Formalizations and Algorithms for Optimized Parallel Protocol Implementation

Stefan Leue

Institute for Informatics
University of Berne
CH-3012 Berne

Philippe Oechslin

Computer Network Lab LTI
Swiss Federal Institute of Technology
CH-1015 Lausanne

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 Section 2 we discuss the sort of 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. Next 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* (Section 3). 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 [8] and [2]. In the next 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 construct a so-called Multi-Layer Dependence graph (Section 4). Third, we identify the path a packet takes through the protocol stack in the so-called common case. The resulting graph is called *Common Path Graph*. We will apply our later optimizations only to the common case part of the specification. Fourth, in Section 6 we relax dependences within the common path graph. This is accomplished in two steps, first the *anticipation of the common case* along the common path (Section 6.1), and second the parallelization of the operations in the common path graph (Section 6.2). The result is a relaxed dependence graph. 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 [12]), and, in particular, *Grouping of Data Manipulation Operations* (see [5], [6] 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 [11] and [10] for further discussion.

Related work. Aspects of hardware and software architecture that increase an implementation's efficiency are discussed in [5], [6], [12], [7] and [14]. The parallelization of protocol implementations as proposed for example in [4] depends entirely on the intuition of the designer and thus its efficiency may be non-optimal. Others ([9]) analyze the data- and message flow dependences between communicating processes, whereas we restrict ourselves to the analysis of local dependences inside processes.

Precursors. Precursors of our work appeared in [11] where we describe the application of our method to a IP/TCP/FTP protocol stack. More technical detail, in particular the algorithms discussed in this paper, can be found in [10].

The role of SDL. The formal specification technique we consider is the CCITT standardized *Specification and Description Language* SDL [3]. 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. The choice of a formal description technique as starting point connects our method to existing techniques and methods in the domain of protocol engineering. Parts of our method are specific to features of SDL. However, we claim that for many other procedural specification methods an easy adaptation is possible.

2 A Discussion of SDL Specifications

2.1 SDL Specifications of Protocol Stacks

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. For reasons of conciseness of the presentation we abstract away from the SDL mechanism of mapping between output and input signals by signal routes and identify sender and receiver of messages simply by identity of the message type.

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 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.

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. 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. Similar arguments can be found in [7].

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.

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. We first analyze dependences inside SDL transitions and then analyse entire protocol stacks.

Syntactic structure of an SDL transition. 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 the sake of conciseness we have limited our considerations to this language subset but we conjecture that it is adequately expressive (see [10]).

3.1 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 runtime 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. Typically a variable used by a statement is found 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 redefinition 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. 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].

[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.

3.2 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. an `INPUT` statement has exactly one successor, and it is the root of the tree, 2. every `TASK` node has at most one successor, and an `OUTPUT` statement is a leaf of the tree.

3.3 Example SDL processes and TDGs

The example in Figure 1 shows a partial view of the specification of a process `N` of which we show only two transitions.

Figure 1 presents a graphical representation of corresponding TDGs starting in nodes `S1` and `S6`. *Solid* line arrows represent elements of *cfd*, and *dashed* line arrows represent elements of *dfd*.

A further example of an SDL process named `N+1` is 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 processes `N` and `N+1` together form the running example of our paper and we refer to it as *Two Layer Stack* TLS. `N` and `N+1` are only partially specified.

[2] This avoids additional *output* dependences, see [13].

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

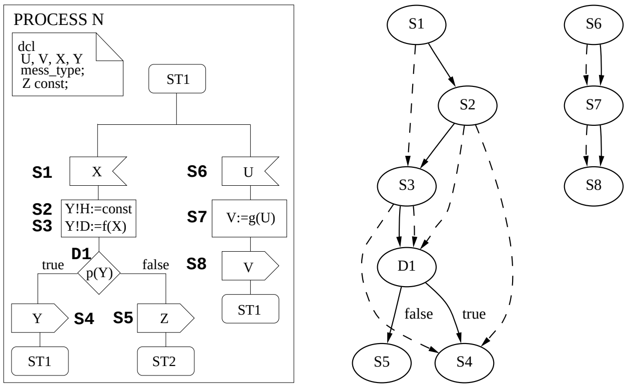

Figure 1: SDL specification (left) and TDGs (right) for process N.

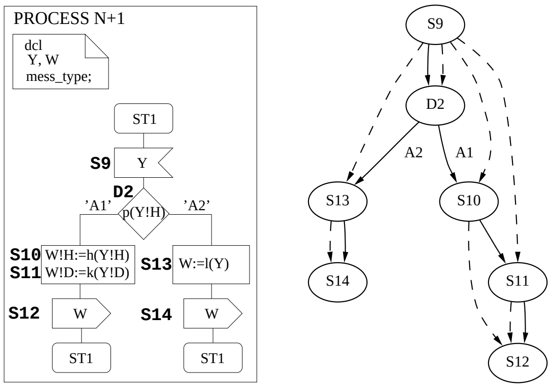

Figure 2: SDL specification (left) and TDGs (right) for process N+1.

4 Dependence Graphs for Protocol Stacks

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. First, we label all TDGs of all processes by so-called *input/output labels*. 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. 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 or from the environment, and end with an `OUTPUT` statement which delivers the processed packet to the environment or 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. To build the input/output labeled TDGs we simply label each input and output statement with the name of the signal they are inputing our outputing. A formal definition of IOTDGs can be found in [10]. 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 starts with an TDG such that its root node corresponds to an input of a packet from the environment. It then appends to each of its leaf nodes TDGs with matching input labels on their root node. The algorithm terminates when all possible compositions have been carried out.

As a result we obtain a set of MLDGs $\mathcal{M}$. Each $M \in \mathcal{M}$ is a multi-edged labeled tree $(S, STT, X, SIG, sttype, cfd, dfd)$. However, 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 with more than one successor is a *branching* node, and that leaf nodes are *exit* nodes. 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. Details of the MLDG construction algorithm can be found in [10].

Example MLDGs Figure 4 shows the set $\mathcal{M}$ of MLDGs 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. 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 $(S2, D2)$.

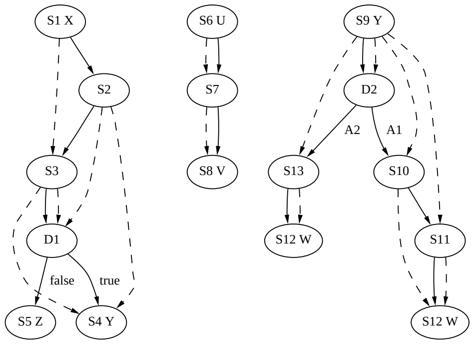

Figure 3: IOTDGs for Example TLS.

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

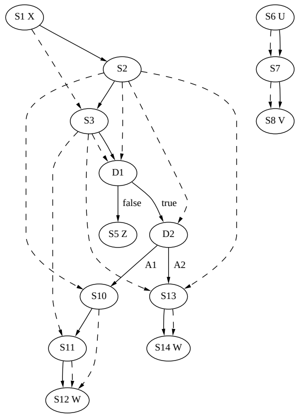

Figure 4: MLDGs for Example TLS.

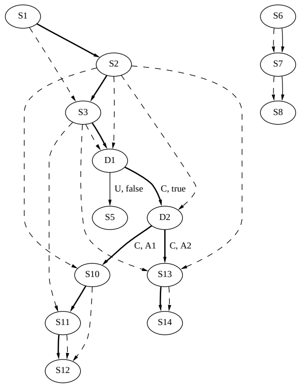

Figure 5: Labeled MLDG for Example TLS.

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. 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.

5 Determination of the Common Path Graph

In the later steps of our optimization method we optimize the processing of a packet only for the 'common case'. We consider our common path determination a generalization of the *Common Path* optimization as advocated in [5]. A major part of the functionality of

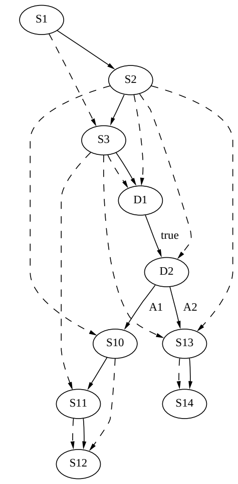

Figure 6: Common Path Graph for Example TLS.

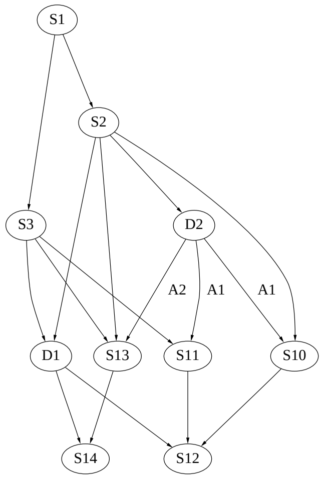

Figure 7: The Relaxed Dependence Graph for Example TLS

a protocol typically aims at detection and treatment of many kinds of exceptions and errors which are usually uncommon in typical high speed communication environments. Due to the low probability of uncommon cases we do not risk a degradation of the overall protocol performance even if the handling of these uncommon cases is inefficient. 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, in particular when the branching does not aim at handling exception cases.

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. It defines a *common path graph* which is a subgraph of the *cfd* graph. 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.

Labeling of MLDGs. 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. 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.

Common Path Graph (CPG). The algorithm for the construction of the CPG can be found in [10]. It simply prunes all those subtrees of the labeled MLDG which start with an edge labeled 'U' in a decision node. 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 this Section we will construct a relaxed dependence graph (RDG) based on which 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 an RDG: *anticipation of the common case*, and *relaxation of dependences.*

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 and thus limit potential parallelism. To enhance potential parallelism we anticipate the outcome of decisions that have only one outcome in the CPG. In [10] we discuss the handling of the uncommon cases in an implementation and argue that there is always a way to handle them consistently. Basically it is done by performing a roll-back to a consistent state when an uncommon case is detected. Anticipation of the common case is applied to the CPG by 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 [10]. 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 ensures 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. An algorithm performing this transformation is given in [10]. We call the resulting directed graph the relaxed dependence graph RDG of a CPG. It should be noted that the RDG 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 of the protocol will be based on the RDG. 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. A further gain in efficiency can be achieved by combining the execution of so-called *Data Manipulation Operations* (DMOs).

We call data manipulation operations (DMOs) operations that manipulate entire data parts of protocol data units. Combining two such operations into one which performs two manipulations at the same time saves an extra storing and fetching of all the data and thus executes much faster ([5], [6] and [1]). Particularly, it is more efficient to wait for all decisions to have been taken before executing DMOs since only then it is known which DMOs will have to be executed ([12]). 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.

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, iff other DMOs exist which can only be executed after some more decisions have been taken. Thus the DMO which depended of B will only be executed after one more decision has been taken. The algorithm is then recursively applied to all decisions that depend on B. An algorithm doing this is described in [10].

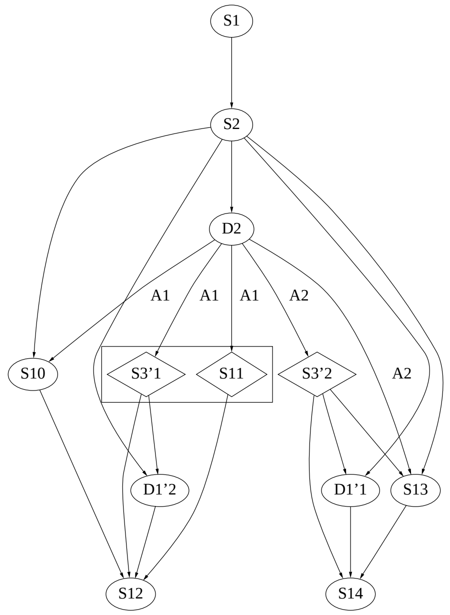

Figure 8: Grouped DMOs

The application of the algorithm to our example is shown in Figure 4.2 and Figure 8. 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.

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 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. We called the result a relaxed dependence graph. The RDG allows to make more efficient schedules for either parallel or sequential execution. 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. 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] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb 1993.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] S. Leue and Ph. Oechslin. A formal approach to optimized parallel protocol implementation. Technical Report IAM-94-03, University of Berne, Institute for Informatics, Berne, Switzerland, 1994.

[11] 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.

[12] 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.

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

[14] 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.