From SDL Specifications to Optimized Parallel Protocol Implementations

Stefan Leue[a] and Philippe Oechslin[b]

[a] Institute for Informatics, University of Berne, Länggassstrasse 51, CH-3012 Berne, Switzerland

[b] Computer Network Lab LTI, Swiss Federal Institute of Technology, DI-LTI EPFL, CH-1015 Lausanne, Switzerland

Keyword Codes: C.1.2; C.2.2; D.2.1; D.2.2
Keywords: Computer-Communication Networks, Network Protocols; Software Engineering, Requirements/Specification; Software Engineering, Tools and Techniques

Abstract

We propose a formalized method that allows to automatically derive an optimized implementation from the formal specification of a protocol. Our method starts with the SDL specification of a protocol stack. We first derive a data and control flow dependence graph from each SDL process. Then, in order to perform cross-layer optimizations we combine the dependence graphs of different SDL processes. Next, we determine the common path through the multi-layer dependence graph. We then parallelize this graph wherever possible which yields a relaxed dependence graph. Based on this relaxed dependence graph we interpret different optimization concepts that have been suggested in the literature, in particular lazy messages and combination of data manipulation operations. Together with these interpretations the relaxed dependence graph can be be used as a foundation for a compile-time schedule on a sequential or parallel machine architecture. The formalization we provide allows our method to be embedded in a more comprehensive protocol engineering methodology.

1. Introduction

Optimized protocol implementation has become an important field of research as network speed has increased much faster than computer processing power over the last decade. We present a method for the mainly automated derivation of efficient implementations of protocol stacks, starting from formal specifications. The rigor in the formalization is useful when implementing our method as a tool, which we are currently doing. In the paper we formalize and generalize optimization approaches that can be found in the literature, in particular in the literature on optimal protocol implementation.

Overview. In Figures 1 and 2 we present a (partial) view of the SDL specification of a two layer protocol stack consisting of two processes named `N` and `N+1`. It will serve as a running example in our paper and we will refer to it as TLS. It is the purpose of our optimization and implementation method to transform specifications similar to TLS into parallelized and optimized implementations. In Section 2 we discuss the sort of

G. Neufield et al. (eds.), *Protocols for High Speed Networks IV*

layered SDL specifications we consider in the paper. Here, we also argue why a direct and faithful implementation of SDL specifications would lead to inefficient implementations. This is mainly due to the structuring of SDL specifications into per-layer processes and the resulting inter-layer asynchronous queue based communication mechanism.

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. For the dependence graphs for example TLS see Figures 1 and 2. The dependence graph construction is an application of methods known from the domain of compiler optimization and parallel compilation as they are for example described in [10] and [3]. Control flow dependences relate directly successive statements (e. g. `S2` and `S3` in Figure 1) whereas data dependences relate statements where the depending statement uses a variable that is defined in the other statement (e. g. `S` and `D1` in Figure 1).

In the second step of our method we perform an optimization and parallelization of the operations which are caused by the processing of a packet. We consider the way the packet takes from the point where it enters the protocol stack to where it exits. Therefore we have to combine transition dependence graphs belonging to different SDL processes. We do so by eliminating the inter-layer communication statements, e. g. the statements `S4` and `S9` in the example TLS. The result is a Multi-Layer Dependence graph. We describe the construction in Section 3.1, for an example see Figure 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. For example, we assume that in the example TLS decision `D1` has one common and one uncommon branch, whereas decision `D2` has two common branches. This resulting graph is called *common path graph*, for an example see Figure 6. We will apply our later optimizations 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 where we assumed a common outcome. Henceforth we treat these decision nodes as if no other node depends on their execution. An example is decision `D1` (see Figure 6).

- *Parallelization:* We construct a relaxed dependence graph by taking 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). For example node `S10` is not data flow dependent on decision node `D2`, but still both nodes may not be executed in any order, because the execution of `S10` depends on the evaluation of `D2`. However, `S10` and `S11` are not data dependent and may thus be executed in parallel (meaning in any order).

Finally, in Section 7 we show how suggestions that have been made in the literature to optimize the implementation of communication protocols can be interpreted based on the relaxed dependence graph. We refer to the concepts of *Lazy Messages* (see [20]), and, in particular, *Grouping of Data Manipulation Operations* (see [7], [8] and [1]).

The optimized and parallelized graph now serves as a foundation for an implementation on either a sequential or a parallel machine architecture. The discussion of implementation aspects such as scheduling is outside the scope of this paper. We refer the reader to [17] and [15] for further discussion.

Related work. Aspects of hardware and software architecture that increase an implementation's efficiency are discussed in [7], [8], [20], [9] and [23]. Hardware implementations for high speed protocols have been proposed in [13]. Special attention has been paid to the parallelization of protocol implementations, so for example in [5] and [24]. However, the parallelization proposed in these papers depends entirely on the intuition of the designer and thus its efficiency may be non-optimal. Therefore automated support for the parallelization is desirable. An approach based on the scheduling of parallel tasks generated by an Estelle compiler is presented in [11]. In [19] the determination of data-flow dependence graphs for parallel implementations of stream processing programs on transputers are described. Others ([22], [14]) analyze the data- and message flow dependences between communicating processes, whereas we restrict ourselves to the analysis of dependences inside processes.

Precursors. Precursors of our work appeared in [17] where we describe the application of our method to a IP/TCP/FTP protocol stack. Further results will appear in [16]. More technical detail can be found in [15].

The role of SDL. The formal specification technique we consider is the CCITT standardized *Specification and Description Language* SDL [6]. We chose this language not because we particularly advocate its suitability as an implementation language, but rather because it enjoys wide acceptance in the protocol engineering community. For an overview of SDL see for example [4]. The choice of a formal description technique as starting point connects our method to existing techniques and methods in the domain of protocol engineering (see for example [18]). We may for example assume that as result of a previous verification step the specifications on which we base our optimization are dead- and livelock free. Also, conformance tests developed based on the formal specification can be directly applied to the implementation. Part of our method (dependence analysis and construction of multi-layer dependence graphs) are specific to features of SDL. However, we claim that for many other procedural specification methods an easy adaptation is possible. The later steps (starting with the CPG construction and down to the optimization steps we describe) are independent of the specification method on which the dependence graph is based.

2. A Discussion of SDL Specifications

2.1. SDL Specifications of Protocol Stacks

SDL is a Formal Description Technique frequently used in the specification of telecommunications systems, in particular for the layered specification of communications protocols. An SDL specification of a protocol stack can usually be structured into different concurrent processes, each one representing the functionality of one protocol layer. A process is structured into transitions which describe its dynamic behaviour. Processes communicate via asynchronous signal queues. In SDL the mapping of the output signals of the sending process to corresponding input signals of the receiving process is done using a relatively complicated mapping of signal names to signal routes, where the signal routes carry the sender and receiver identification information. For reasons of conciseness of the presentation we abstract away from this mechanism and identify sender and receiver of messages simply by identity of the message type. Thus, in the Example TLS the message `Y` sent out by process `N` is consumed by process `N+1`.

2.2. Inadequacy of 'Faithful' Implementations

By the term *faithful* 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 that the SDL specification is directly compiled so that every statement in the SDL specification is mapped to a statement in the implementation, that every SDL process corresponds to a process in the implementation, and 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 potentially inefficient.

- *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 general design decisions, determines the degree of parallelism of a specification. It also means that without optimizations 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]). The design of communication protocols is often governed by the principle that *'a good specification is a highly modular and layered specification'*. Though from a structured-design point of view a layered design may be desirable, we stipulate that in order to derive efficient parallel protocol implementations such a layered design is obstructive. This is mainly due 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 [9].

- *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 is very inefficient to implement the exchange of data in a non-distributed system via asynchronous queues. Instead, the protocol data will be stored in a local memory and the communication between the processes will be by shared variables.

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 in order to enable parallel and combined execution of statements belonging to different processes.

3. Dependence Analysis for SDL Processes

In this section we explain how a Dependence Graph can be obtained by syntactical analysis from an SDL specification. For a definition of the mathematical notation we use here and in later Sections see the Appendix. First we will explain how transitions as basic building blocks of SDL process specifications can be formalized and then how entire protocol stacks can be represented as graphs, based on the graphs representing the 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. For a complete description of the syntax of SDL transitions see [4]. The syntactic subset we have chosen is a concise subset of the full SDL syntax. It allows for the analysis of standard protocol specifications as presented in [4]. For the sake of conciseness we have limited our considerations to the language subset described above but we conjecture that an adequate treatment of most of these constructs is possible when extending our method.

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. This so-called dependence graph represents the two types of dependences between the statements of a specification, namely control flow and data flow 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 it 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.

A statement usually describes operations on process variables in which these are usually referenced in two different ways.

- 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.

- We say that a statement S_n *defines* a variable x iff it assigns an initial or new value to the variable without referencing its previous value. A typical example is the definition of a variable on the left hand side of an assignment statement.

A pair of statements (s_1, s_2) is in the *data flow dependence* relation *dfd* if (s_1, s_2) is in the transitive closure of the *cfd*-relation[1] 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[2]. 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[3].

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 $T = (S, STT, X, sttype, cfd, dfd)$ 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. $(\forall (v, w) \in dfd)(define(v) \subseteq use(w))$.

3.4. Example SDL processes and TDGs

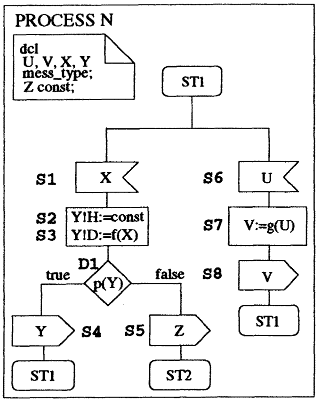
In the following examples we give the SDL specification of a transition in graphical representation (SDL-GR) on the left hand side, and equivalent specification in textual form (phrase representation, SDL-PR) in the middle of the chart, and a resulting dependence graph on the right hand side. We add labels `Sn` and `Dn` to help us to identify regular and decision statements, respectively. However, these labels are not part of the specification.

The example in Figure 1 shows a partial view of the specification of a process `N` of which we show only two transitions. 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 `DECISON` statement `D1` has possible successor statements

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

[2] This avoids additional *output* dependences, see [21].

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



```
    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;
     ENDDECISISON;
S6   INPUT(U);
S7   TASK V:=g(U);
S8   OUTPUT(V);
     NEXTSTATE ST1;
    ...
    ENDPROCESS N;
```

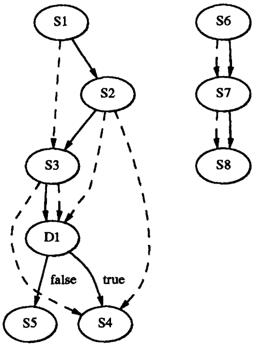


Figure 1. SDL-PR (left) and SDL-GR (middle) specifications and TDGs (right) for process N

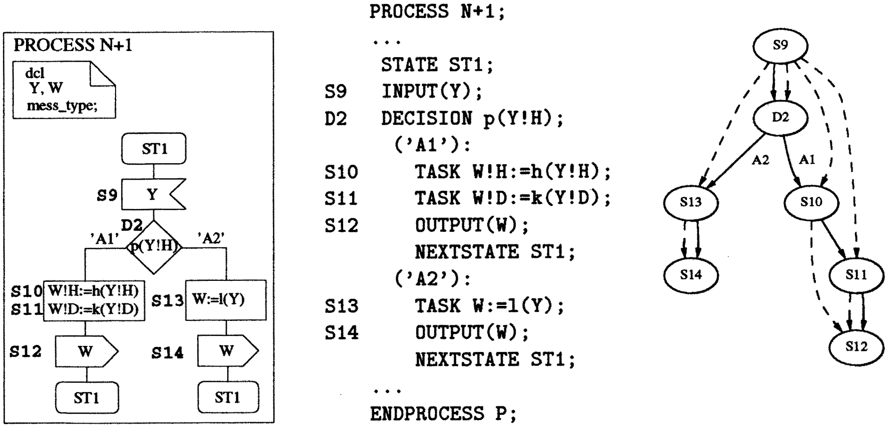
S4 and S5, the respective control flow dependence edges are labeled for illustrative purposes by *true* and *false*. The data flow dependences are so that S3 depends on S1 because of variable X, whereas D1 and S4 both depend on S2 and S3 because of the use of variable Y[4]. Figure 1 presents a graphical representation of this TDG which we call T_1, namely on the right hand side by the graph starting in node $S1$. *Solid* line arrows represent control flow dependencies, thus elements of *cfd*, and *dashed* line arrows represent elements of *dfd*. The dependences for the transition starting in statement S6 are obvious.

For our later argumentation, which aims at combining multiple processes to one process, we need a further example of an SDL process, namely the process named N+1 presented in Figure 2. The syntactical analysis leads to the transition dependence graph T_3 shown on the right hand side of Figure 2. The structure of the dependence graph is quite similar to the structure of the dependence graph for process N. It should also be noted that the decision D2 does not have a boolean evaluations, instead it evaluates to strings A1 or A2.

4. Dependence Graphs for Protocol Stacks

As we saw in Section 2 protocol stacks are usually specified by a set of independent concurrent processes. Each of these processes consists of a number of transitions. In Section 3 we described how to syntactically analyze each of these processes in order to

[4] One may envisage Y!H to stand for the header and Y!D for the data part of a protocol data unit or a packet.



```
     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;
     ...
     ENDPROCESS P;
```

Figure 2. SDL-PR (left) and SDL-GR (middle) specifications and TDGs (right) for process N+1

derive a set of transition dependence graphs for each process. 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 transitions. 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 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 formalize the concept of labeling of root and leave nodes of TDGs by the appropriate signal names as follows. Let $T = (S, STT, X, \mathit{sttype}, \mathit{cfd}, \mathit{dfd})$ 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: $\mathit{sttype}(\mathit{root}(I)) = \mathit{input}$, and $(\forall x \in \mathit{leaves}(I))(\mathit{sttype}(x) = \mathit{output})$.

Example IOTDG In Figure 3 we show the three IOTDGs representing the TDGs for Example TLS.

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 transition 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 algorithm 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 IOTGs (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 that 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.

Algorithm 1

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

Example MLDG Figure 4 shows the set $\mathcal{M}$ which we obtain by applying our algorithm to the IOTDGs of our example TLS. It contains two MLDGs, one with root `S1` and one with root `S6`. In order to illustrate the input/output labeling we also retained the respective labels at the nodes. 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 $(S2, D2)$.

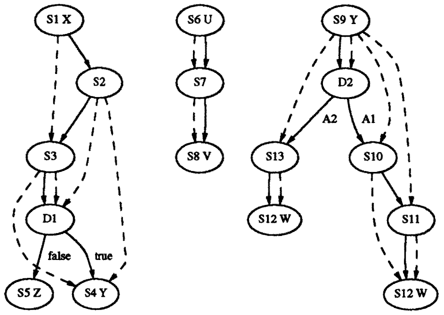


Figure 3. IOTDGs for Example TLS.

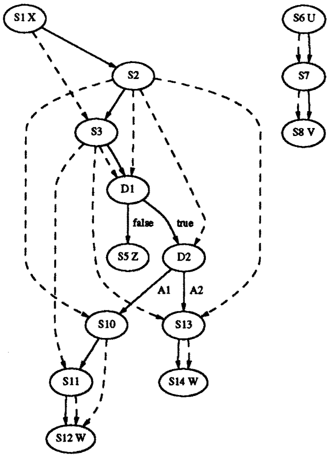


Figure 4. MLDGs for Example TLS.

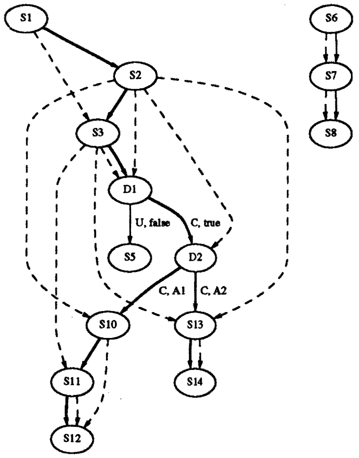


Figure 5. Labeled MLDG for Example TLS.

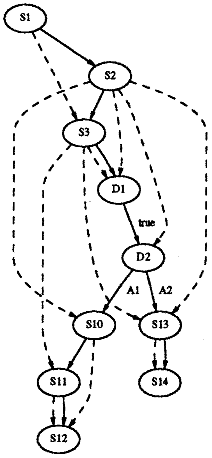


Figure 6. Common Path Graph for Example TLS.

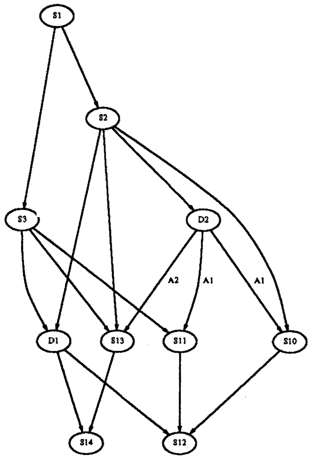


Figure 7. The Relaxed Dependence Graph for Example TLS

Justification for MLDG construction. When building the MLDG we modified the original SDL specification in two ways. Firstly, we ignored the asynchronous queue communication mechanism, and secondly, 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 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. The elimination of the `OUTPUT / INPUT` statement pair can be justified by the fact that we preserved all control flow and data flow dependences. Thus, the above argument concerning the safety properties now holds for all those statements which are direct predecessors or successors of the `OUTPUT / INPUT` statements that we eliminated. Concluding 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. Also, as opposed to work reported in [12] in the context of ESTELLE we do not eliminate the asynchronous queue communication mechanism between adjacent layer processes by replacing these processes by one product automaton because this would induce an extreme blow-up in the complexity of the implementation.

5. Determination of the Common Path Graph

The later steps of our optimization method rely on the assumption that we optimize the processing of a packet only for the 'common case' (we will come to a clearer understanding of this expression in this Chapter). In Chapter 6 we introduce optimization steps that anticipate certain common decision results according to a common case assumption. We consider our common path determination a generalization of the *Common Path* optimization 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 detection 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, optimizing the common case implies that we need to take care of uncommon cases using alternate non-optimized error-case implementations. But, as we argued above, because of the low probability of these error handling cases we can tolerate the non-optimized 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 that 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 value (the *common* ones, labeled with 'C') and those for which the probability is below a certain value (the *uncommon* ones, labeled with 'U'). The labeling of the decision edges is described in Section 5.1. It defines a *common path graph* which is a subgraph of the *cfd* graph. Hence, our further optimization 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.

Example Common/Uncommon Labeled MLDG. Figure 5 presents and example of a common / uncommon labeled MLDG. Note that the labeling of the branching edges yields a tree which represents the common path of the processing of a packet. This tree, which is indicated by bold solid line *cfd* edges in Figure 5, is obtained by traversing an MLDG so that no decision edge with label U is traversed.

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. One way of finding out which decisions are uncommon is to analyze a working implementation using for example a tool like it has been proposed in [2][5]. In case such analyses are not available it may be necessary to use simulation techniques or estimations in order to determine whether a particular decision edge belongs to the common or the uncommon case.

5.2. Common Path Graph (CPG)

The very easy algorithm for the construction of the CPG can be found in [15]. It simply prunes all those subtrees of the labeled MLDG which start with an edge labeled 'U' in a decision node.

Example CPG. In Figure 6 we present the CPG derived from the common / uncommon labeled MLDG in Figure 5. The subgraph that has been removed is the graph starting with the edge $(D1, S5)$. The subgraphs starting in node $D2$ have both been retained as they both represent common decisions.

6. Construction of the Relaxed Dependence Graph

In the previous chapters we have shown how a common path graph (CPG) can be derived from an SDL specification based on a control flow and data flow dependence analysis. In this Section we will construct a relaxed dependence graph (RDG) based on which

[5] The authors describe a tool called *Chitra* which analyzes program execution sequences yielding a semi-Markov chain model representing the time behavior of a program.

the original specification can be implemented. We conjecture that the implementation of the RDG will be functionally equivalent to the faithful implementation, but will execute faster. We propose the following steps to generate a RDG.

- *Anticipation of the common case:* In this step we ignore that certain statements depend on a decision, namely for those decisions where we assumed a common outcome. Henceforth we treat these decision nodes as if no other node depends on their execution.

- *Parallelization:* We construct a relaxed dependence graph by taking the data flow dependence relation of the anticipated 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. Two statements can be executed in parallel iff in the RDG they do not dependent on each other.

6.1. Anticipation of the Common Case

The CPG may contain decisions with only one outcome in the CPG. As we will see in the next transformation, decisions enforce an execution order, namely that a node must be executed after all decisions it depends on have been taken. Decisions thus limit potential parallelism. To enhance potential parallelism we anticipate the outcome of decisions that have only one outcome in the CPG. We treat such decisions as if they represented tasks instead of decisions. In [15] we discuss the handling of the uncommon cases in an implementation and argue that there is always a way to handle them consistently. Anticipation of the common case is applied to the CPG changing the type of those decision nodes which have only one successor in the *cfd* relation of the CPG to *task.* The algorithm can be found in [15].

In our example, anticipating the common case results in changing the statement type of D1 from*decision* to *task.*

6.2. Relaxation of Dependences

In this transformation we remove dependences from the CPG graph to allow its parallel execution. More precisely we remove all dependences and replace them by a smaller set of *relaxed* dependences. There are three precedence relations that the relaxed dependence graph must enforce. *Data flow dependences*: 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 of a decision node may not be executed before this decision has been taken. *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 visible. Thus all statements which are no exit nodes must be executed before executing an exit node. The result of the transformation is a relaxed common path graph (RDG) in which the *cfd* and *dfd* relations have been replaced by a relaxed dependence relation *rxd.* We create the *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. Then we examine each node of the RDG to see if it already depends (directly or transitively) from its nearest preceding decision node in the *cfd* relation. If not, we add a dependence between the examined node and the nearest decision node. This ensures that a node is not executed before the last decision it depends on. Finally we check that 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, we add relaxed dependences between the given node and the concerned exit nodes.

Algorithm 2 is an algorithm for the construction of the RDG. Starting with a given, possibly 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 resulting RDG. The algorithm first selects a set 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 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 node, a new 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 of y in the cfd_C relation of the CPG but not in the *rxd* relation of the RDG (VII. and VIII.).

Algorithm 2

```
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. $\mathrm{rxd} := \mathrm{dfd}_C$

III. FOR ALL $n \in S_C - \mathrm{root}(C)$

    IV. SELECT $D_n$ SO THAT
        $\{s \in \mathcal{D} \mid (s,n) \in \mathrm{cfd}_C^+ \wedge (D_n, s) \in \mathrm{cfd}_C^+\} = \emptyset$
    V. IF $(D_n, n) \notin \mathrm{rxd}^+$ THEN $\mathrm{rxd} := \mathrm{rxd} \cup \{(D_n, n)\}$

VI. FOR ALL $m \in S - \mathrm{leaves}(C)$

    VII. FOR ALL $x \in \mathrm{leaves}(C)$
        VIII. IF $(m,x) \in \mathrm{cfd}_C^+ \wedge (m,x) \notin \mathrm{rxd}^+$ THEN $\mathrm{rxd} := \mathrm{rxd} \cup \{(m,x)\}$
```

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. Figure 7 shows the RDG for the CPG in Figure 6. We see that S2 and S3 depend on S1 but not on each other. This means that once S1 has been executed S2 and S3 can be executed in parallel.

7. Optimizations based on the Relaxed Dependence Graph

An implementation will be based on the RDG. In particular, the scheduling of the operations on a given hardware architecture is a central task of any implementation, be it at compile- or at run-time. When scheduling the operations the scheduler may take advantage of the relaxation of dependencies in the RDG. The execution of an operation may be scheduled at a different point of time compared to its execution according to the sequential SDL specification. In particular, the scheduler may schedule the processing of certain operations whenever it seems optimal, for example when the required resources and data are available. A further gain in efficiency can be achieved by combining the execution of so-called *Data Manipulation Operations* (DMOs).

Grouping of Data Manipulation Operations. We call data manipulation operations (DMOs) operations that manipulate entire data parts of protocol data units. 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 thus executes much faster. 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 [20] that in presence of decisions along the path of execution of a protocol, it is better 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 is referred to as *lazy messages.* Our algorithm is a generalization of this technique. In order to enable the joint execution of DMOs 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.

An algorithm for grouping of DMOs. We propose a recursive algorithm that starts at the root of the RDG. Let B be the name of the node the algorithm is applied to. The algorithm distributes the DMOs that depend of B over each decision that depends of B, called B', iff other DMOs exist which can only be executed after B'. The algorithm is then recursively applied to all decisions B' that depend on B. Algorithm 3 does the operation described above. It is applied to the root node of a RDG C. It also takes as input the *cfd* relation of the CPG from which C was derived. The node it is applied to is called B. For each DMO D which depends of B (I.) a second DMO D_2 depending on an other decision node is searched in the subset of nodes that may be executed if D is executed (II.). If such a DMO exists, the decision node B' depending of B and leading to D_2 is found (III.). Then D is removed from the graph (IV.-VI.) and several copies of D, called D'_i, are created, one for each possible evaluation of B' (VII.-X.). Dependences are added from D'_i to all exit nodes which can be reached from B' with the corresponding evaluation of the predicate (XI.). Once all DMOs depending of B have been treated, the algorithm is applied to all decision nodes depending of B (XII. & XIII.). The algorithm stops when B has no more successors which are decision nodes.

Algorithm 3
RecursiveCombine(B)

```
I. FOR ALL $\{D \in \mathcal{DMO} \mid (B,D) \in \mathrm{rxd}\}$
    II. IF $\exists D_2 \in \mathcal{DMO} \mid (D,D_2) \in \mathrm{cfd}^+$ AND
        $\{s \in S_C \mid (D,s) \in \mathrm{rxd}^+ \wedge (s,D_2) \in \mathrm{rxd}^+\} = \emptyset$
        III. $B' = s \in \mathrm{branchnodes}(C) \mid (B,s) in \mathrm{rxd} \wedge (s,D_2) \in \mathrm{rxd}^+$
        IV. $S_C := S_C - \{D\}$
        V. $\mathcal{DMO} := \mathcal{DMO} - \{D\}$
        VI. $\mathrm{rxd} := \mathrm{rxd} - \{\{D\} \lhd \mathrm{rxd} \cup \mathrm{rxd} \rhd \{D\}\}$
        VII. FOR ALL $N_i \in \{B'\} \lhd \mathrm{cfd}$
```

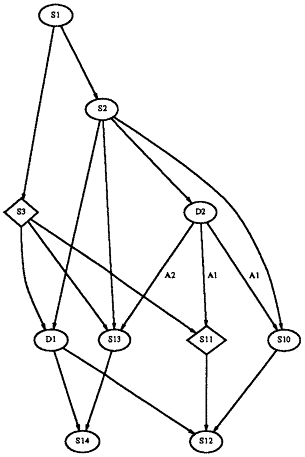


Figure 8. CPG with marked DMOs (diamonds)

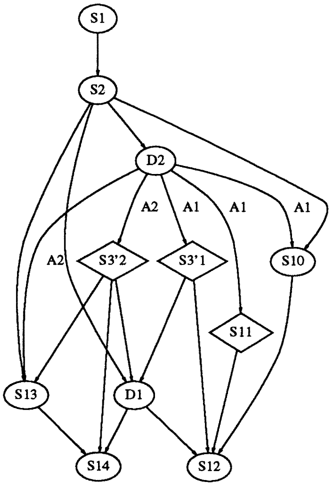


Figure 9. Grouped DMOs

```
        VIII. $S_C := S_C \cup \{D'_i\}$
        IX. $\mathcal{DMO} := \mathcal{DMO} \cup \{D'_i\}$
        X. $\text{rxd} := \text{rxd} \cup (B', D'_i)$
        XI. $\text{rxd} := \text{rxd} \cup \{(D'_i, x), x \in leaves(C) \mid (N_i, x) \in \text{cfd}^+\}$
XII. FOR all nodes $newB \in B \lhd \text{rxd}$ and $\text{sttype}(newB) = decision$
    XIII. call $\text{recursiveCombine}(newB)$
```

Example. The application of the algorithm to our example is shown in Figure 8 and Figure 9. The two DMOs identified are S3 and S11. S3 is replicated 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. In the final implementation the schedule of the operations has to be such that depending on the evaluation of the decision predicate D2 either S3'1 or S3'2 is executed before D1, but not both.

8. Conclusions

In this paper we presented formalizations and algorithms for the derivation of optimized protocol implementations from SDL specifications. We started with a syntactical dependence analysis for SDL processes. We then showed how multiple dependence graphs can be combined to multi-layer dependence graphs. Next we determined the common path graph, a subgraph of a multi-layer dependence graph which represents the common case of processing of a packet in the protocol stack. This graph was the basis for an optimization by anticipating the evaluation of some decision statements in the CPG, and then by relaxing the dependences. This essentially meant to omit control flow dependences and to only consider data flow dependences and dependences that express the dependence of a statement from the evaluation of a decision predicate. We called the result a relaxed dependence graph. When scheduling the operations on a given hardware architecture the scheduler may take advantage of the relaxation of dependencies in the RDG in particular by scheduling certain operations at a different point of time compared to the sequential execution in the SDL specification. In particular we showed how the optimization concepts of lazy messages and grouping of Data Manipulation Operations can be interpreted based on the Relaxed Dependence Graph.

We are currently developing a toolset for the support of our method. The toolset will consist in an SDL parser which generates dependence graphs, and a set of graph optimizing routines. The graph optimizing algorithms have already been implemented, the SDL parser is currently under development. Furthermore, we have implemented a prototype tool to support the scheduling aspect of the implementation. The fact that we have provided a rigorous formal description of our method clearly supports the implementation of such a toolset. It also connects our method well to other formally supported steps of an overall protocol engineering methodology, like testing and validation.

Acknowledgments. The work of both authors was supported by the Swiss National Science Foundation. We would like to thank Peter Ladkin for very helpful commentary on an earlier draft of this paper.

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, November 1992.
3. U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211-243, Feb 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. CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, 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. J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE*

Network Magazine, pages 20–24, january 1992.

10. 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*, pages 319–349, July 1987.
11. 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. To appear.
12. B. Hofmann and W. Effelsberg. Efficient implementation of Estelle specifications. Technical report Reihe Informatik, Nr. 3/93, University of Mannheim, Mannheim, Germany, 1993.
13. 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.
14. P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 248–259. ACM Press, 1992.
15. S. Leue and Ph. Oechslin. A formal approach to optimized parallel protocol implementation. Technical report, University of Berne, Institute for Informatics, Berne, Switzerland, 1994.
16. S. Leue and Ph. Oechslin. Formalizations and algorithms for optimized parallel protocol implementation. In D. Lee et al., editor, *Proceedings of the 1994 International Conference on Network Protocols ICNP-94*. IEEE Computer Society Press, 1994. To appear.
17. 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*, Lisbon, Sep. 1993. To apear.
18. M. T. Liu. Protocol engineering. In M. C. Yovitis, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, Inc., 1989.
19. A. Mitschele-Thiel. On the integration of model-based performance optimization and program implementation. In *4th Workshop on Future Trends of Distributed Computing Systems*, 93.
20. 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.
21. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.
22. W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM TOPLAS*, 21(3):399–442, 1991.
23. 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 C*, volume C-9, pages 53–68. IFIP, NORTH-HOLLAND, 1993.
24. 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.

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

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

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

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

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

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 $\mathit{range}(\{x\} \triangleleft f)$. 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:

- $(\exists v \in V)((E \triangleright \{v\} = \emptyset)) \wedge (\forall w \in V, w \neq v)(E \triangleright \{w\} \neq \emptyset))$ (we call v the *root*),
- $(\forall v, w \in V)((E \triangleright \{v\} = \emptyset) \rightarrow (v, w) \in E^+)$ (all nodes are reachable from the root),
- $E^+ \cap E^* = \emptyset$ (there are no cycles), and
- $(\forall v \in V)(| \{v\} \triangleleft E | \leq 1)$ (every node except for the root has exactly one predecessor).

Furthermore, for a tree $T = (V, E)$ we define: $\mathit{root}(V, E) \triangleq \{v \in V \mid E \triangleright \{v\} = \emptyset\}$, $\mathit{leaves}(V, E) \triangleq \{v \in V \mid \{v\} \triangleleft E = \emptyset\}$, $\mathit{branchnodes}(V, E) \triangleq \{v \in V \mid (| \{v\} \triangleleft E |) > 1\}$, and $\mathit{branchedges}(V, E) \triangleq \mathit{branchnodes}(V, E) \triangleleft E$.

Multi-edged and Labeled Trees.

- Let $E_1 \dots E_n \subseteq V \times V$ for $n \geq 1$. Then we call $T = (V, E_1 \dots E_n)$ a *multi-edged tree* iff (V, E_1) is a tree.
- Let $T = (V, E_1 \dots E_n)$ a multi-edged tree. Let $D_1 \dots D_n$ denote sets which are pairwise disjoint from any other set in sight. Let $L_1 \dots L_n$ denote functional relations with $L_i \subseteq (E_i \times D_i)$. Then we call $T = (V, E_1 \dots E_n, D_1 \dots D_n, L_1 \dots L_n)$ 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.