A Formal Approach to Optimized Parallel Protocol Implementation

Stefan Leue[1] and Ph. Oechslin[2]

[1]Institut für Informatik, Universität Bern, Länggassstr. 51, CH-3012 Bern, Switzerland
[2]Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, Laboratoire Téléinformatique, CH-1015 Lausanne, Switzerland

Keywords: CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors); C.2.2 [Computer-Communication Networks]: Network Protocols; D.2.1 [Software Engineering]: Requirements/Specification; D.2.2 [Software Engineering]: Tools and Techniques; D.3.4 [Programming Languages]: Processors.
Additional Keywords: Specification and Description Language (SDL); Data- and Control Flow Analysis; High Speed Protocols; Parallel Protocol Implementation; Protocol Optimization.

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.

Correspondence and offprint requests to: S. Leue, e-mail: leue@iam.unibe.ch, or Ph. Oechslin, e-mail: oechslin@di.epfl.ch

Contents

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 Figure 1 we present a (partial) view of the SDL specification of a two layer protocol stack. 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 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 2 and 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 [FOW87] and [BENP93]. Control flow dependences relate directly successive statements (e. g. `S2` and `S3` in Figure 2) 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 2).
- In 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 5.
- 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 7. 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 7).
  - *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 [OP91]), and, in particular, *Grouping of Data Manipulation Operations* (see [CJRS89], [CT90] and [AP93]).

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

Related work. Efficiency of implementation has become an imperative requirement in the context of high speed protocols. Aspects of hardware and software architecture that increase an implementation's efficiency are discussed in [CJRS89], [CT90], [OP91], [CWWS92] and [TW93]. Hardware implementations for high speed protocols have been proposed in [KS89]. Special attention has been paid to the parallelization of protocol implementations, so for example in [BZ92] and [WF93]. 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 [FH94]. In [MT93] the determination of data-flow dependence graphs for parallel implementations of stream processing programs on transputers are described. Others ([PP91], [LS92]) analyze the data- and message flow dependences between communicating processes, whereas we restrict ourselves to the analysis of dependences inside processes.

Precursors. A precursors of our work has appeared in [LO93] where we describe the application of our method to a IP/TCP/FTP protocol stack.

The role of SDL. The formal specification technique we consider is the CCITT standardized *Specification and Description Language* SDL [CCI92]. 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 [BHS91]. 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 [Liu89]). We may for example assume that as result of a previous verification step the specifications on which we base our optimization are dead- and live-lock 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. The *Two Layer Protocol Stack* (TLS) example of two protocol processes `N` and `N+1` which we assume to belong to adjacent layers of some protocol stack are presented in Figure 1. This will be the running example in the remainder. Both processes are only partially specified. Process `N` accepts either a message of type `X` from a non-specified lower layer service, which is then processed and sent out as a message of either type `Y` or type `Z`, or it accepts a message of type `U` which after processing is being sent out as a message of type `V`. In the remainder we shall sometimes abuse terminology in that we say *a message* `X` instead of *a message of type* `X`. Process `N+1` accepts a message `Y` which is processed and sent out as a message `W`. In SDL the mapping of the output signals of the sending process to

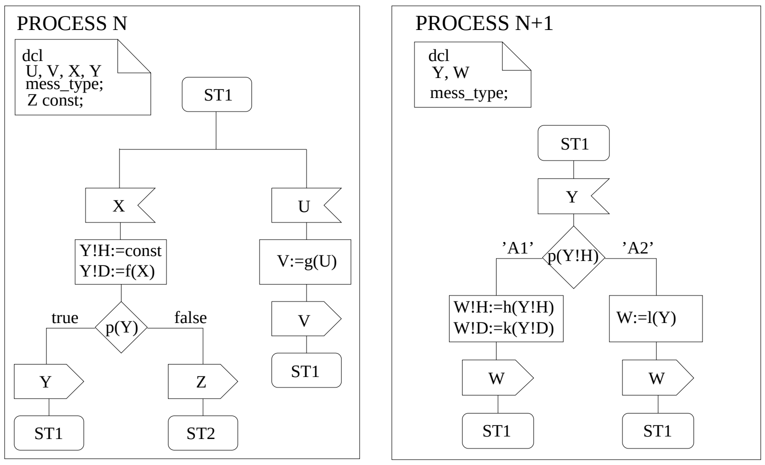


Fig. 1. The Two Layer Protocol Stack (TLS) Example.

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, the message `Y` sent out by process `N` is consumed by process `N+1`.

Structure of an SDL Specification. An SDL specification has the following structure. A protocol stack consists of a set of concurrent processes, one for each layer protocol entity. These communicate via asynchronous signal queues. The processing inside each process is sequential. At run-time the set of processes that form an SDL specification run concurrently, so an SDL specification can be seen as a collection of sequential processes that run in parallel. Each process can be structured into a set of transitions, each transition leading from a symbolic state to another or the same symbolic state, triggered by an input signal (see for example Figure 1). The labels that identify the different states allow for loops and branchings in the control flow of the processes. A transition may lead to many successor states, the choices are either made in decisions or by checking the different `INPUT` events by which a transition can be triggered, thus leading to branching and looping in the control flow of a process.

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 not efficient.

- *No explicit parallelism:* Although SDL processes run concurrently the processing inside an SDL process is strictly sequential. This means that the structuring of the specification into processes, which in many cases is influenced by 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 [BHS91]). 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 [CWWS92].

- *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 so that parallel and combined execution of statements belonging to different processes is enabled.

3. Dependence Analysis for SDL Processes

In this section we explain how a 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. The statement `STATE` denotes the current symbolic state and precedes a transition. The statement `NEXTSTATE` denotes the next symbolic state into which the system transits after executing the steps in the transition body. The `STATE` and `NEXTSTATE` statements do not belong to the transition body. We assume that the transition body has the following syntactical structure. A transition starts with an `INPUT` statement. An `INPUT` statement may be followed by a `TASK`, `DECISION` or `OUTPUT` statement, or it may be directly followed by a `NEXTSTATE` statement. The `TASK` statement may be followed by a `DECISION`, `TASK`, `OUTPUT` or `NEXTSTATE` statement. A `DECISION` statements may be followed by a `TASK`, `DECISION`, `OUTPUT` or `NEXTSTATE` statement. An `OUTPUT` statement is the final statement of a transition and always followed by a `NEXTSTATE` statement.

Justification. 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 [BHS91]. 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 can be accomplished 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 specification, namely control flow and data flow dependences.

Dependences. We now describe the different types of dependences between statements informally.

- 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. We will in later optimization steps, in particular when parallelizing the dependence graph, have to ensure that statements will only be executed when the decision on which they depend has been taken.
- 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. It should be noted that for reasons of simplicity we only allow one variable to be defined in one statement, hence all assignment statements are *single assignment statements*.
- 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 $T = (S, STT, X, \mathit{sttype}, \mathit{cfd}, \mathit{dfd},)$ where $\mathit{cfd} \subseteq S \times S$, $\mathit{dfd} \subseteq \mathit{cfd}^+$, $STT = \{\mathit{input}, \mathit{decision}, \mathit{task}, \mathit{output}\}$, $\mathit{sttype} \subseteq S \times STT$ is a functional relation (relating a statement to a statement type), $\mathit{use} \subseteq S \times \mathcal{P}(X)$ is a functional relation (relating a statement to the set of variable names which are being *used* in it), and $\mathit{define} \subseteq S \times X$ is a partial functional relation (relating a statement to the variable name which is being *defined* in it), satisfying the following conditions:

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

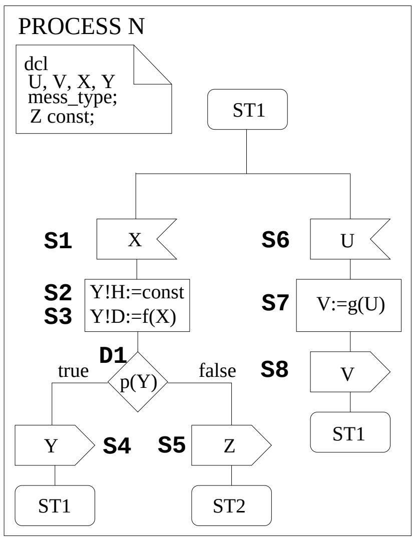
[2] This avoids additional *output* dependences, see [PW86].

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

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

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.



```
     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;
```

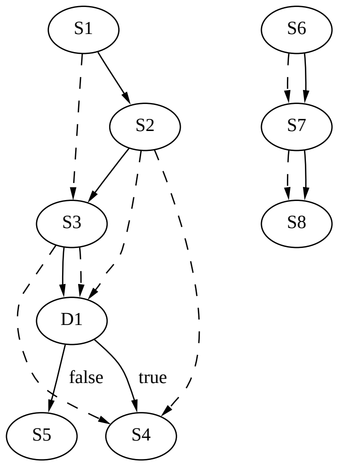
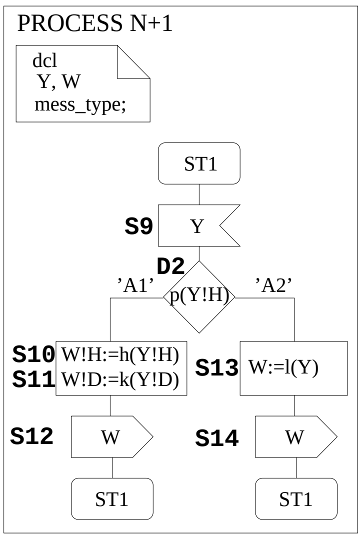


Fig. 2. SDL-PR (left) and SDL-GR (middle) specifications and TDGs (right) for process N

The example in Figure 2 shows a partial view of the specification of a process `N` of which we show only two transitions. The transition on the left hand side leads from state `ST1` via statements `S1`, `S2`, `S3`, `D1` and either via `S4` to a successor state `ST1` or via `S5` to successor state `ST2`, depending on the evaluation of the decision predicate `p(Y)`. This transition is triggered by the input of an `X` signal. In statements `S2` and `S3` the variable `Y` is defined. We assume that a variable of type `mess_type` is defined as a record, and that for example the expression `Y!H` refers to the first component of the record `Y` and `Y!D` to its second component[4]. The evaluation of the decision predicate `p(Y)` determines whether a message `Y` or a message `Z` will be issued, and hence whether the successor state will be `ST1` or `ST2`. The dependences are as follows. The *control flow* dependence follows the linear sequence of the statements `S1`, `S2`, `S3` and `D1` and then branches to either `S4` or `S5`. The `DECISON` statement `D1` has possible successor statements `S4` and `S5`, the respective control flow dependence edges are labeled for illustrative purposes by *true* and *false*. The data flow dependences are so

[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.

that S3 depends on S1 because of variable X, whereas D1 and S4 both depend on S2 and S3 because of the use of variable Y. Figure 2 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*. It should be noted that the labels in the nodes are only annotations which allow us to refer to single nodes more easily. When in state ST1 process N may execute two different transitions, depending on whether the signal available at the head of the input queue is of type X or of type U. Above we described the transition for the first case, for the second case the second transition leads from ST1 via statements S6, S7 and S8 to state ST1. The syntactical analysis as described above leads to the transition dependence graph T_2. The graph representation can be found on the right hand side of Figure 2.



```
          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;
```

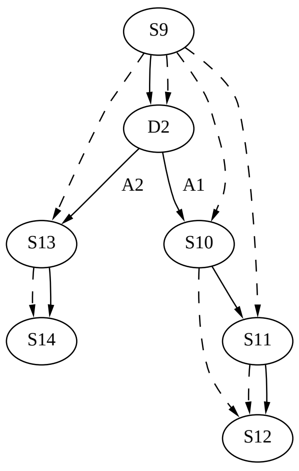


Fig. 3. SDL-PR (left) and SDL-GR (middle) specifications and TDGs (right) for process N+1

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 first presented in Figure 1. The syntactical analysis leads to the transition dependence graph T_3 shown on the right hand side of Figure 3. 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.

Preview. The purpose of the transformation of SDL specifications into dependence graph is to obtain an algebraic representation for later optimizations, in particular parallelizations. The parallel execution of statements is allowed if they are neither neither directly nor indirectly data flow dependent on each other. Looking at the TDG T_1 on the right hand side of Figure 2 we see that for example statements S3 is control flow, but not data dependent on statement S2. This reveals a potential for parallelization.

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 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 $T = (S, STT, X, sttype, cfd, dfd)$ 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 $insig \subseteq ((S \cap root(T)) \times SIG)$ and $outsig \subseteq ((S \cap leaves(T)) \times SIG)$ denote functional relations. We define an Input-Output labeled Transition Dependence Graph (IOTDG) as a tuple $I = (S, STT, X, SIG, sttype, cfd, dfd, insig, outsig)$ for which the following conditions hold: $sttype(root(I)) = input$, and $(\forall x \in leaves(I))(sttype(x) = output)$.

Example IOTDG In Figure 4 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 $root(Z)$ 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 $root(Z)$ 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 $root(Z)$. 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
   $(\forall T_i')(\forall T_j)(insig(root(T_i')) \cap \bigcup_{j \neq i} outsig(leaves(T_j)) = \emptyset)$;
II. $\mathcal{M} := \emptyset$;
III. FOR ALL $T_i' \in \mathcal{T}'$
   IV. $M := T_i'$;
   V. $\mathcal{Z} := \{T \in \mathcal{T} \mid outsig(leaves(M)) \cap (insig(root(T))) \neq \emptyset\}$;
   VI. WHILE $\mathcal{Z} \neq \emptyset$
      VII. FOR ALL $Z \in \mathcal{Z}$
         VIII. SELECT $x \in leaves(M)$ SO THAT
            $(outsig(x) \in insig(leaves(root(Z))))$;
         IX. $S_M' := S_M \cup S_Z - \{x\} - root(Z)$;
         X. $X_M' := X_M \cup X_Z$;
         XII. $sttype_M' := S_M' \lhd (sttype_M \cup sttype_Z)$;
         XIII. $cfd_M' := cfd_M \cup cfd_Z - (cfd_M \rhd \{x\}) - (root(Z) \lhd cfd_Z)$
            $\cup \{domain(cfd_M \rhd \{x\}) \times range(root(Z) \lhd cfd_Z)\}$;
         XIV. $dfd_M' := dfd_M \cup dfd_Z - (dfd_M \rhd \{x\}) - (root(Z) \lhd dfd_Z)$
            $\cup \{(v,w) \in \{domain(dfd_M \rhd \{x\}) \times range(root(Z) \lhd dfd_Z)\} \mid define(v) \subseteq use(w)\}$;
         XV. $M := (S_M', STT_M, X_M', sttype_M', cfd_M', dfd_M')$
      XVI. $\mathcal{Z} := \{T \in \mathcal{T} \mid outsig(leaves(M)) \cap (insig(root(T))) \neq \emptyset\}$;
   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 $(S, STT, X, SIG, sttype, cfd, dfd)$. 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.

Entry, Exit and Branching Nodes. 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 5 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 $(S2, D2)$.

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

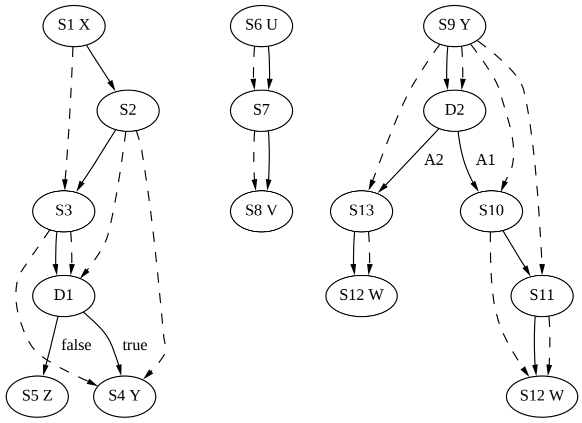


Fig. 4. IOTDGs for Example TLS.

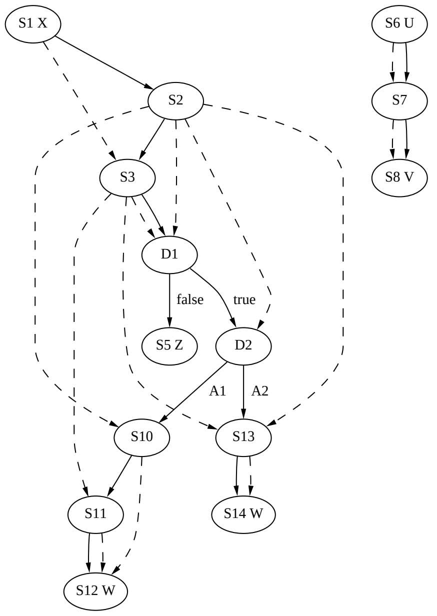


Fig. 5. MLDGs for Example TLS.

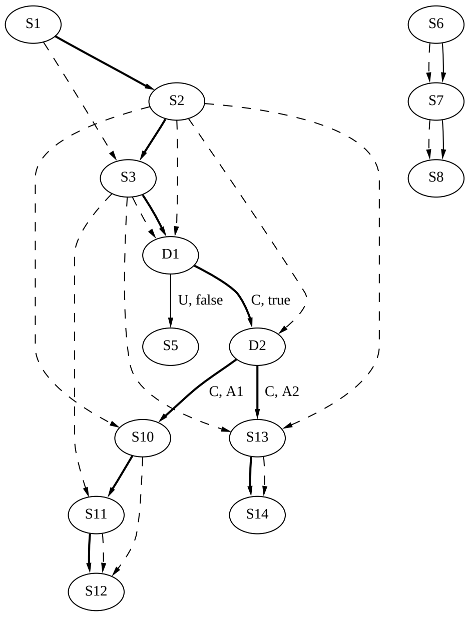


Fig. 6. Labeled MLDG for Example TLS.

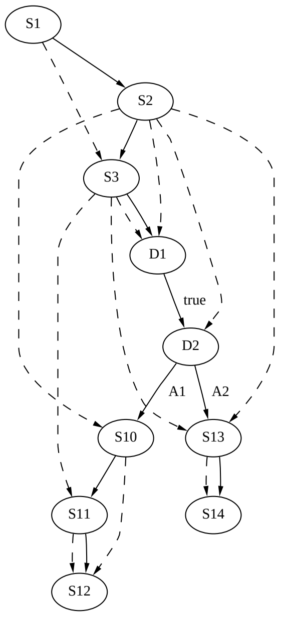


Fig. 7. Common Path Graph for Example TLS.

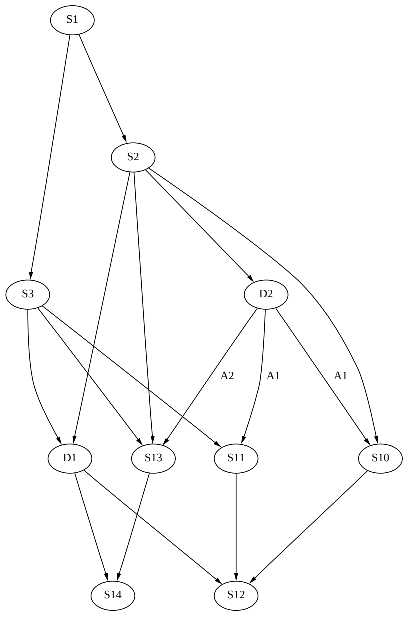


Fig. 8. The Relaxed Dependence Graph for Example TLS

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 [HE93] 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). Restricting the optimization to the common case has the advantage of reducing the complexity of the code that needs to be optimized and therefore leads to more compact optimized code modules. Furthermore, in Chapter 6 we introduce optimization steps that anticipate certain common decision results according to a common case assumption. These optimization steps, which rely on relaxing the dependences of statements before and after certain decisions, would be impossible without the common case assumption. We consider our common path determination a generalization of the *Common Path* optimization as advocated in [CJRS89].

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 6 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 6, 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 [ADM92][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)

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

Algorithm 2.

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

Example CPG. In figure Figure 7 we present the CPG derived from the common/uncommon labeled MLDG in Figure 6. 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.

Preview. In the rest of this document we will focus on the optimization and implementation of the common path of protocols. However, the result of the dependence analysis in Chapter 3 has been a set $\mathcal{M}$ of MLDGs, whereas this Section only addresses the determination of a CPG based on a single MLDG. We expect that the user decides which elements of $\mathcal{M}$ he wishes to be optimized by the later optimization steps, based on a similar common/uncommon decision as we discussed earlier.

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 and data flow dependence analysis. 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 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.

[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.

6.1. Anticipation of the Common Case

This is the first transformation we apply to the CPG. 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 thath 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. What does this exactly imply? A successor of an anticipated decision can henceforth be executed before the outcome of the decision is known. If the outcome corresponds to the anticipation we have a gain in parallelism. If, however, the anticipated decision has an uncommon outcome then statements which have already been executed in anticipation may have to be canceled. In Section 8 we discuss the handling of the uncommon case 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 using algorithm 3. It selects all decision nodes that have only one successor from the set S_C of the nodes of the CPG (I.) and changes the type of these nodes to *task* (III.).

Algorithm 3.
```
I. SELECT $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ SO THAT
   $(\forall D_i)((sttype(D_i) = decision) \wedge (\mid \{D_i\} \lhd cfd \mid = 1))$
II. FOR ALL $D_i \in \mathcal{D}$ DO
    III. $sttype(D_i) := task$
```

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

Example In our example, anticipating the common case results in changing the statement type of D1 from*decision* to *task*. In Figure 8 we see that after the next transformation it will for example be possible to execute D1 after S11.

6.2. Relaxation of Dependences

In this transformation we remove dependences from the CPG graph to allow its parallel execution[6]. 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

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

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 4 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 4.
```
I. SELECT $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ SO THAT
   $(\forall D_i)(sttype(D_i) = decision \vee D_i = root(C))$
II. $rxd := dfd_C$
III. FOR ALL $n \in S_C - root(C)$
    IV. SELECT $D_n$ SO THAT
        $\{s \in \mathcal{D} \mid (s, n) \in cfd_C^+ \wedge (D_n, s) \in cfd_C^+\} = \emptyset$
    V. IF $(D_n, n) \notin rxd^+$ THEN $rxd := rxd \cup \{(D_n, n)\}$
VI. FOR ALL $m \in S - leaves(C)$
    VII. FOR ALL $x \in leaves(C)$
        VIII. IF $(m, x) \in cfd_C^+ \wedge (m, x) \notin rxd^+$ THEN $rxd := rxd \cup \{(m, x)\}$
```

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

Example. Figure 8 shows the RDG for the CPG in Figure 7. 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. Examples are checksum calculation and encryption of data. Combining two such operations into one that does two manipulations at the same time saves an extra storing and fetching of all the data and thus executes much faster. This has already been demonstrated in [CJRS89]. It is also central to the work reported in [CT90] and [AP93]. Particularly, it has been shown in [OP91] 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. The grouping of DMOs means that we tightly couple the processing of operations which in the RDG are not dependent, so that their joint processing requires less execution time.

In order to enable the joint scheduling the RDG has to be modified. It has to be taken into account that when grouping the execution of two DMOs so that one operation depends on a decision higher up in the RDG than the other operation, the higher operation must be executed along every possible path through the RDG. It is thus necessary to distribute DMOs over the RDG.

Example. In our example we assume S3 and S11 to be DMOs. We identified them manually as we have not defined formally how a DMO is characterized in SDL. We include all identified DMOs in a set called $\mathcal{DMO}$. The condition that must hold to combine two DMOs is that there may not exist a node which depends from the first DMO and from which the second DMO depends. Such a node would clearly have to be executed after the first DMO but before the second, thus preventing their combined execution. For two DMOs to be executed at the same time, all decisions of which they depend must have been taken. In our example, even if S3 does not depend of D2 we have to execute S3 after D2 because only then we know if S11 will have to be executed at all. To make sure that S3 is executed after D2 we make it depend directly of D2 rather then on the node it is originally depending of. We have, however, to take in account that S3 will have to be executed independently of D2. Thus we need to 'distribute' S3 over all possible evaluations of D2. Distributing a DMO over the possible evaluations of a decision predicate means that we make one copy of the node representing the DMO for each possible outcome of the decision. In our example there will be two copies, one corresponding to 'A1' ($S3'_1$) and one to 'A2' ($S3'_2$). If D2 evaluates to 'A2' we can execute a combined DMO $S3'_2$/S11. If D2 evaluates to 'A1' then we execute $S3'_1$ alone.

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 5 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 5.

```
RecursiveCombine($B$)
  I. FOR ALL $\{D \in \mathcal{DMO} \mid (B, D) \in rxd\}$
    II. IF $\exists D_2 \in \mathcal{DMO} \mid (D, D_2) \in cfd^+$ AND
        $\{s \in S_C \mid (D, s) \in rxd^+ \wedge (s, D_2) \in rxd^+\} = \emptyset$
      III. $B' = s \in branchnodes(C) \mid (B, s) in rxd \wedge (s, D_2) \in rxd^+$
      IV. $S_C := S_C - \{D\}$
      V. $\mathcal{DMO} := \mathcal{DMO} - \{D\}$
      VI. $rxd := rxd - \{D \lhd rxd \cup rxd \rhd D\}$
      VII. FOR ALL $N_i \in B' \lhd cfd$
        VIII. $S_C := S_C \cup \{D_i'\}$
        IX. $\mathcal{DMO} := \mathcal{DMO} \cup \{D_i'\}$
        X. $rxd := rxd \cup (B', D_i')$
        XI. $rxd := rxd \cup \{(D_i', x), x \in leaves(C) \mid (N_i, x) \in cfd^+\}$
  XII. FOR all nodes $newB \in B \lhd rxd$ and $sttype(newB) = decision$
    XIII. call recursiveCombine($newB$)
```

The algorithm creates groups of DMOs that are selected by the same evaluation of a decision. Such groups of DMOs can be combined to one complex DMO which can be implemented in a much more efficient way. The actual rewriting of the DMOs into one complex DMO is done during implementation. One method for doing this is given in [AP93].

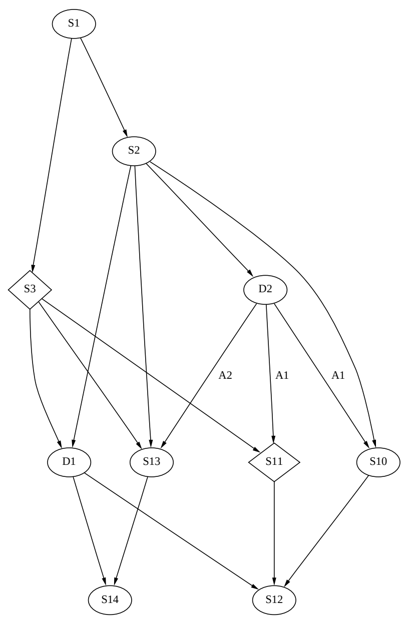


Fig. 9. CPG with marked DMOs (diamonds)

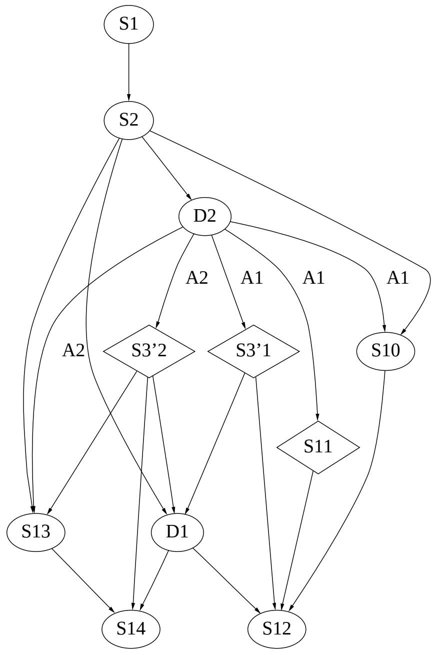


Fig. 10. Grouped DMOs

Example. The application of the algorithm to our example is shown in Figure 9 and Figure 10. 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. Implementing the Optimized Graph

In the previous chapters, we have seen how a multi-layer dependence graph (MLDG) can be derived from a SDL specification, how a common path graph (CPG) can be extracted from the MLDG and how this CPG can be transformed to a relaxed dependence graph (RDG). This chapter describes the final part of our method, namely the implementation of a specified protocol, based on an RDG. Implementing the RDG means that we map the statements corresponding to each node to a set of software- or hardware instructions. When performing this mapping we have to consider three aspects. First, we have to preserve the ordering constraints imposed by the RDG. Second, assuming the availability of parallel processing resources the operations have to be scheduled according to the ordering constraints, the resource requirements and the expected time consumption of every operation. Finally we have to take care of the fact that our RDG only addresses the common case, i. e. we need to solve the problem of the alternate processing when a packet belongs to the uncommon case.

8.1. Preserving Ordering Constraints

The RDG imposes a set of ordering constraints on the operations to be executed. In general, this is a partial order. If we look back at the RDG in Figure 10 it is easy to see that for the subset $\{D2, S10, S11\}$ of operations the following partial order, expressed informally in terms of a process algebra like behavior expression, is `(D2; (S11 || S11))`. Any interleaving trace derived from this expression is the trace of a valid implementation, e. g. the traces $(D2, S10, S11)$ and $(D2, S11, S10)$. However, for the exact derivation of an optimal implementation these possible interleavings do not provide sufficient information, in particular for the following two reasons.

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

The resource dependency and the time duration arguments above give rise to a scheduling problem that has to be solved in order to derive an optimal implementation of the RDG on a given hardware architecture.

8.2. Scheduling

Scheduling problems are a domain of Operational Research. For a general overview and classification of scheduling problems see [LLKS93]. The problem we have to solve is to decide at what time and with which resources to execute the operations of the RDG without violating neither the ordering constraints nor the resource allocation constraints. The ordering constraints are given by the RDG, the resource constraints are given by the machine on which the protocol is implemented. An optimal schedule can basically be found using following two steps:

- First compile a list of all independent resources available in the system on which the protocol is implemented. For each operation of the common path graph define which resources may be needed and estimate the execution time.
- Second, given this information all possible schedules can be enumerated and the ones minimizing the time at which the exit nodes are reached can be selected. The enumeration is replaced by a heuristic in more complex problems.

The exact nature of the problem varies with the kind of hardware on which the protocol is implemented. Here are two examples:

In [MT93] data stream applications are scheduled on multi-transputer systems. As all processing elements are equivalent the main goal is to distribute the operations in a way such that the workload is equally distributed and the communication between transputers is minimized. The optimal schedule is found using a branch and bound with underestimate (BBU) heuristic. In [LO93] we present an example of an implementation of a TCP/IP stack. The hardware consists of two processors, one general purpose and one specialized, which share some common resources like busses and memories. In this case the distribution of operations over resources is given by the specificity of the resources. The main goal, here, is to avoid that one of the two processors is idle while it waits for the other to release a resource it needs. Again branch and bound heuristics can be used to find an optimal schedule.

8.3. Dealing with the Uncommon Case.

When deriving the the RDG from the initial specification we assumed a common case, and as a consequence we anticipated the results of some of the decisions along the common path. This means that we presumed a certain evaluation of some of the decisions and removed dependencies of statements depending on these decisions. In other words, some operations have been decoupled from the decision predicates by which they were 'guarded' in the original specification. For example, a division by zero may be executed concurrently with the test for non-zeroness of the respective operand. In the original specification of our example TLS (see Figure 2 and 3) the execution of `D2` (through `S4`) depends on the evaluation of decision `D1` to `true`. However, in the RDG in Figure 10 `S4` does *not* depend on the evaluation of `D1`. This implies that `D2` may even be executed *before* `D1` is evaluated. Apparently, consistency ensuring mechanisms have to be applied. This leads to the following three requirements.

- First, as we argued before we need to have a classical implementation of the whole protocol available. This classical implementation, which covers all decisions, exception handling mechanisms etc., takes over control when the optimized implementation detects that a packet violates the common case assumption, namely if a test does not evaluate to the value which was anticipated during optimization.
- Second, because we saw that operations may be executed prior to the evaluation of a decision predicate by which they were originally guarded, all operations must be robust. This means that no matter when an operation is executed it is ensured that the system will not enter a failure state.
- Third, when the processing control is handed over to the classical implementation the state of the system when the packet has entered the protocol stack through an entry node has to be reestablished. To ensure that this initial state can always be reestablished we suggest using the following mechanism.
  - We distinguish operations in reversible and irreversible operations. We claim that most operations are reversible, in particular operations reading data or copying data from one location, modifying the data and writing it to a second location. Reversible operations can then be undone when control goes to the classical implementation by carrying out the appropriate inverse operation.
  - All those operations which are irreversible, and we claim that that is only a minor part of all operations, need to be secured by a checkpointing mechanism. This means that the data which is affected by these operations will be checkpointed before the respective operation is executed. If not all decisions are

evaluated in the way it was anticipated, i. e. the packet is not processed according to the common case, the checkpoint information can be used to undo all irreversible operations.

It arises the justified question how advantageous our optimization is in light of these time consuming consistency ensuring mechanisms. We claim that the resetting to the initial state only occurs very infrequently, and that only very few operations are actually irreversible and thus need to be protected by checkpoints.

9. 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.

In general, implementing the RDG means that we map the statements corresponding to each node to a set of software instructions or hardware modules. When performing this mapping we have to consider three aspects. First, we have to preserve the ordering constraints imposed by the RDG. Second, assuming the availability of parallel processing resources the operations have to be scheduled according to the ordering constraints, the resource requirements and the expected time consumption of every operation. Finally we have to take care of the fact that our RDG only addresses the common case, i. e. we need to solve the problem of the alternate processing when a packet belongs to the uncommon case. A preliminary description of these implementation aspects is given in [LO93] (see also Section 8).

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

[ADM92] 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.

[AP93] M. Abbott and L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5), October 1993.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb 1993.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

[BZ92] 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.

[CCI92] CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, Geneva, 1992.

[CJRS89] 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.

[CT90] 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.

[CWWS92] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE Network Magazine*, pages 20–24, january 1992.

[FH94] 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.

[FOW87] 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.

[HE93] B. Hofmann and W. Effelsberg. Efficient implementation of Estelle specifications. Technical report Reihe Informatik, Nr. 3/93, University of Mannheim, Mannheim, Germany, 1993.

[KS89] 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.

[Liu89] M. T. Liu. Protocol engineering. In M. C. Yovitis, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, Inc., 1989.

[LLKS93] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *Handbook in Operation Research and Management Science, Vol IV.* North-Holland, 1993.

[LO93] S. Leue and P. 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.

[LS92] 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.

[MT93] 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.

[OP91] 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.

[PP91] W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM TOPLAS*, 21(3):399–442, 1991.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[TW93] 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.

[WF93] C. M. Woodside and R. G. Franks. Alternative software architectures for parallel protocol execution with synchronous IPC. *IEEE/ACM Transactions On Networking*, 1(2):178–186, April 1993.

Appendix

Notation and Definitions

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

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

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

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

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

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

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

Digraphs and Trees. Let V denote a set and let $E \subseteq V \times V$, then we call $T = (V, E)$ a *digraph*. We call T a *tree* if and only if the following additional conditions hold:

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

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

Multi-edged and Labeled Trees.

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

Operations on Trees.

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