxd*).

There are three types of precedence constraints which the relaxed dependence graph has to enforce:

- *Data flow dependences*: the data flow dependence relation as defined by the CPG has to be respected (a node using a variable may not be executed before a node which defines that variable).
- *Control flow dependences*: a node which is (directly or transitively) control flow dependent on a decision or root node may not be executed before the decision or root node has been executed[12].
- *Final execution of exit nodes:* Exit nodes must be the last nodes to be executed because they are the point where a protocol interacts with its environment and makes the result of the processing available to the environment. Thus all non-exit nodes must be forced to be executed prior to executing an exit node, and auxiliary dependences need to ensure this.

The Algorithm. Starting from an anticipated CPG C we create the RDG and its *rxd* relation in three steps. First, we include all elements of the original CPG's *dfd* relation in *rxd*. This will ensure that data dependences are respected in the RDG. Then we examine each node of the RDG to see if it already depends (directly or transitively) on its nearest preceding decision or root node in the original CPG's *cfd* relation. If not, we add a dependence between the examined node and that nearest decision or root node. This ensures that a node is not executed before the last decision it depends on is executed. Finally, we check whether all exit nodes reachable from a given node in the CPG are also dependent of that node in the RDG. If this is not the case, then we add relaxed dependences between the given node and the concerned exit nodes. This last step ensures that the exit nodes, which check whether the anticipations of common outcomes of decisions are justified for the respective packet or whether the execution has to be rolled back, are actually executed as last steps.

Algorithm 4 is the RDG construction algorithm. Starting with an anticipated CPG C it uses the cfd_C and dfd_C relations to create the *rxd* relation over $S_C \times S_C$ of the

[11] By parallel execution of the graph we more precisely mean the parallel execution of the implementation of the statements represented by the nodes of the graph.

[12] Note that some former decision nodes were anticipated in the anticipation step and that these node are now considered regular non-decision nodes.

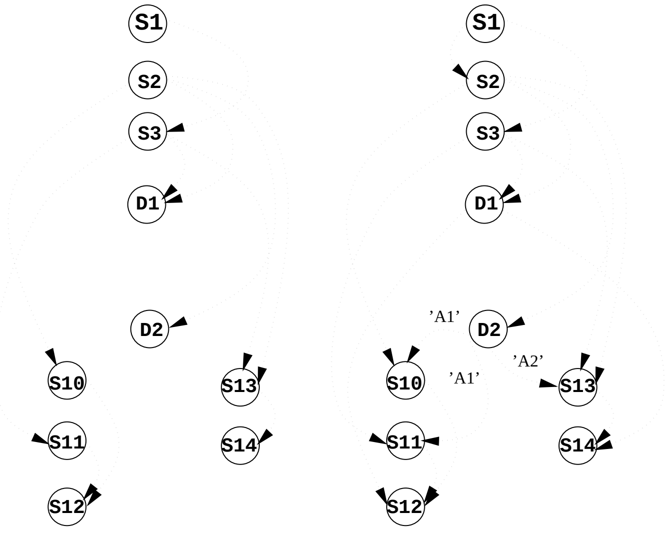

Figure 8: Control-flow dependence relaxed (left) and complete RDG (right)

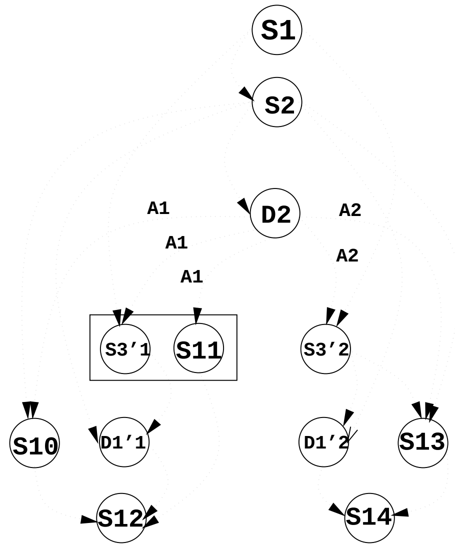

Figure 9: Dependence graph with grouped DMOs

resulting RDG. The algorithm first selects a set $\mathcal{D}$ of all decision nodes of the graph plus the root of the graph (I.). It includes all elements of dfd_C into rxd (II.). Then, for every node x of the graph, it finds the nearest node in $\mathcal{D}$ from which x is transitively dependent in the cfd_C relation (III.). If in the rxd relation x is not yet transitively dependent of that nearest node, a new auxiliary dependence is added. Next, all nodes except the exit nodes of the graph are examined. A dependence is added (V.) between an examined (VI.) node y and each exit node which is transitively dependent on y in the cfd_C relation of the CPG but not in the rxd relation of the RDG (VII. and VIII.).

Algorithm 4

I. SELECT $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ SO THAT
$(\forall D_i)(sttype(D_i) = decision \vee D_i = root(C))$
II. $rxd := dfd_C$
III. FOR ALL $n \in S_C - root(C)$
  IV. SELECT D_n SO THAT
  $\{s \in \mathcal{D} \mid (s, n) \in cfd_C^+ \wedge (D_n, s) \in cfd_C^+\} = \emptyset$
  V. IF $(D_n, n) \notin rxd^+$ THEN $rxd := rxd \cup \{(D_n, n)\}$
VI. FOR ALL $m \in S - leaves(C)$
  VII. FOR ALL $x \in leaves(C)$
    VIII. IF $(m, x) \in cfd_C^+ \wedge (m, x) \notin rxd^+$ THEN
    $rxd := rxd \cup \{(m, x)\}$

We call the resulting directed graph $R = (S_C, rxd)$ the relaxed dependence graph for CPG C. It should be noted that R is not a tree any more.

Example. Figure 8 shows the RDG for the anticipated CPG in Figure 7. The dependence graph on the left hand side of Figure 8 is the one obtained after executing step II of the algorithm when only the dfd relation of the original graph is retained. The complete RDG is shown on the right hand side of Figure 8. It is obtained by adding following auxiliary dependences: In order to ensure the dependence of nodes from their closest decision or root node the auxiliary edges $(S1, S2), (D2, S10), (D2, S11)$ and $(D2, S13)$ have been added. Additionally, in order to ensure that the exit nodes $S12$ and $S14$ are actually executed last, and to avoid the anticipated decision node $D1$ to be executed last, the auxiliary dependences $(D1, S12)$ and $(D1, S14)$ were added. We see that $S2$ and $S3$ depend both on $S1$, but they do not depend on each other. This means that once $S1$ has been executed $S2$ and $S3$ can be executed in parallel.

7 Optimisations based on the RDG

The RDG will be the basis for an implementation of the common path portion of the protocol stack. The run-time or compile-time scheduling of the operations in the RDG on a given hardware architecture is an important task of an implementation. When scheduling the operations, the scheduler may take advantage of the relaxation of dependences in the RDG. In particular, relative to other operations the execution of an operation may be scheduled more advantageously in an order different from the order prescribed in the original sequential SDL specification, and in parallel with other operations. A further gain in efficiency can be achieved by combining the execution of so-called *Data Manipulation Operations* (DMOs), which also depends on some freedom in the ordering of operations. In this Section we will discuss how this concept can be interpreted based on an RDG.

7.1 Grouping of Data Manipulation Operations.

We call data manipulation operations (DMOs) operations that manipulate entire data parts of protocol data units. 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 and therefore executes much faster than the non-combined execution of both operations. This has already been demonstrated in [7]. It is also central to the work reported in [8] and [1]. Particularly, it has been shown in [28] that in presence of decisions along the path of execution of a packet in the protocol stack it is advantageous to wait with the execution of DMOs until all decisions have been taken. At that point the set of DMOs to be executed is known and the DMOs can be combined. The technique of deferment of the execution of operations is often referred to as *lazy messages* in the literature.

In this Section we present an algorithm which manipulates the RDG such that it is possible to implement DMOs in a combined fashion. In order to enable joint execution of operations the RDG has to be modified. It has to be taken into account that when grouping the execution of two DMOs so that one operation depends on a decision higher up in the RDG than the other operation, the higher operation must be executed along every possible path through the RDG. It is thus necessary to distribute DMOs over the RDG.

Example. Let us assume that the operations $S3$ (`TASK Y!D:=f(X)`) and $S11$ (`TASK W!D:=k(Y!D)`) in the example (see Figure 8) are DMOs. In a real-world example $S3$ can be thought of as a translation routine translating every byte of the message `X` and assigning it to the data part of message `Y`, whereas $S11$ might be another such operation on the same data resulting in the data part of message `W`. The identification of DMOs as such is a manual task here, however, it is certainly possible to partly automate the detection of DMOs in the SDL specification.

We include all identified DMOs in an RDG in a set called $\mathcal{DMO}$, hence in this example $\mathcal{DMO} = \{S3, S11\}$. Note that it is not possible to combine two DMOs if there exists a node which is *rxd* dependent on one DMO, if at the same time the second DMO is *rxd* dependent on this node. Such a node would clearly have to be executed after the first DMO but before the second, thus defeating the combined execution of the DMOs. For two DMOs to be executed at the same time, all decisions on which their execution depends must have been taken before the combined execution can be permitted. In our example, even if $S3$ does not depend on $D2$ we would nevertheless have to execute $S3$ after $D2$ because only then we know whether $S11$ will need to be executed at all. To make sure that $S3$ is executed after $D2$ we modify the RDG so that $S3$ depends directly on $D2$ rather than on $S1$ (which is the node it is originally depending on in the RDG).

We have, however, to take into account that $S3$ will have to be executed independently of the evaluation of the decision node $D2$. We therefore need to *"distribute"* $S3$ over all possible evaluations of $D2$ or, more precisely, over all subgraphs with root note $D2$. Distributing a DMO over the possible evaluations of a decision predicate means that we make one copy of the node representing the DMO for each possible outcome of the decision. In our example there will be two copies: one corresponding to the `'A1'` evaluation of $D2$ (we will call this new node $S3'_1$), and one corresponding to the `'A2'` evaluation ($S3'_2$). If $D2$ evaluates to `'A2'` we can execute a combined DMO $S3'_2/S11$. If $D2$ evaluates to `'A1'`, then we execute $S3'_1$ alone. The subgraphs depending on a distributed DMO also need to be distributed. In our example $D1$ depends on $S3$ and not on $D2$ which means that $D1$ needs to be executed after $S3$ for any evaluation of the decision $D2$. Thus we will create two copies of $D1$, denoted $D1'1$ and $D1'2$, which will each depend on one of the copies of $S3$.

7.2 An Algorithm for Grouping of DMOs

We propose a recursive algorithm that starts at the root of a given RDG (see Algorithm 5). The algorithm is first applied to the root node of a RDG $C = (S_C, rxd)$ and then recursively to the whole rest of the graph. The algorithm also takes as input the *cfd* relation of the CPG from which the RDG C was originally derived. This will help in determining the different possible evaluations of decisions. Let R be the name of the node which the algorithm is currently applied to. Note that the decision nodes in an RDG form a tree if the nodes in between decision nodes are eliminated and replaced by an edge. The algorithm distributes the DMOs that depend transitively on R over each decision node depending transitively on R (we refer to any one of these as D) if and only if other DMOs exist which can only be executed after D. The algorithm is then recursively applied to all decisions D which depend on R.

Starting from a node R, the algorithm is applied to all subgraphs selected for each possible evaluation of the decision R. These evaluations are identified by the direct successors E_R of R in *cfd* (step I.). For an arbitrary node X, *rleaves*(X, C) is the set of leaf nodes in C that can possibly be executed (reached) after node X. For the root R of C, *rleaves*(R, C) contains all leaves of C. For a node that depends on a decision, the set of reachable leaf nodes is smaller. *rleaves*(X, C) is equal to *rleaves*(Y, C) iff X and Y can only be executed after the same number of decisions have been executed with the same evaluation. *rleaves*$(X, C) \subset$ *rleaves*(Y, C) iff X can only be executed after one or more decisions have been evaluated after Y. In step II $\mathcal{B}$ is selected to be the set of DMOs that can be executed in the subgraph selected by E_R but only after one or more decisions of that subgraph have been evaluated. If this set is not empty then D, the next decision to be executed in the subgraph, is identified (IV.). In step V $\mathcal{A}$ is selected to be a set of DMOs that can be executed in the subgraph selected by E_R *before* D is executed. These are the DMOs that we want to distribute over D. The second part of the selection condition makes sure that there is a DMO B in $\mathcal{B}$ with which each A_i can actually be combined. This is only possible if there is no node X which is *rxd* dependent on A_i so that B is *rxd* dependent on X. In step VI the set $\mathcal{S}$ of nodes to be distributed over D is selected. It contains $\mathcal{A}$ plus all nodes that depend on any DMO in $\mathcal{A}$ but not on D. These nodes are then duplicated for all possible evaluations of D (VII.). A graph $C' = (\mathcal{S}', rxd')$ containing a copy of the nodes to be duplicated aswell as the *rxd* edges between them is created together with a bijection f that relates original nodes to their duplicates (VIII.). The set DMO is adjusted to also contain the duplicated DMOs (IX.). In steps X and XI the duplicated graph is added to the original graph. In step XII additional dependences are added to connect the duplicated graph to the original graph. These are a) a dependence between D and every DMO, b) dependences from predecessors of nodes in S to the corresponding duplicate node in $\mathcal{S}'$ and c) dependences between nodes of $\mathcal{S}'$ and the successors of corresponding nodes in $\mathcal{S}$ as long as the successors are executed for the evaluation of D identified by E_D. Once the nodes selected in $\mathcal{S}$ have been duplicated for each evaluation of D, they are removed from the graph (XIV.). The algorithm is then recursively applied to D (XV.).

Algorithm 5

RecursiveCombine(R)

I. FOR ALL $E_R \in range(\{R\} \lhd cfd)$

II. SELECT $\mathcal{B} = \{B_1, \ldots, B_l\} \subseteq \mathcal{DMO}$ SO THAT $(\forall i)(rleaves(B_i, C) \subset rleaves(E_R, C))$

III. IF $\mathcal{B} \neq \emptyset$

IV. SELECT $D \in branchnodes(C)$ SUCH THAT $rleaves(D, C) = rleaves(E_R, C)$

V. SELECT $\mathcal{A} = \{A_1, \ldots, A_m\} \subseteq \mathcal{DMO}$ SO THAT $(\forall i)((rleaves(A_i, C) = rleaves(E_R, C))$ $\wedge (\exists B \in \mathcal{B})(\neg \exists X \in S_C)(((A_i, X) \in rxd^+) \wedge ((X, B) \in rxd^+)))$

VI. SELECT $\mathcal{S} = \{S_1, \ldots, S_n\} \subseteq S_C$ SO THAT $(\forall i)((S_i \in \mathcal{A}) \vee (((domain(rxd^+ \rhd \{S_i\}) \cap \mathcal{A}) \neq \emptyset) \wedge ((D, S_i) \notin rxd^+)))$

VII. FOR ALL $E_D \in range(\{D\} \lhd cfd)$

VIII. LET $C' = (\mathcal{S}', rxd')$ A GRAPH AND LET $f : \mathcal{S} \mapsto \mathcal{S}'$ A BIJECTION SUCH THAT $(\forall S_i, S_j \in \mathcal{S})((f(S_1), f(S_2)) \in rxd'$ IFF $(S_1, S_2) \in rxd)$

IX. $\mathcal{DMO} := \mathcal{DMO} \cup range(\mathcal{DMO} \lhd \{)$

X. $S_C := S_C \cup \mathcal{S}'$

XI. $rxd := rxd \cup rxd'$

XII. $rxd := rxd \cup \{(D, X) \mid X \in (\mathcal{S}' \cap \mathcal{DMO})\} \cup$ $\{(\mathcal{X}, \{(\mathcal{Y})) \mid (\mathcal{X}, \mathcal{Y}) \in rxd \wedge \mathcal{X} \notin \mathcal{S} \wedge \mathcal{Y} \in \mathcal{S}\} \cup$ $\{(\{(\mathcal{X}), \mathcal{Y}) \mid (\mathcal{X}, \mathcal{Y}) \in rxd \wedge \mathcal{X} \in \mathcal{S} \wedge \mathcal{Y} \notin \mathcal{S} \wedge$ $rleaves(\mathcal{Y}, \mathcal{C}) \subseteq rleaves(\mathcal{E}_\mathcal{D}, \mathcal{C})\}$

XIII. NEXT E_D

XIV. $S_C := S_C - \mathcal{S}$; $rxd := rxd - \{(X, Y) \mid (X, Y) \in rxd \wedge (X \in \mathcal{S} \vee Y \in \mathcal{S})\}$

XV. RecursiveCombine(D)

XVI. NEXT E_R

The algorithm creates groups of DMOs that are selected by the same evaluation of a decision. The actual rewriting of the DMOs into one complex DMO is done during implementation (see for example [1]).

Example. An application of the algorithm to our example is exemplified in Figure 9. We defined two nodes, namely $S3$ and $S11$, to be DMOs. $S3$ is duplicated for each evaluation of $D2$, yielding $S3'1$ and $S3'2$. If $D2$ evaluates to `'A1'` then a combined DMO $S3'1/S11$ can be executed. If $D2$ evaluates to `'A2'`, then $S3'2$ is executed alone. $D1$ originally depends on $S3$ and therefore has to be executed for any evaluation of $D2$. Consequently we also duplicate $D1$, yielding $D1'1$ and $D1'2$.

8 Implementing the Optimised Graph

In the previous Sections we have shown how a multi-layer dependence graph (MLDG) can be derived from an SDL specification, how a common path graph (CPG) can be extracted from the MLDG, and how this CPG can be transformed into a relaxed dependence graph (RDG). This Section addresses final aspects of the method, namely the implementation of the considered protocol stack based on the derived RDG.

Implementing the RDG means that we map the statements corresponding to each node to a set of software- or hardware instructions. When performing this mapping we have to consider the following three aspects: First, we have to respect the *ordering constraints* on the operations as specified by the *rxd* relation of the RDG. Second, assuming the availability of parallel processing resources, the operations have to be *scheduled* on the hardware resources according to the ordering constraints, the qualitative resource requirements, and the expected time consumption of every operation on particular hardware components. Finally, we have to take care of the fact that the RDG only describes the common case of packet processing, i. e. we need to provide for an alternate processing when a packet belongs to an uncommon case. This includes assuring that the system is in a consistent state after a packet has been detected not to comply with the common case.

8.1 Preserving Ordering Constraints

The RDG imposes a set of ordering constraints on the operations to be executed. In general, this is a partial order. If we look back at the RDG in Figure 8 it is easy to see that for the subset $\{D2, S10, S11\}$ of operations the following partial order, expressed informally in terms of a process algebra like behavior expression, is `(D2; (S10 || S11))`. Any interleaving trace derived from this expression is the trace of a valid implementation, e. g. the traces $(D2, S10, S11)$ and $(D2, S11, S10)$. However, for the exact derivation of an optimal implementation these possible interleavings do not provide sufficient information, in particular for the following two reasons.

- The operations may be executed in a machine environment with limited parallel processing resources, so the theoretical maximal possible degree of parallelism may not always be attainable. Also, the processing resources may not be homogeneous and certain operations may have particular requirements of the particular characteristics of the resources on which they are to be executed.
- Furthermore, operations are not atomic, as the interleaving model suggests, but they have a duration. This also means that they may be executed partly simultaneously, and one operation may be executed simultaneously with a sequence of different other operations. All relations that are valid for two or more convex intervals are possible for the operations in the RDG. However, for two operations A and B where B de-

pends on A we require that A has to be finished before B starts.

8.2 Scheduling

Concludingly, the RDG defines ordering constraints on the operations that need to be executed in an implementation. However, in order to come to an implementation the target hardware architecture also has to be taken into account. Assuming that the implementation is supposed to run on a parallel hardware architecture this leads to the problem of deriving an optimal schedule. The schedule does not only reflect the order of the execution of operations, but also answers questions about how long an operation will occupy a certain hardware component. Last, because we cannot assume that all parallel components of the hardware have equal qualitative characteristics, the schedule will also have to respect qualitative constraints, like which operation has to be executed on which hardware component.

8.3 Ensuring Consistency - Treatment of Uncommon Cases

The RDG we derived from the initial specification is based on the so-called common case assumption. This means that we assume that the packets processed inside the RDG all comply with the assumptions made to determine the common path through the protocol stack, e.g. that they are error-free, that they do not require exception handling, etc. As a consequence we anticipated the results of some of the decisions along the common path. This means that we presumed a certain evaluation of some of the decisions and removed dependences of statements depending on these presumed decisions. In other words, some operations have been decoupled from the decision predicates by which they were 'guarded' in the original specification. This may lead to inconsistent sequences of operation. For example, a division by zero may be executed concurrently with the test for non-zeroness of the respective operand if we assumed that non-zeroness is the common case. In the original specification of our example (see Figure 4) the execution of `D2` (through `S4`) depends on the evaluation of decision `D1` to `true`. However, in the RDG in Figure 8 `S4` does *not* depend on the evaluation of `D1`. This implies that `D2` may even be executed *before* `D1` is evaluated. A possible inconsistency can only be detected when the processing of a packet reaches an exit node.

Apparently, consistency ensuring mechanisms have to be applied. This leads to the following three requirements.

- First, as we argued before we need to have a faithful and complete backup implementation of the whole protocol stack available. The backup implementation covers all decisions, exception handling mechanisms etc. as foreseen in the original specification. It takes over control when the optimised implementation detects that a packet violates the common case assumption, namely if a test does not evaluate to the value which was anticipated during the common path determination.
- Second, because we saw that operations may be executed prior to the evaluation of a decision predicate by which they were originally guarded, all operations must be robust. This means that no matter when an operation is executed it is ensured that the system will not enter a failure state.
- Third, when the processing control is handed over to the classical implementation, the state of the system when the packet has entered the protocol stack through an entry node has to be reestablished. To ensure that this initial state can always be reestablished we suggest using the following mechanism.
  - We distinguish operations in reversible and irreversible operations. We claim that most operations are reversible, in particular operations reading data or copying data from one storage location into a register, modifying the data, and writing it to a second storage location. These operations are reversible (because the unmodified data is still available in the old location), and they can easily be undone when control needs to be transferred to the backup implementation.
  - All those operations which are irreversible, and we expect that this is only a minor part of all operations, need to be secured by a checkpointing mechanism. This means that the data which is affected by these operations will be checkpointed before the respective operation is executed. If not all decisions are evaluated in the way it was anticipated, i. e. the packet is not processed according to the common case, the checkpoint information can be used to undo all irreversible operations.

Discussion. It arises the justified question how advantageous our optimisation is in light of these time consuming consistency ensuring mechanisms. We assume that the resetting to the initial state only occurs very infrequently, namely when an uncommon case has been reached. This holds in particular in high speed communication protocols where error rates are low, and flow control mechanisms are very often omitted. Also, we expect that only very few operations in high speed protocols are irreversible and require a checkpointing for the state of the protocol stack. However, when uncommon cases occur more and more often it is clear that there will be a break-even point between the efficiency gain due to the parallel and resequenced operation, and the resource consumption for consistency ensuring mechanisms.

8.4 Case Study: an IP/TCP/FTP Protocol Stack

In [21] we presented the application of our method to the SDL specification of an IP/TCP/FTP protocol stack. We first mapped operations or sequences of operations in the protocol stack to statements in the SDL specification. (The granularity of the resulting set of operations in the SDL specification greatly influences the complexity of the dependence graphs). We identified 21 statements (operations and decisions) in the specification. Some of the operations were procedure calls which hid more complex operations. We determined a common path, constructed a dependence graph, and determined a relaxed dependence graph.

Based on the relaxed dependence graph we combined two DMOs, namely the TCP checksum calculation, and the translation from internal into external ASCII representation inside the FTP layer. We scheduled the operations on a hardware architecture with limited parallelism which consisted of independent *medium* and *host* interface components, two *FIFO queues* feeding the interfaces, a special purpose *Data Manipulation Unit*, a general purpose *microprocessor*, and a *random access memory* unit. We assigned resource consumptions and qualitative resource constraints to each of the operations, and applied an enumerative scheduling algorithm to this problem.

In the optimal schedule the DMOs were executed jointly, and in parallel with other operations (both DMOs were scheduled to be executed on the data manipulation unit, whereas the other operations were executed in parallel on the microprocessor). The optimal schedule would have been executed within 414 process cycle time units, whereas the strictly sequential execution of the packet processing along the common path in a 'faithful' fashion according to the SDL specification would have taken 1018 processor cycle time units.

9 Conclusions

9.1 Recapitulation

We considered a method for the derivation of optimised, parallel implementations from SDL specifications of protocol stacks. We argued that the efficiency of the protocol processing is crucial and that it is inefficient to implement SDL specifications of protocol stacks 'faithfully'.

To overcome this deficiency we presented formalisations and algorithms for the derivation of optimised protocol implementations from SDL specifications. We started with a syntactical data- and control-flow dependence analysis of SDL processes. Then we assumed that packets are processed in a sequence of steps on their way through the protocol stack and we removed the boundaries between different protocol layers by showing how multiple dependence graphs can be combined to multi-layer dependence graphs. Next we determined a so-called common path graph (CPG), a subgraph of a multi-layer dependence graph which represents the common steps of processing of a packet within the protocol stack. This allowed us in the next step to anticipate the evaluation of some decision statements in the CPG, and then to relax the dependences inside the graph by abstracting away from the sequential control flow dependences. As a consequence we only retained data flow dependences and dependences that express the dependence of a statement from the evaluation of a decision predicate. Based on this relaxed dependence graph (RDG) we were then able to perform a grouping of data manipulation operations, as described in Algorithm 5. Our algorithm is an extension of work described in [28] in the sense that a) our algorithm only delays operations as far as necessary to combine the operations (unlike until the last moment of the processing of a packet as in [28]), b) it takes into account that some DMOs can not be combined due to dependences to intermediate operations, and c) it is applied at compile time thus yielding better performance than when dynamically combining operations. A compile-time approach has been proposed in [1], however, this proposal does not address the treatment of decisions and data- and control flow dependences. The resulting graph finally acted as a basis for the implementation of the protocol stack, subject to the solution of a scheduling problem.

In general, implementing the RDG means that we map the statements corresponding to each of its nodes to a set of software instructions or hardware modules, subject to the following three conditions. First, we have to preserve the ordering constraints imposed by the RDG. Second, assuming the availability of parallel processing resources the operations have to be scheduled according to the ordering constraints, the resource requirements and the expected time consumption of every operation. Finally we have to take care of the fact that our RDG only addresses the common case, i. e. we need to solve the problem of the alternate processing when a packet belongs to the uncommon case. We discuss some of the implementation aspects in [21] (see also Section 8).

The fact that we have provided a rigorous formal description of our method clearly supports the implementation of our algorithms in a comprehensive toolset. It also connects our method well to other formally supported steps in telecommunications systems engineering, like testing, verification and validation.

Finally, we should mention that although the first steps of our method (generation of the TDGs and MLDGs) are tailored to the use of SDL as specification formalism it could be easily adapted to layered protocol specifications based on other specification methods. The later steps, however, are independent of the choice of a specification language.

9.2 Perspective

Improved Message Flow Graph Analysis. We made significant facilitating assumptions concerning the assumed message flows between processes in our model. These were in particular the assumption that one sending of a signal corresponds to exactly one receiving of that signal by the partner process. The method will gain a lot in flexibility if more sophisticated message flows can be treated.

Lateral Communication. In our method we have so far assumed that the processing of a packet is a non-interrupted sequence of operations from the point where the packet enters the protocol stack, to where it exits. We have not treated effects of lateral communication, namely when processes exchange control data like flow control information in addition to the protocol data we considered. Each such lateral communication would entail in our model an exit point from the protocol stack, and many exit points reduce the possible efficiency gain of our method considerably.

Tool Support. We have developed a tool set called OpParIm to support the method described in this paper [21]. The tool uses a Yacc/Lex based SDL parser in order to derive TDGs from SDL specifications, then allows for the user-guided execution of our optimising algorithms based on the derived TDGs, and finally yields an RDG. Finally, the derivation of an optimal schedule based on the RDG, the resource constraints, and the hardware architecture can be automated. The optimal schedule which we proposed in [21] for the IP/TCP/FTP protocol stack example was generated automatically. However, automated scheduling is not yet part of the OpParIm toolset.

Acknowledgements

We wish to thank Reinhard Gotzhein and Peter Ladkin for their helpful commentary on an earlier version of this manuscript.

References

[1] M. Abbott and L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5), October 1993.

[2] M. Abrams, N. Doraswamy, and A. Mathur. Chitra: Visual ananlysis of parallel and distributed programs in the time, event, and frequency domains. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):672–685, 1992.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

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

[5] T. Braun and M. Zitterbart. Parallel transport system design. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[6] ITU-TS (formerly CCITT). Recommendation Z.100: Specification and Description Language (SDL). ITU, Geneva, 1992.

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

[8] 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.

[9] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[10] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE Network Magazine*, pages 20–24, January 1992.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[12] S. Fischer and B. Hofmann. An Estelle compiler for multiprocessor platforms. In R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors, *Formal Description Techniques, VI*, IFIP Transactions C, Proceedings of the Sixth International Conference on Formal Description Techniques. North-Holland, 1994.

[13] M. Goldberg, G. Neufeld, and M. Ito. A Parallel Approach to OSI Connection-Oriented Protocols. In *Proceedings of the 3rd IFIP Workshop on Protocols for High-Speed Networks*, Stockholm, Sweden, May 1992.

[14] M. Heddes and E. Rütsche. A survey of parallelism in communication subsystems. Research Report RZ 2570, IBM Zurich Research Laboratory, 1994.

[15] D. Hogrefe. SDL and OSI: On the use of CCITT-SDL in the context of OSI. Habilitation Thesis, University of Hamburg, 1989.

[16] N. C. Hutchinson and L. L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[17] M. Ito, L. Takeuchi, and G. Neufeld. A Multiprocessing Approach for Meeting the Processing Requirements for OSI. *Journal on Selected Areas in Communications*, pages 220–227, February 1993.

[18] A. S. Krishnakumar and K. Sabnani. VLSI implementation of communication protocols - a survey. *IEEE Journal on Selected Areas in Communications*, 7(7):1082–1090, September 1989.

[19] P.B. Ladkin and B.B. Simons. *Static Analysis of Interprocess Communication.* Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[20] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering.* Doctoral dissertation, University of Berne, Switzerland, December 1994.

[21] S. Leue and Ph. Oechslin. OpParIm – A Method and Tool for Optimized Parallel Protocol Implementation. Full paper version of [24], submitted for publication, June 1995.

[22] 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.

[23] S. Leue and Ph. Oechslin. From SDL specifications to optimized parallel protocol implementations, extended abstract. In M. Ito and G. Neufeld, editors,

Proceedings of the 4th International IFIP Workshop on Protocols for High Speed Networks. Chapman & Hall, 1994. To appear.

[24] 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, Lisbon, September 1993.

[25] M. T. Liu. Protocol engineering. In M. C. Yovitis, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, Inc., 1989.

[26] A. Mitschele-Thiel. On the integration of model-based performance optimization and program implementation. In *Proceedings of the Fourth IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 196–202, Lisbon, September 1993.

[27] Ph. Oechslin. *Implémentation Optimisée de Protocoles à Hauts Débits.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1995. In French.

[28] 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.

[29] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[30] W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM TOPLAS*, 21(3):399–442, 1991.

[31] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–31, February 1990.

[32] Y.H. Thia and C.M. Woodside. High-speed OSI protocol bypass algorithm with window flow control. In B. Pehrson, P.Gunningberg, and S. Pink, editors, *Protocols For High-Speed Networks III*, IFIP Transactions C, volume 9, pages 53–68. North-Holland, 1993.

[33] K. J. Turner, editor. *Using Formal Description Techniques.* John Wiley & Sons, 1993.

[34] C. M. Woodside and R. G. Franks. Alternative software architectures for parallel protocol execution with synchronous IPC. *IEEE/ACM Transactions On Networking*, 1(2):178–186, April 1993.

Appendix

Notation and Definitions

Relations. Let $f \subseteq R \times R$ denote a binary relation over a set R, let $x, y \in R$ and S a set. We define the following *restrictions* and *operators* on a relation f.

$$f \rhd S \ \triangleq\ \{(a,b) | (a,b) \in f \ \wedge\ b \in S\}, \text{ and}$$

$$S \lhd f \ \triangleq\ \{(a,b) | (a,b) \in f \ \wedge\ a \in S\}.$$

$$\mathit{domain}(f) \triangleq \{a \mid (\exists b \in R)((a,b) \in f)\},$$

$$\mathit{range}(f) \triangleq \{b \mid (\exists a \in R)((a,b) \in f)\},$$

$$\text{and } \mathit{field}(f) \triangleq \mathit{domain}(f) \cup \mathit{range}(f)$$

A relation f is *functional* if and only if each element in its domain is related to a unique element in its range. For a functional relation f and an $x \in R$ we sometimes write $f(x)$ to denote $\mathit{range}(\{x\} \lhd f)$. We use f^+ to denote the transitive closure of a relation f, and f^* to denote the transitive reflexive closure of f.

Digraphs and Trees. Let V denote a set and let $E \subseteq V \times V$, then we call $T = (V, E)$ a *digraph.* We call T a *tree* if and only if the following additional conditions hold: $(\exists v \in V)((E \rhd \{v\} = \emptyset)) \wedge (\forall w \in V, w \neq v)(E \rhd \{w\} \neq \emptyset))$ (we call v the *root*), $(\forall v, w \in V)((E \rhd \{v\} = \emptyset) \rightarrow (v, w) \in E^+)$ (all nodes are reachable from the root), $E^+ \cap E^* = \emptyset$ (there are no cycles), and $(\forall v \in V)(\mid \{v\} \lhd E \mid \leq 1)$ (every node except for the root has exactly one predecessor). Furthermore, for a tree $T = (V, E)$ we define: $\mathit{root}(V, E) \triangleq \{v \in V \mid E \rhd \{v\} = \emptyset\}$, $\mathit{leaves}(V, E) \triangleq \{v \in V \mid \{v\} \lhd E = \emptyset\}$, $\mathit{branchnodes}(V, E) \triangleq \{v \in V \mid (\mid \{v\} \lhd E \mid) > 1\}$, and $\mathit{branchedges}(V, E) \triangleq \mathit{branchnodes}(V, E) \lhd E$. Furthermore, let $v \in V$, then we define $\mathit{rleaves}(v, (V, E)) \triangleq \mathit{range}(\{v\} \lhd E^+) \cap \mathit{leaves}(V, E)$.

Multi-edged and Labeled Trees.

- Let $E_1 \ldots E_n \subseteq V \times V$ for $n \geq 1$. Then we call $T = (V, E_1 \ldots E_n)$ a *multi-edged tree* iff (V, E_1) is a tree.
- Let $T = (V, E_1 \ldots E_n)$ a multi-edged tree. Let $D_1 \ldots D_n$ denote sets which are pairwise disjoint from any other set in sight. Let $L_1 \ldots L_n$ denote functional relations with $L_i \subseteq (E_i \times D_i)$. Then we call $T = (V, E_1 \ldots E_n, D_1 \ldots D_n, L_1 \ldots L_n)$ a *multi-edged labeled tree.* We shall slightly abuse notation in that we extend the notations $\mathit{root}(T)$ and $\mathit{leaves}(T)$ to multi-edged labeled trees, in the obvious way.

Operations on Trees.

- Let $T = (V, E)$ denote a tree and let $x \in V$. We define $T' \triangleq \mathit{prune}(T, x)$ iff $V' = V - \mathit{range}(\{x\} \lhd E^+)$ and $E' = E - (E \rhd \mathit{range}(\{x\} \lhd E^+))$.
- Let T denote a multi-edged labeled tree and let $x \in V$. We define $T' \triangleq \mathit{mlprune}(T, x)$ iff $V' = V - \mathit{range}(\{x\} \lhd E_1^+)$ and the following conditions hold for all i: $E_i' = E_i - (E_i \rhd \mathit{range}(\{x\} \lhd E_1^+))$ and $L_i' = L_i - (\mathit{range}(\{x\} \lhd E_1^+) \lhd L_i)$.

Glossary

CPG	Common Path Graph
cfd	control flow dependence
DMO	Data Manipulation Operation
dfd	data flow dependence
FTP	File Transfer Protocol
IOTDG	Input/Output Labeled Transition Dependence Graphs
IP	Internet Protocol
ITU-TS	International Telecommunications Union, Telecommunciations Standardisation Sector
MLDG	Multi-Layer Dependence Graph
OSI	Open Systems Interconnection
RDG	Relaxed Dependence Graph
SDL	Specification and Description Language
TCP	Transmission Control Protocol
TDG	Transition Dependence Graph

Stefan Leue (M'95 / ACM S'93 - ACM'95) received his Master's Degree in Computer Science (Diplom-Informatiker) from the University of Hamburg, Germany, in 1990, and his Ph.D. degree (Dr. phil.-nat.) from the University of Berne, Switzerland, in 1995.
From 1991 to 1995 he was a full-time research associate and doctoral candidate at the Department of Computer Science and Applied Mathematics of the University of Berne, Switzerland. Since March 1995 he is an Assistant Professor at the Department of Electrical and Computer Engineering of the University of Waterloo, Canada. His research interests are in the area of software engineering for telecommunications and distributed systems, in particular in the specification, verification, implementation of communication protocols and services. His email address is sleue@swen.uwaterloo.ca.

Philippe Oechslin (M'93) received his EE diploma in '91 from the Swiss Federal Institute of Technology in Lausanne (EPFL). In '91 he worked at the information systems research laboratory of AT&T Bell Labs in Holmdel where he designed an ATM layer interface chip. Back in Switzerland he got his Ph.D. degree from EPFL in '95 and now works as a senior research assistant at the communication networks laboratory of EPFL. His main interest is the "Web over ATM" project where interoperation between IP and ATM networks is explored as well as advanced WWW applications. His e-mail address is oechslin@di.epfl.ch