A Domain-Specific Language for Protocol Stack Implementation in Embedded Systems
Yan Wang
A Domain-Specific Language for Protocol Stack Implementation in Embedded Systems
YAN WANG

A Domain-Specific Language for Protocol Stack Implementation in Embedded Systems
Abstract

Embedded network software has become an area of increasing importance for both research and industry as more and more applications are built on networked embedded systems. Modern devices and applications require newly designed or revised protocols which have to be implemented. Also, well-known infrastructure protocol stacks have to be reimplemented on new hardware platforms and software architectures. However, implementing protocol stacks for embedded systems remains a time-consuming and error-prone task due to the complexity and performance-critical nature of network software. It is even more so when targeting resource-constrained embedded systems, as implementations also have to minimize energy consumption and meet memory constraints.

This thesis addresses how to facilitate protocol stack implementations for embedded systems and how to determine and control their resource consumption by means of a language-based approach. It aims at a domain-specific language (DSL) that supports abstractions suitable for the implementation of protocol stacks. Language technologies in the form of a type system, a runtime system and compilation can then be used to generate efficient implementations.

In the work presented in this thesis, we give background on DSL implementation techniques. We also investigate common practices in network protocol development to determine the potential of DSLs for embedded network software. Finally, we propose a domain-specific embedded language (DSEL), Protege (Protocol Implementation Generator), for declaratively describing overlaid protocol stacks. In Protege, a high-level packet specification is dually compiled into an internal data representation for protocol logic implementation, and packet processing methods which are then integrated into the dataflow framework of a protocol overlay specification. The Protege language offers constructs for finite state machines to specify protocol logic in a concise manner, close to the protocol specification style. Protege specifications are compiled to highly portable C code for various architectures.

Four attached papers report our main results in more detail: an embedded implementation of the data description calculus in Haskell, a compilation framework for generating packet processing code with overlays, an overview of the domain-specific language Protege, including embedding techniques and runtime system features, and a case study implementing an industrial application protocol.
Acknowledgements

First of all, I would like to thank my supervisor, Dr. Verónica Gaspes for inspiring my research area and supporting me in the work I am going to present.

Likewise, thanks go to all the people who made it possible for me to carry out my research: my main supervisor, Prof. Thorsteinn Rögnvaldsson, my co-supervisor, Prof. Dimitar Driankov, as well as the company CEO Per-Arne Wiberg and his team at Freecmove, for supporting our project. Thanks to Jörgen Palmhager from HMS Industrial Networks for suggesting an interesting real-world scenario and providing hardware for tests. I would also like to thank Prof. Walid Taha for his important feedback to parts of my thesis. Special thanks to Prof. John Hughes for generously joining my follow-up committee and following up my Ph.D. study.

It is a pleasure to work at Centre for Research on Embedded Systems (CERES) at Halmstad University. Prof. Bertil Svensson and Prof. Magnus Jonsson deserve big thanks for leading the CERES and making it such a nice workplace. I would also like to thank Prof. Tony Lasson for his support to the master thesis projects under my supervision. Furthermore, I want to thank Eva Nestius, Jessika Rosenberg and Christer Svensson for helping me with administrative tasks.

I would like to thank all the current and former members of CERES. In particular, I thank Anita, Olga and Zain for their good friendship. Thanks to my former college and friend Xing for her mental support and useful suggestions. Thanks to Nicolina, Katrin, Hoai, Jerker, Marlin, Anders and Roland, for giving me help in many things. Thanks to Elisabeth for the CERES internal newsletter. Thanks to Wagner, Edsion, Kristina, Annette and Kristoffer for being good Ph.D. fellows.

Thanks to my friends who I met in Sweden: Yushe, Joacim, Wei, Tanya, Anna, Jens, Phu, Kevin, Sabeen, Alessandra, Wenjie, Maexuan and Xuerui. Our parties, trips and cooking sessions make my life in Sweden more enjoyable.

Last but not least, hugs to my family, especially to my grandpa and my parents. They have supported me and encouraged me during all my life. And my heartfelt thanks go to Jost. Thanks for his love, understanding, patience, support and encouragement.
Contents

1 Introduction ................................................... 1
  1.1 Contributions ........................................... 3
  1.2 Outline of the Thesis .................................... 4

2 Background ..................................................... 7
  2.1 Network Software ........................................ 7
  2.2 Embedded Network Software ............................. 8
  2.3 Existing Support for Protocol Stack Implementation ...... 10

3 A Language-Based Approach .................................... 13
  3.1 Domain-Specific Languages .............................. 13
    3.1.1 DSLs vs. Libraries .................................. 15
    3.1.2 Designing a DSL .................................... 16
    3.1.3 Implementing a DSL ................................ 17
    3.1.4 Domain-Specific Embedded Languages ............ 18
  3.2 Domain-Specific Languages for Network Software ...... 20
    3.2.1 Why a DSL for Network Software? ................ 20
    3.2.2 Existing DSLs for Network Software Development .. 21

4 Practices for Implementing Network Software .................. 27
  4.1 Protocol Specification and Analysis .................... 27
    4.1.1 Packet Specification ................................ 28
    4.1.2 Protocol Logic Specification ....................... 28
    4.1.3 Analysis ............................................. 29
  4.2 A Software Engineering Perspective ..................... 30
  4.3 Improving Code Structure for Embedded Network Software .. 32
    4.3.1 Packet Processing .................................. 32
    4.3.2 Protocol Logic ...................................... 36
    4.3.3 Layering ............................................ 37
    4.3.4 System Related Facilities .......................... 41
5 Summaries of Attached Papers
   5.1 Embedding a Data Description Language into Haskell . . . . . . . 47
   5.2 Generating Packet Processing Code from Data Descriptions . . . 49
   5.3 The Protege Language and its Implementation . . . . . . . . . . . . 50
   5.4 Modbus, a Case Study for Protege . . . . . . . . . . . . . . . . . . . . . . 52

6 Conclusions and Future Work 55

Bibliography 59
List of Figures

2.1 The four layers of the TCP/IP protocol suite ............... 8
3.1 The payoff of software development .......................... 14
4.1 A general Modbus packet ...................................... 28
4.2 Client state diagram ........................................... 29
4.3 Parsing code (fragments) for Modbus packets .......... 33
4.4 FSM code for trigger event send (send function) ........ 38
4.5 FSM code for timeout and error processing trigger events 39
4.6 Modbus protocol data structure ......................... 40
4.7 Modbus protocol stacks (graph) ............................. 41
4.8 Constructing a data structure for layering (example) .... 42
5.1 The syntax of DDC ............................................. 48
5.2 Protocol specification code ................................. 50
5.3 Protocol stack overlay example ............................ 51
5.4 Modular decomposition of Modbus for a Protege implementation 53
List of Publications

The publications on which this thesis is based are appended:


List of additional publications:


LIST OF PUBLICATIONS


Chapter 1

Introduction

Today’s world is full of computerized systems which communicate and interact with each other, including conventional computers, smart phones, vehicle electronics, specialized process controllers in industrial setups, etc. Network software is an essential part of any such system, providing the necessary services to establish reliable and efficient data transfer. The performance of network software often has large impact on the overall system performance. Therefore, implementing network software is a system programming task of major importance. New upcoming devices and networking standards create an ongoing demand to maintain and customize existing implementations and to create entirely new ones.

Modern network software realizes complex functionality in the various layers of a protocol stack where protocols build on each other. Individual protocol implementations can be very large and complex, complicating correctness checks and maintenance. For example, a TCP implementation can include thousands of lines of code [KKM99]. In addition, a range of optimization techniques designed to make protocol code more efficient tend to make implementations even more intricate. For example, implementing a protocol stack in a monolithic way can reduce the overhead caused by layers and save code space [Dum07]. However, customizing such a monolithic protocol stack is almost impossible. When functionalities are changed or new functionalities are introduced, instead of reusing existing implementations, code for the whole protocol stack may need to be rewritten.

When implementing protocol stacks for embedded systems, additional non-functional requirements have to be met. Such systems are usually constrained in code and memory size, and often also in energy consumption. For example, small wireless sensor nodes only have a few kilobytes of code space and RAM, and run on on-board batteries. Furthermore, embedded systems are often built on special hardware, and in many cases, better performance and lower resource consumption is obtained by reducing and customizing protocol functionality.
to particular applications. For example, Modbus [Mod06a] has 21 standard
public functionalities, but only a few are needed on a particular machine.

The state-of-the-art is to carry out protocol stack implementations for em-
bedded systems at low abstraction level. They are traditionally implemented
using C or even assembly language. Using these low-level languages, program-
ners cannot benefit from achievements of modern programming language de-
sign such as static type systems and automatic memory management. Low-
level programming leads to tedious and error-prone code for protocol stack
implementations and network software in general. Also, specifications are not
mapped to implementations in a straightforward manner, which makes it dif-

cult to debug and maintain the code. Furthermore, low-level optimizations
potentially introduce more errors and can become an obstacle to future cus-
tomization. In order to improve on time-to-market, scalability, maintainability
and product evolution, it is relevant to investigate how to increase program-
ing efficiency and improve program development support for protocol stack
implementations in embedded systems.

This opens opportunities for a language-based approach in the form of a
Domain-Specific Language (DSL) [Hud97, MHS03]. DSLs have proved their
ability to support code consistency, performance, systematic code reuse, and
portability, in the development of software for various areas, such as finan-
cial products [PES00], communication services [CHR+03], hardware design
[BCSS98], cryptographic algorithms [Lew07] and network protocols [Mcg04,
BME98]. Based on domain-specific knowledge, a DSL provides domain ab-
stractions and notations, and thereby allows domain-oriented compiler opti-
mizations, constraint enforcement, and language-level debugging in a friendly
way for domain experts.

In this thesis, we apply state-of-the-art DSL technology to protocol stack
implementation in embedded systems. We describe an approach to make im-
plementing protocol stacks easier and more modular, while meeting perfor-
mance demands and keeping energy consumption and memory usage low. We
have developed the domain-specific language Protege (Protocol Implemen-
tation Generator) as a compiled language embedded in Haskell. Protege inte-
grates a data description language for specifying protocol packets with con-
structs to define protocol state machines, and combinators to overlay protocols
to a protocol stack. Protege descriptions are compiled to C code that has sta-
tically determined memory requirements. Entire protocol stacks are compiled
at once, providing opportunities for global optimization. The generated code
is also easy to integrate into other parts of an existing system.
1.1 Contributions

Scientific contributions

Analysis of protocol implementation practices for embedded systems In order to motivate the constructs of Protege and its runtime system, we analyze and discuss the common practices of protocol implementation in embedded systems. A number of design choices and implementation details have been derived from this analysis and can prove useful in other contexts.

A DSL approach to protocol stack implementation Using a domain-specific language approach and compilation, we provide programming abstractions and a framework to implement protocols in embedded systems, simplifying protocol development and ensuring the code quality. Advantages of our approach over conventional implementations are: higher abstraction level, increased modularity, comparable performance, and better portability.

A DSL implementation by embedded compilation For the implementation of Protege, we have used several different techniques of deep and shallow language embedding and compilation to C code, including the compilation of helper functions. We thereby contribute a rich case study in language embedding and compilation.

Major concrete contributions

A study of domain-specific language techniques (Chapter 3) We show the state-of-the-art in DSLs, with a particular focus on domain-specific embedded languages, and on existing DSLs for network software development.

An analysis of embedded network software implementations (Chapter 4) We analyze common implementation practices for protocol stacks in embedded systems. We suggest programming disciplines to improve protocol stack implementations and to derive and substantiate design decisions for the domain-specific language envisioned.

A library for processing ad-hoc data formats in Haskell (Paper I) We have developed a library for processing ad-hoc data formats in Haskell based on the Data Description Calculus [FMW06]. It uses dependent types to describe various forms of ad-hoc data, which is a generalization of the part of the Protege language that describes packets.

A specialized formalism for packet format specification using dependent types (Paper II) We have proposed a formal notation (based
on the Data Description Calculus [FMW06]) to specify packet formats using
dependent types. Fields within a packet are specified by physical layout, de-
pendencies among field contents and constraints over the values of fields. Then
fields can be composed into packets by using packet combinators.

A suite of domain-specific notations and abstractions for protocol
logic implementation (Paper III)  We have designed a suite of notations
and abstractions for protocol logic implementation that closely resemble the
ones used in protocol specifications.

An approach to construct overlaid protocol stacks (Paper III)  We
have developed protocol combinators for constructing protocol stacks from
individually implemented protocol modules.

An embedding in Haskell (Paper III)  We have embedded the Protege
language in Haskell, profiting from Haskell’s features, especially rich static
typing and good support for modularity. Our results confirm that Haskell is a
convenient host for embedding a domain-specific language.

A compiler from Protege code to C code (Paper III)  We have de-
volved a compiler in Haskell that generates C code from Protege protocol
descriptions.

A runtime system for Protege (Paper III)  We have developed a run-
time system for the Protege language, which specifically targets resource con-
strained platforms such as bounded runtime memory consumption.

A case study using Protege in an industrial setting (Paper IV)  We
have implemented the Modbus protocol over TCP/IP and over a serial line, and
tested it using an industrial gateway. We demonstrate Protege’s advantages
for software productivity, easy maintenance and code reuse, and show that it
achieves many desirable properties of industrial embedded network software.

1.2 Outline of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 explains the
background of this work. Chapter 3 introduces domain-specific languages and
their implementation techniques. We give an overview of DSL technology and
present approaches to DSL design and implementation. We also discuss rel-
ated work in DSLs for network software in this chapter. In Chapter 4, we
identify and characterize protocol development practices in embedded systems
in general, and suggest particular implementation disciplines from which we
1.2. OUTLINE OF THE THESIS

have derived the design choices for our language Protege. Chapter 5 summarizes the attached papers presenting our language development. The first paper [WG08] describes a Data Description Calculus [FMW06] implementation in Haskell, on which we later built the packet layout specification in Protege. The second paper [WG09] proposes a small DSL for packet parsing and marshaling, including packet overlays for protocol stacks. This language is restricted to packet processing. We have used its code generator and libraries for the packet processing part of Protege. The third paper [WG11b] presents the domain-specific embedded language Protege as a whole, giving details on the embedding techniques we have used (a shorter summary of the language can also be found in [WG10]). The fourth paper [WG11a] presents a case study for Protege, implementing the Modbus [Mod06a] protocol for industrial process controllers. Chapter 6 concludes with a summary and directions for future work.
Chapter 2

Background

2.1 Network Software

Network software is a special class of system software deployed in communication networks to provide a set of well-defined services [LL92] that ensures connections and information sharing among a set of network devices. Complex functionality of network software is usually divided into smaller individual pieces, called protocols, to make design and implementation easier. A protocol is a set of rules and formats governing the interaction between communication peers. A protocol is a self-contained specification unit for particular independent functions: it provides its own services to other protocols and explicitly makes certain assumptions about others. Thus, protocols are ideally designed and implemented in isolation and can be modified without unduly affecting others. Protocols are typically organized according to a hierarchical layered architecture, called protocol stacks, to provide complex services. The hierarchy of protocols need not be linear. A given protocol may support multiple higher-layer protocols and may use several communication services provided by lower-layer protocols. Hence the protocols on a host form an acyclic directed protocol graph.

Protocols provide services to the upper layer while receiving services from the layer below, and protocols in the same layer normally have similar communication functionalities. As an example, Figure 2.1 shows the TCP/IP stack, presented as a 4-layer system [Ste94]. The application layer handles the details of the particular applications. Telnet, for example, provides a bidirectional interactive text-oriented communication service between two hosts. The transport layer protocols provide end-to-end communication services for applications in the layer above. One of the most well-known transport layer protocols is the Transmission Control Protocol (TCP) that supports reliable connection-oriented transmissions. Another prominent protocol in this layer is
User Datagram Protocol (UDP), which is connection-less and only supports simple messaging transmissions. The network layer protocols provide packet delivery service between peers in the network. For example, both IPv4 and IPv6 (Internet Protocol version 4 and version 6) can be used to select the next-hop host for an outgoing packet and pass it to the appropriate link layer protocol for transmission. In addition, the network layer provides error detection and diagnostic capability. The Internet Control Message Protocol (ICMP) for example, can be used to send error messages indicating that a host could not be reached. The link layer is the lowest layer in the TCP/IP stack. Protocols in this layer handle a host’s link, including the device driver and the corresponding network interface card for physically interacting with the media, e.g., cable, radio, etc. In this layer, the Address Resolution Protocol (ARP) is used to map an IP address to a host’s hardware address in the local network, and the Reverse Address Resolution Protocol (RARP) performs the inverse mapping.

<table>
<thead>
<tr>
<th>Application</th>
<th>Telnet, FTP, HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport</td>
<td>TCP, UDP</td>
</tr>
<tr>
<td>Network</td>
<td>IPv4, IPv6, ICMP</td>
</tr>
<tr>
<td>Link</td>
<td>ARP, RARP, PPP</td>
</tr>
</tbody>
</table>

Figure 2.1: The four layers of the TCP/IP protocol suite

2.2 Embedded Network Software

An *embedded system* is a combination of computer circuitry and software that is built into a product for specific purposes such as control, monitoring and communication. Whereas the embedded systems of the past were usually realized mostly in hardware, nowadays advances in chip technology have made it possible to program complex and pervasive software. Embedded software has become an application area of increasing importance [Lee00], which has taken over what mechanical and dedicated electronic systems used to do in the past. For example, a modern automobile may contain dozens of embedded systems to control processes within the car including brake balance control, air conditioning and ignition system, as well as inter-vehicle communication and driver assistance system. Without software, it would be impossible to computerize all of these features without the inclusion of a mass of complicated, error-prone electronics. Embedded systems are designed based on the concept of the microcontroller to deliver maximum performance for minimum resource usage. Microcontrollers integrate a CPU, a small amount of memory, and simple I/O functionality on a single chip, so their embedded software has to be developed
2.2. EMBEDDED NETWORK SOFTWARE

under resource constraints in terms of memory and sometimes energy. In addition, embedded software development usually has extended correctness and safety requirements: an embedded system might be controlling a machine that is expected to run continuously for months or even years without human intervention. The software of such systems thus has to be developed and tested or verified much more carefully than general-purpose computer software would require.

*Embedded network software* is network software for embedded systems, and most notoriously, communication protocol stacks for embedded systems. A number of particular complications arise from either the nature of embedded network software or from common practices in the development process.

**Constrained resources** Embedded software often has to fulfill rigorous resource constraints, as embedded systems are usually resource limited in terms of memory and energy. For example, tiny embedded systems like wireless sensor nodes only have a few kilobytes of RAM and code memory, and they are powered by batteries. Minimizing resource usage becomes a priority in embedded network software development, hence developing embedded software implies more attention to implementation-level details than when developing a desktop variant.

**Need for customization** Embedded systems are typically application specific systems, and embedded network software can be consequently highly customized. There are protocols which were not particularly designed for, but are broadly used in embedded systems. In such a case, a universal flexible implementation covering every detail is often not desirable for embedded network software. The implementation is instead specialized to what is actually needed, to save memory space and computing power. For example, TCP/IP provides Internet connectivity for computers in general, but its implementation in embedded systems is usually limited to only the TCP/IP kernel to make good use of the limited resources. In the case of an Internet enabled power system controller for instance, TCP packet fragmentation is not needed at all, because transmitted commands are known to take up only a few bytes. For dedicated embedded system protocols, it is essential to find a good compromise between *required* ("must") and *optional* ("may") functionalities, and a good protocol implementation is a tailored one. The automation protocol Modbus [Mod06a] is a good example: it has 21 standard public functionalities, but in practice, only a few of them are used to run on any particular machine.

**System dependency** Embedded network software is usually an integrated part of a device’s operating system (if one is used), and tightly coupled to hardware and other operating system facilities. It is therefore difficult to port an implementation from one architecture to another. For example, a protocol im-
plementation for the wireless sensor platform Tmote Sky [Mct05] is not easily ported to a platform that uses Contiki [Dun07] instead of TinyOS [LMG+04], or even no operating system at all. In addition, a number of non-functional aspects, such as real-time properties and concurrency, are essential for a protocol implementation. For example, the choice between different scheduling algorithms like Earliest Deadline First (EDF) [LL73] or Least Slack Time First (LSTF) [RS94] affects the protocol’s real-time properties. Thus, these non-functional aspects normally need to be an integral part of the embedded implementations [Lee02].

**Low-level coding** Because of its tight integration into the embedded operating system (OS), embedded network software is developed using low-level programming languages such as C or even assembly. As a result, these implementations cannot benefit from achievements of modern programming language design, like object modeling, static type systems and automatic memory management. Development easily becomes tedious and error-prone. For example when parsing packets in C, it is common to extract field values using low-level operations, such as offsets, bit-shifts and bit-masks. Lacking suitable abstraction mechanisms for packets, a large amount of similar code usually needs to be written and is often created using copy-and-paste followed by small modifications. This is a common source of errors as it is easy to get some details wrong, and such code will also become troublesome to maintain after a short time.

### 2.3 Existing Support for Protocol Stack Implementation

To maximize efficiency, or to meet the constraints imposed by hardware limits, programmers typically implement a protocol stack in a monolithic way. For each protocol, assumptions about the existence and the details of the design of related protocols in the stack are made, and the programming logic is deeply woven into the fabric of the whole implementation. The implementation uIP [Dun03] for embedded systems implements TCP/IP as a *monolithic stack*, i.e. without any layer structure. The code is extremely tightly coupled and leaves out certain TCP mechanisms from the application interface to minimize the resulting code size. For example, the uIP implementation does not support soft error reporting, which is not essential for host-to-host communication. In the long term, this approach turns out to be impractical, because of growing interdependencies of performance and functionality aspects. Moreover, programming in this way is exceedingly difficult even for experienced programmers.
2.3. **EXISTING SUPPORT FOR PROTOCOL STACK...**

In *layered* protocol stack implementations, distinct layers are responsible for different services of the communication. Protocols can be implemented in a modular fashion to reduce complexity and to make reuse and configuration possible. Individual layers are easier to both write and verify than complex, monolithic stacks. Since the dependencies among protocols are explicit, it is easy to build special-purpose protocol stacks by assembling existing protocols, while it is often difficult to tailor a monolithic stack to specific application requirements without rewriting large portions. In a layered protocol stack implementation, any unneeded protocol can be removed or replaced to reduce the surplus functionality and tailor the networking parts to application needs. For example, for applications that do not require reliable connection-oriented transmission, TCP can be removed or replaced by UDP.

The price for these advantages when following the modular layer structure in the implementation is a considerably increased overhead in protocol processing [HP91]. Protocol boundaries in the stack will impose a number of separate function calls, and inhibit direct access to other layers' internal data structures. The challenge is to combine the modularity of layered protocol stacks with the good performance of monolithic protocol stacks. The solutions that have been proposed to this problem all take one of two approaches.

*System-based approaches* provide systematic software architectures for constructing protocol stacks, in order to clearly express the modular structure and protocol layer composition. Following such a system-based optimization approach will produce a well-structured implementation, while enabling a few optimizations that are specific to protocol stacks. For instance, Integrated Layer Processing [CT90] proposes rewriting existing protocol stacks such that the processing done by the different layers is pipelined, avoiding message copies every time a layer boundary is crossed. The *z*-kernel [HP91] uses a particular thread concept: an initial up-call and one reserved thread per packet. Chameleon [DOH07] is an architecture for organizing protocol stacks in sensor networks, which aims at isolating low-level packet processing from other aspects of protocol stack implementation. Headers are removed and attributes for further packet processing are introduced.

*Language-based approaches* go one step further and use compilation techniques to reduce the performance penalty for protocol layering. A compiler can automatically perform protocol-oriented optimizations based on specific common behaviors of protocols. For example, the last action taken in a message output function of a protocol is always to invoke the next lower layer's output function. When the lower output function returns, the original output function is done and also returns. Responding to this behavior, Morpheus' *short-circuit return* optimization [AP93] saves one jump assembler instruction per protocol layer, i.e., the output functions with no further work are bypassed in the sequence of procedure returns. For the lower output function, there is no need to jump back to the current output function after it finishes it task, instead it should jump back to the current one's caller. Another example is...
that protocol stack processing may consist of some frequently executed code segments. For instance, one message output function might always call another message output function in the next lower layer. Promela++ [BME98] offers programmer-annotated abstractions for explicitly specifying this kind of execution path as a fast path, for a more efficient composition of multiple protocol layers. According to the fast path, procedure calls can be inlined to form a tightly packed single function which removes almost all call overheads. The fast path also enables further optimizations by the C compiler since inlining allows the C compiler to optimize across function boundaries using standard techniques, such as copy propagation and constant folding.

That said, it is clear that compilation techniques and language-centric development contribute to produce efficient code for the domain of network software. Taking this language-based approach further requires creating an entire language that is especially tailored to the needs of protocol stack implementations, a Domain-Specific Language.
Chapter 3

A Language-Based Approach

Domain-Specific Languages (DSL) have been around for decades and are commonly used in many disciplines. Rather than a general purpose language that can be used for any kind of software problem, a DSL focuses on a particular problem domain. A well-chosen DSL can make code easy to understand, easy to write, easy to modify, easy to reuse, and less likely to breed bugs, thus improving the productivity of those working with it. DSLs also improve the communication between programmers and the domain experts, as a program in a DSL can act as both executable software and a description of the domain problems.

3.1 Domain-Specific Languages

A Domain-Specific Language (DSL) is a programming language dedicated to a particular application domain or problem [Hud97, MHS03]. It offers an appropriate but restricted suite of notations and abstractions for expressing the domain’s concepts at a high abstraction level, and is usually more declarative than imperative. The person supposed to use a DSL is the domain expert rather than the skilled programmer. With DSLs, domain experts can concentrate on what to compute rather than how to compute. Repetitive tasks and common programming patterns are captured as language constructs, their implementations can use particular domain-specific optimization techniques and rely on shared library code. In this way, a DSL largely automates the process of turning a specification into an efficient executable – the expert is writing code without realizing that she is actually coding, and specification and implementation are consistent by construction. In brief, a DSL mediates a collaboration between domain experts and programmers, leading to software that is more usable, more portable, more reliable, and more understandable.
Domain-specific languages have been a popular way to shorten the distance from ideas to products in software engineering for a long time already. The benefits of DSLs have been delivered in various computing domains. Classical and well-known examples include SQL for database queries, VHDL for hardware design, Matlab for technical computing, HTML for inter-linked web content, or LaTeX for typesetting. More recently, DSL technology has become more widely used in various profitable commercial domains, such as encryption and pricing of financial products. The Cryptol language [Lew07] is a relatively new DSL successfully distributed as a commercial product, which enables constructing cryptographic hard- and software with ease, reliability, and high assurance. In the financial world, Lexifi [PES00] provides a complete solution for designing, pricing, analyzing, and processing complex financial products. Their products have proved useful for a broad customer community, e.g., salespeople, quantitative analysts, risk managers and middle office professionals, working in different fields: investment, asset management and private banking.

![Figure 3.1: The payoff of software development](Hud97)

Typically, DSLs are small languages with limited control structures and simple syntax and grammar. However, designing and implementing a DSL from scratch is fairly difficult and time-consuming [Hud97]. Typical standard tasks for creating any new programming language are also required for DSL development: designing the syntax and type system, writing a parser, writing the code generator or interpreter, etc. Furthermore, the initial design might turn out not to be suitable, implying that all of these steps have to be repeated. During development, a DSL also tends to grow, as new modules, procedures, and data structures are found to be necessary. Consequently, all kinds of difficulties associated with language evolution will arise: redesigning the grammar, rewriting the parser and the code generator. But once a DSL has become usable, domain experts can express requirements much more directly, resulting in shorter time-to-market and fewer errors. Figure 3.1 shows that DSL development is a long-term investment: the initial cost of a DSL's design and
3.1. DOMAIN-SPECIFIC LANGUAGES

development is high, and the gain in productivity and reduction of maintenance costs only come later.

3.1.1 DSLs vs. Libraries

In a general purpose language (GPL), a library is a collection of resources used to perform specific, well-defined operations. Libraries contain code and data and provide pre-defined common modules as services to multiple programs for code sharing and modular programming. As the implementation details are hidden by the incorporated libraries, all programs can be more modular, faster to write, and easier to update. DSLs and libraries both facilitate software development, e.g., they are both approaches to software reuse [Kru92]. As Mernik et al. declared [MHS03], any GPL can act as a DSL in combination with an application library.

From the programmer's perspective, the main difference between using a library and a DSL is that in the first case, programmers have to have both domain knowledge and some programming skills in the respective GPL. For example, to implement a simple frame using a GUI library in Java, programmers need to have considerable general knowledge about Java: instantiating objects, declaring variables, calling methods, etc., which is outside the GUI development domain.

From a technical point of view, DSLs allow much more freedom to implement domain-specific notations than user-definable syntactic elements of a GPL. Also, typical constructs and abstractions of the problem domain may not always be mapped straightforwardly to functions or objects in a GPL. As an example, consider a data format for ad-hoc data processing [FMW06]: the dependent sum type expression \( \Sigma x: \tau_1, \tau_2 \). This expression allows to constrain a field using the value of a previous field. Variable \( x \) can be used to refer to the value of the first field \( \tau_1 \) when specifying \( \tau_2 \). For instance, \( \Sigma x: \text{Char}. \text{String}(x) \) specifies a first field of type \text{Char}, followed by a second field of type \text{String} which is terminated by the character presented in the first field. As a whole, the expression describes a sequence of characters started and terminated by a same character. This would be hard to specify in the C language. A DSL can also be presented to the users in various ways, and provide a range of tools for the user, such as pretty printing and graphical input. For instance, a spreadsheet application like Excel can be considered as a visual toolbox for business users. A DSL also embodies domain knowledge accumulated from experienced programmers. For example, it can enforce good implementation disciplines that take advantage of the specifics of the domain. It helps write and maintain programs in the domain quickly and effectively. Since DSLs precisely capture the semantics of the application domain, i.e., no more and no less than what is needed, they also enable more properties of programs to be verified and optimized. For example, in the language TAP [Mcg04], the definition
of an application protocol can be directly validated, and an implementation
generated from the same textual definition.

3.1.2 Designing a DSL

Following Consel [Con04], the design of a DSL is a subjective process with
certain similarities with a craft.” He proposes a methodology for DSL design
in systematic steps. The first step is to analyze a program family which exhibits
implementation expertise for domain-specific optimizations, and enables to
extract common program patterns for designing syntactic constructs. After
identifying a program family, developers can develop a library and suggest
domain specific operations, values and states for defining a domain abstract
machine. Then the main ingredients formed by the abstract machine and the
common program patterns lead to the design of a DSL.

Identifying a program family A program family is a set of related pro-
grams with enough similar characteristics. Normally, there is a large set of
programs devoted to a particular domain. If the scope of computations is too
wide, analysis will lead to a general purpose language. Identifying a program
family is an intuitive process. To delimit the scope of computations, it is sug-
gested to restrict the family either horizontally or vertically. In the protocol
implementation domain, one could for example consider to address only the
protocols in the application layer (horizontal) [Mcg04], or to focus only on the
TCP/IP suite (vertical) [KKM99]. The specific analysis of a program family
will lead to identifying repetitive program patterns, which can then be ab-
stracted into syntactic constructs.

From a program family to a library A program family naturally leads
to the development of a library which encapsulates identified domain-specific
operations. These operations become reusable software components abstracted
over commonalities. Some of the operations will expose different versions and
variations of implementation, to be translated as parameterizations. And the
objects manipulated by the library functions may suggest a set of domain-
specific objects.

From a library to an abstract machine The abstract machine defines
the runtime model for a DSL, which is a combination of a domain-specific set
of instructions and states. Library functions explicitly or implicitly pass argu-
ments to represent some notion of states, which not only model the run-time
environment, but also ensure the safety of some instructions. The common
domain-specific operations in the library are turned into domain-specific in-
structions and states. Domain-specific instructions are similar to library oper-
3.1. Domain-Specific Languages

actions in terms of dynamic semantics, but they are generated by a compiler or
invoked by an interpreter, rather than directly used by a programmer.

From an abstract machine to a DSL  After defining the abstract machine,
actions are staged into two categories: either they become compiler actions
which correspond to the static semantics of a language, or they become runtime
actions which correspond to the dynamic semantics. Finally, domain specific
optimizations are applied, following the implementation expertise according to
the analysis result of the program family.

3.1.3 Implementing a DSL

DSL constructs can be recognized and interpreted using a standard fetch-
decode-execute cycle [MHS03]. This is an attractive option if the language does
not require high execution speed or if it provides highly dynamic features. DSLs
implemented in this way can be easily extended. An example is page rendering
from HTML in web browsers: content can be displayed while still downloading
the remainder, and network latency dominates the performance in most cases.

DSL constructs can also be translated into constructs of a target base lan-
guage and possibly into library calls. Normally, high-level programming lan-
guages are implemented using a compiler. DSLs implemented in this way can
be statically analyzed and optimized. A classical example is the parser genera-
tor Yacc/Bison [Joh75] and its various successors. A syntax description in the
form of a grammar is translated into suitable parsing functions to produce an
abstract syntax tree, including an analysis of the grammar for its properties
and potential problems.

A DSL can be implemented using a preprocessor which simply translates
DSL constructs one by one into constructs of a base language. Macros and
subroutines are typical language extension mechanisms used for DSL imple-
mentation. DSLs implemented in this way can be limited by the processor of
the base language.

Sometimes, a GPL compiler or interpreter can be extended with domain
specific optimization rules and/or a special code generator. Some compilers
provide a suitable interface to influence their internals for this purpose. For
example, the Glasgow-Haskell Compiler enables user-defined rewrite rules for
domain-specific optimizations [PTH01].

Embedding is a technique to implement the domain-specific syntax and
semantics inside another existing language, which is called the host language
and is normally general-purpose and fully fledged. DSL constructs are encoded
by the host's data types and operators. The domain-specific semantics is rep-
resented in the host logic. Both domain-specific constructs and non-domain-
specific functions and operations of the host language can be part of the new
language. Thus a domain-specific problem can be expressed in the idiom and
at the level of abstraction of the problem domain. As the domain semantics
and syntax are captured concisely, users do not need to be familiar with the host language [Hud96], DSLs implemented in this way can inherit most of the design decisions from the host language, but can also be limited by the syntax of their host.

3.1.4 Domain-Specific Embedded Languages

A great deal of the startup cost of a DSL development usually goes to non-domain-specific parts, before tailoring features towards the domain of interest. For instance, arithmetic expressions are necessary for almost all kinds of programming, and parsing and type-checking them are tedious standard tasks. Embedding can be a convenient shortcut to implementing a DSL, because design and implementation decisions for these non-domain specific aspects can be inherited from the host language [Hud96]. Thus, the development can focus exclusively on the domain problems that can be solved much faster than for a full DSL. This approach lowers the bar of DSL development and makes it applicable more broadly and more effectively, by avoiding the typical hurdles of programming language implementation.

A DSL that is implemented using an embedding approach is called a Domain-Specific Embedded Language (DSEL) [Hud97, Ell99]. It embeds domain-specific concepts and features into an existing host programming language to take advantage of the host’s infrastructure and tools. Like every language, a DSEL has its syntax and semantics, and it can be fitted with a type system, interpreter or compiler, and runtime system. A common characteristic of all DSELs is that their syntax is embedded as a subset of the host language, and it gives a similar look-and-feel of special syntax as a DSL does. A syntactically correct program in a DSEL is thus formed according to the rules of the host language (a syntactically correct program in the host language) and all syntax-related tools of the host language (especially the entire parser) are reused.

In order to realize the semantics, different embedding approaches exist and have their particular advantages and drawbacks. The abstract syntax of the DSEL can be reproduced as data in the host language, which is called a deep embedding. Values in a DSEL program are represented by a syntax tree; its functionality (the DSEL semantics) is constructed from the syntax at a later stage of processing. Another approach, called a shallow embedding, is to realize not the DSEL syntax, but the DSEL semantics in the host language. In this case, values of the DSEL directly map to values in the host language, and DSEL functionality is directly constructed by the embedding of the syntactic constructs, as functionality realized in the host language. A DSEL with a shallow embedding can be essentially considered as a library that augments the host language [EFM03].

One attractive property of shallow embedding is that all of the features of the host language, libraries, type system, etc., are available in the DSEL
3.1. DOMAIN-SPECIFIC LANGUAGES

directly. However, there are important arguments against embedding semantics in some cases. Sometimes the ideal host language is not the dominant language in the target domain, either because of the performance [EFM03] or system integration [BCSS98]. For example, C is traditionally used in system programming. C code to switch on an LED means simply to set a value of a special register in hardware. This is easily done in C, but may be cumbersome to express in the host language Haskell. In such cases, a code generator is needed to generate code in a target language from the abstract syntax in the host language, a major case for deep embedding. Deep embeddings manifest the syntax of the DSEL as an abstract syntax tree data structure which can be conveniently manipulated and translated. One important subclass of deeply embedded languages has the code generator implemented by the host language as well; this is called embedded compilation for domain-specific languages [LM99]. As a result, a function of the DSEL will not directly compute values when run; instead, the result is another function declaration in form of the target language. It is also possible to have several different semantic interpretations for one program represented in the host language. Deep embedding is semantically more flexible.

Functional programming languages [Hug84, Wad92] are in general considered as suitable host language for embedding [Hud97]. For instance, Haskell has a number of advantages over other languages. First and foremost, Haskell supports higher-order functions. In consequence, a DSEL construct in Haskell can be thought of as a higher-order algebraic structure, a first-class value that has the appearance of special syntax. Second, Haskell has light-weight layout-based syntax, a property which directly affects the readability of a DSEL implemented in it. If the syntax of the host language is too heavyweight, for instance containing an excessive amount of keywords or parentheses, the embedded DSL will inherit this and possibly require preprocessing or extending the compiler to make code more readable. Third, the monad concept in Haskell allows one to cleanly define any desired control structure in a procedural programming style. Finally, its static type system allows very sophisticated constraints to be placed on the use of DSEL components and their relationships with other parts of the language, and to detect type mismatches at compile time.

Haskell has previously been successfully used to implement a range of DSEls for different domains, e.g., a pretty-printer for pretty-printing [Hug95], Parsec for parsing [HM96], Lava for hardware design [BCSS98], Lexifi's MLFi for evaluating financial contracts [PES00], Pan for image manipulation [EFM03], Cryptol for code encryption and decryption [Lew07], and Paradise for generating Excel add-ins [AMS08].
3.2 Domain-Specific Languages for Network Software

3.2.1 Why a DSL for Network Software?

There are several potential benefits to employing a DSL in network software development.

Expressiveness and readability  In protocol stack implementations, buffers, timers, and packets are domain-specific concepts that are rarely supported by general purpose languages with existing libraries. A DSL can offer an appropriate suite of high-level constructs and abstractions with a natural vocabulary for manipulating specific concepts in the domain. Consequently, programs are more expressive, and also more straightforward, i.e. easier to read and develop.

Automatic code generation  Implementing protocol stacks is sometimes tedious and involves repetitive tasks, for example, parsing and marshaling packets. Implementation parts can for instance vary because of different packet formats, but generally follow one and the same pattern. Typically, this leads to implementations written using copy-and-paste followed by small modifications, a major source of errors as it is easy to get some details wrong. With a DSL, it is possible to generate this repetitive code automatically from specifications in a safe and systematic way.

High-level abstractions  A DSL can offer guidelines and built-in functionality for constructing programs. The support routines can either be an integral part of a language, as language primitives, or, in some cases, not visible at the source code level but instead automatically applied where needed. The compiler generates implementation details by converting the explicitly specified code, adding common program patterns and necessary technical boilerplate code automatically. For example, a compiler for a protocol implementation language might be able to generate the appropriate locking transparently for multiprocesssing, so that protocol source code is independent of the degree and style of multiprocesssing. DSLs provide a clear and elegant way to write better programs with less code, and less code also means less programming space for errors.

Code sharing  The compiler can automatically supply code and data structures, and make implementation decisions that the programmer would otherwise have to specify. In the protocol development domain, we can often find generic functions shared by different protocols. For example, protocols fragment and reassemble packets: any TCP segment whose size is greater than the Maximum Segment Size (MSS) needs to be fragmented into smaller pieces.
and reassembled upon receipt. IP does exactly the same, using the Maximum Transmission Unit (MTU) to limit the size of its IP datagrams. This commonality can for instance be expressed by dedicated subtypes in a DSL, i.e. a protocol of subtype Fragmentable has to specify a maximum fragment size, and the compiler will check that the implementation conforms to this type, and introduce utility functions to fragment and reassemble packets.

**Code quality** DSLs are a good medium to enforce constraints, because they increase the amount of context available to the compiler, and restrict the design space, realizing a good implementation discipline. General-purpose languages, in contrast, would allow programmers to use their own algorithm in place of a support routine, and thereby circumvent domain-specific constraints on the data structures. And of course, all kinds of subtle mistakes can happen when doing so. For instance, the out-of-bounds array access, a typical programming error, is likely to happen when manually extracting erroneous values from a packet field in the buffer, due to the flexibility on accessing memory. A DSL could avoid it by design, because it will not offer any explicit low-level memory access or pointer arithmetic.

**Optimizations** The domain-specific constructs, along with the enforced design philosophy, put the compiler in a better position to make domain-specific optimizations. As we have already mentioned in Section 2.3, protocol-oriented optimizations can be achieved automatically. In contrast, conventional languages and their optimization techniques only offer limited opportunities related to protocol stacks, and protocol-specific optimizations in general can only be achieved manually.

**Static analyses** A DSL can guarantee syntactically and semantically unambiguous formal descriptions of the domain, and thereby allows static analyses of various program properties. For example, if a DSL requires the protocol developer to specify the control flow of a protocol layer in a formal semantic model, the specification can be verified by a validator against programmer-specified safety requirements. In particular, the programmer can specify that the protocol never deadlocks by requiring that some particular state in the protocol is reached infinitely often (done in Promela++ [BME98]).

### 3.2.2 Existing DSLs for Network Software Development

During the past decade, DSLs for protocol implementations have been studied extensively. Some languages have been designed with the primary goal of making protocol implementation easy; providing flexible structure to construct protocols. Some languages are dedicated to particular tools and groups of protocols. For example, Z2z [BRLM09] is for specifying network protocol gate-
ways. Some languages such as Prolac [KKM99] and Morpheus [Abb93] provide
constructs to make composition and decomposition of complex protocols easy.
They are intended to facilitate programming, reusability and optimizations.
These languages are not based on a formal protocol model, but rather make
it easier to translate informally specified properties of protocols from natural
language into code.

Some languages are designed with the goal of bridging the gap between
verification and implementation. Such languages are sometimes based on a
formal protocol specification model, in strict notations which are intended for
automatic model checking or manual verification. Protocols specified in these
languages can be directly verified. Protocols can also be specified by a more
permissive and flexible protocol specification, from which a model specification
is generated. The model abstracts away the details obscuring the properties
of the protocols and places strict limits on the actions of the specifications.
For example, Promela++ [BME98] can produce input for a model checker
and generate executable C code. In these languages, packet processing is ei-
ther constructed in a simple way (as in TAP [Mcg04]), or exported from an-
other language (for instance, RTAG [And88] uses C, and Z2z [BRLM09] uses
Zebu [BRLM10]).

To our knowledge, there is no research apart from ours that targets DSLs
for protocol stack implementations including packet processing, protocol logic,
and protocol composition with the particular consideration of embedded sys-
tem constraints (energy and memory efficiency and cross-layer optimizations).
Some earlier protocol languages have provided important ideas and influenced
the design of Protege. In the following, we discuss these protocol languages
and their influence on our work in more detail.

Morpheus

Morpheus [AP93, Abb93] is an object-oriented DSL for protocol implemen-
tation. Its protocol abstractions and protocol-oriented compiler optimizations
are based on the experience gained from the z-kernel project [HP91]. Although,
the design and implementation of Morpheus remained incomplete (there is no
grammar and no compiler), our design has adopted important ideas proposed
by Morpheus.

In Morpheus, a protocol is built up out of sub-protocols, which enables a
large degree of modularity and structures the implementation. Sub-protocol
code can be reused in multiple protocols, and modules can be added and re-
moved easily for customizations. A sub-protocol is characterized by its particu-
lar functionality, or "shape": Morpheus distinguishes three shapes multiplexer,
worker and router, and arbitrary protocol functionality is realized by com-
posing one or more sub-protocols. The classification into shapes mainly aims at
improving code sharing and data structure reuse. Using Morpheus, the pro-
grammer implements a customized sub-protocol by deriving it from the base
class for the different shapes and implicitly using predefined code and data
structures. The shapes model enforces clean protocol implementations, but it
also puts many constraints on the protocols. As a result, some existing proto-
colls may not be implementable in Morpheus, e.g. multi-cast protocols do not
fit into the framework of Morpheus.

Protege's basic abstractions are heavily inspired by Morpheus. Similar to
Morpheus's shape concept, our protocols are represented as instances of a par-
ticular protocol class, e.g., a network protocol. Furthermore, we use a hierarchy
of type classes to classify protocols with different types of functionality and for
further extension. For example, our network protocol class is a subclass of the
base protocol class with extra operations such as getting a network connection.
This design is also similar to the protocol signatures used by FoxNet [BHL01]
which is the TCP/IP protocol stack implementation written in SML.

Morpheus also concentrates on reducing per-layer overhead by using protocol-
oriented compiler optimizations. for both latency and throughput. Here, we
have adopted the Morpheus idea of integrated layer processing for Protege.

Promela++

Promela++ [BME98] is an extension of the Promela [Hol91] protocol valida-
tion language. It was designed to support both validation and implementation
for user-level layered protocols and protocol stacks, which allows automatic
verification of protocol correctness and is compiled to C.

Protege has adopted ideas of Promela++ for its design of protocol logic.
In Promela++, each protocol has to specify the protocol state and the packet
headers it uses, and the major control structure is based on finite state auto-
matata. Protocol logic is described in an event-based model. An event may
contain a reference to an outgoing or incoming packet. Protege also uses this
approach of protocol states and events.

Protege and Promela++ take different approaches to layering, even though
Promela++, like Protege, intended to reduce the performance penalty for lay-
ering, and both provide explicit language constructs to encapsulate protocols
and compose them into stacks. Promela++ runs each protocol in a separate
thread, and adjacent protocols communicate using FIFO packet queues. As
discussed in more detail in Section 4.3, Protege uses one thread per packet
and takes the z-kernel [HP91] approach for encapsulating protocols, a uniform
set of abstractions for protocols, and up-calls [HP91] to handle concurrency.
The reason is that packet queues between protocols would require consider-
able buffer space and increase the memory footprint, which is undesirable in
an embedded system.

Promela++ is designed to be very similar to C, to make it easy to in-
tegrate low-level C code into a protocol implementation. However, because of
modifications and additional restrictions, Promela++ code appears to be more
difficult to write than pure C; for example, when using primitives for timers
and memory allocation. When Promela++ code is converted to Promela for
validation, all low-level code for the implementation details will be hidden,
and the essential characteristics of the protocol remains, e.g., interactions of
processes.

Prolac

Prolac [KKM99] is a statically-typed, object-oriented language for network
protocol implementation. The design goal of Prolac is to make protocol imple-
mentations readable, which is also one of our goals. Like Protege, Prolac aims
at a light-weight syntax and compiles to C.

The basic units of program in Prolac are modules, which are associated with
fields and methods and compare to classes in an object-oriented language.
Modules may extend other modules through inheritance, and may provide
new definitions for the methods of their superclass. Prolac tries to realize
a declarative programming style, e.g., method bodies are mostly expressions
instead of statements. In addition, users are encouraged to break up large
methods into smaller pieces with maybe no more than 5 lines of code and
meaningful names, to make protocol implementation easier to understand.

Modules are manipulated by module operators which give information to
the compiler that the compiler can use to improve protocol efficiency. For
example, the compiler understands an Inline directive for function calls.

A drawback of Prolac is that it lacks abstractions for the protocol domain.
Primitives such as timers and buffers are not provided by Prolac, but instead
introduced by uninterpreted C code that is copied into the generated code, and
memory management is completely left to the programmer (using kmalloc).
A case study of a Prolac TCP implementation [KKM99] included 1300 lines
of Prolac code supported by 700 lines of C code including timers and other
operating system mechanisms.

TAP

The most recent DSL for protocol implementation which is comparable to our
approach is TAP [Mcg04] from 2004. TAP is a DSL for describing asynchronous
message-passing protocols, similar to our work. While our work includes proto-
col layering, TAP concentrates on modeling single application-layer protocols
that use TCP/IP (for which a runtime interface is provided by TAP).

The main novelty of TAP is to provide two different execution models: an
abstract execution model for design and correctness verification, and a concrete
execution model for protocol implementation. The two models are shown to
be equivalent, so a protocol verified under the abstract model will preserve the
intended behavior in its concrete interpretation, which is based on a compiler
generating C code.
3.2. DSLS FOR NETWORK SOFTWARE

The TAP notation is based on Abstract Protocol notation (AP) [Meh98], extended with timer facilities. AP is an abstract formal notation for specifying protocols, which facilitates formal reasoning about protocols, but the set of provided language abstractions is very restricted. Therefore, protocols with traditional informally written RFC-style specifications are sometimes cumbersome to express in TAP.

The strength of TAP lies in the modeling of protocol logic. It describes the interaction as a message exchange between several sender and receiver processes that run concurrently, on the same machine or on different machines connected in the network. Each process consists of local states and a set of actions, which can be time interval bounded to model timing-related behavior. An action describes the computation performed by a process using send statements and assignment statements. C functions can be used directly.

Packets in TAP are specified as sequences of fields which are either an integral value (constant or variable) or a byte array representing an uninterpreted payload. Specifying a constant field has the effect of filtering out invalid messages, similar to an equality constraint on a field in Protege. TAP cannot describe message fields that have other, more complex types, or that should satisfy more complex constraints. In this case, the TAP code has to include hand-written C functions for parsing and marshaling. Such extra code will be integrated into the automatically generated C code to parse and marshal a complete message.

As we have seen in this overview, a number of existing languages for protocol stack implementation provide interfaces to include low-level C code, and need it to realize their functionality. Some languages stay close to C for better overall integration, while others try to keep the abstract level of specification, sometimes using languages more formal than the established RFC style. In the next chapter, we discuss practices and suggestions for the specification and implementation of protocols, to motivate the features of our DSL Protege.
Chapter 4

Practices for Implementing Network Software

In this chapter, we identify and characterize protocol development practices in embedded systems and in general. The analysis of protocol specification and implementation techniques helps to identify crucial features and typical problems in protocol development. Our discussion is geared towards our language Protege. We discuss the protocol specification style, propose to structure code in a favorable way for automatic code generation, and identify and discuss necessary features of the Protege runtime system. However, our analyses and the implementation disciplines we propose can also be useful to improve maintenance and code reuse in conventional implementations.

We use the Modbus protocol [Mod06a, Mod06c, Mod06b] as a running example. Modbus is a widely used industrial automation protocol, which we have used for a case study (see the attached Paper IV [WG11a]). It is a typical industrial protocol with rich functionality, but relatively simple data structures. The core functionality of Modbus is given as the Modbus application protocol [Mod06a], which specifies a number of functions to read and manipulate device state. There are 21 functions defined in the standard, but many non-standard extensions exist. As an independent layer, an underlying communication infrastructure is specified in two variants, serial port or TCP [Mod06b, Mod06c], making the resulting protocol stack either ModbusSerial or ModbusTCP.

4.1 Protocol Specification and Analysis

Protocols are typically standardized and defined in their protocol specifications. Programmers implement protocols according to their specifications. The protocol specification can be broken into two conceptual parts, called the packet
specification and the protocol logic specification. For each part, there is an established style of specification that is commonly used.

4.1.1 Packet Specification

A packet specification describes messages exchanged on a network. The packet specification consists of the physical organization of a message in terms of the packet format, usually presented using figures. It also consists of dependencies among field contents as well as constraints over the values of some fields, usually given as informal explanations written in a natural language. As an example, Figure 4.1 shows the general Modbus packet format [Mod06a]. Physical packet layout is specified as bit-length for all header fields in the figure. We can see that the Function Code field takes 8 bits. Additional constraints and fields dependencies are specified in the accompanying text. For example, a constraint on field Function Code is: Valid codes are in range of 1 ... 247 decimal. An extreme example of a dependency between fields is that both the content and even the physical layout of field Other Fields depends on the value of field Function Code. For example, a function code 03 in Modbus stands for “Read Holding Registers”; and the Other Fields field in the reply message will indicate the read data (payload) size in one byte field Byte Count. In a request message with Function Code 07 (for “Read Exception State”), the Other Fields of the client (request) message is empty and thus takes 0 bytes because no additional information is provided.

![Figure 4.1: A general Modbus packet](image)

4.1.2 Protocol Logic Specification

Protocols include the behavior of concurrently executing processes, which are specified in protocol logic specification by procedure rules. These rules are used for governing interactions by message exchanges between communicating peers. A convenient abstraction for protocol logic is to specify it by a Finite State Machine (FSM). Protocol behavior is expressed in terms of protocol states and state transitions. Protocol state symbolizes the assumptions that each process in the system makes about the others. States transitions define what actions a process is allowed to take, which events it expects to happen, and how it will respond to those events. Protocol processes execute a variety of actions during processing, for example, data manipulations such as checksum
4.1. PROTOCOL SPECIFICATION AND ANALYSIS

computation, allocating buffers, scheduling resubmission tasks, setting time-out timers, manipulating and looking up protocol states, reassembling packets, etc. All these actions fall into four abstract categories: user calls, sending packets, receiving packets, and timer events [Var05], used as input and output for the protocol FSM.

As an example, Figure 4.2 shows the finite state machine which specifies the Modbus client mode behavior. Initially, the client is in state Idle, the only state in which a request can be sent. In case a unicast request is sent, the client goes into the Waiting for reply state and starts a time-out timer. Only when the expected server answers the request does the client go to the state Processing reply; unexpected answers are dropped. If no reply is received before the timer expires, the client goes back to the Idle state directly, otherwise the reply packet is processed (and may cause a retry if an error occurs).

Figure 4.2: Client state diagram

Protocol logic also includes stated assumptions about the environment in which the protocol is expected to execute. There are rules that determine the interaction between peers. For example, Modbus is a client-server protocol, i.e., a server always listens in the network and gives responses to clients which try to connect. When a legal request from a client is received, “the server echoes to the request the original function code”. There are also rules that define how a protocol interacts with adjacent protocols. For example, the Modbus application protocol’s “communication can be done as well on serial line as on an Ethernet TCP/IP network” [Mod06a].

4.1.3 Analysis

A protocol specification defines the externally observable functionality of nodes in a network. However, protocol specifications usually leave large space for
Implementation decisions. Thus, the same specification could lead to rather
different implementation results. Reasons for this include:

- Specifications contain both mandatory and non-mandatory requirements,
  traditionally indicated by the use of words must (for mandatory func-
tionality), should (for desired, but unnecessary functionality), and may
  (for truly optional features). Thus, implementors can freely make imple-
mentation decisions according to the requirements of different scenarios.

- Specifications do not describe the method of implementation. For exam-
  ple, a timer might be required for a response timeout, but an implementor
  is free to choose the timer implementation (for instance, whether a timer
  queue or a timing wheel is used to realize timers [Var05]).

- Specifications do not include machine(or system)-dependent protocol de-
  tails. For example, Modbus uses a big-endian representation for addresses
  and data items. The implementation for 8 bit micro-controllers and for
  16 bit micro-controllers could be different.

4.2 A Software Engineering Perspective

Before looking at implementation practice for embedded network software, we
would like to add some thoughts from a software engineering perspective. Soft-
ware engineering defines a methodology and process models for development,
and categories for reasoning about software properties: the dual nature of re-
quirements, and different categories of complexity (or "difficulties" [Bro87]).

As in any kind of software system, there are two types of complexities to
be found in an embedded network software system: essential complexity and
accidental complexity [Bro87]. Essential complexity is inherent in the nature of
the problem to be solved, and therefore unavoidable. All reasonable solutions
to a problem will be complicated in the same way. In contrast, accidental com-
plexity arises during the software development process. It is not inherent and
it is caused by the approach chosen to solve the problem. In embedded network
software development, the essential complexity derives from the desired ser-
vice a protocol provides, and from the behaviors which a protocol performs.
So this kind of complexity can normally be derived from the protocol speci-
fication. Accidental complexity refers to how the desired functionalities of a
protocol are realized, which is usually related to the protocol implementors’
decisions rather than to the specified requirements.

For protocol implementation, the division into essential and accidental com-
plicity seems to largely follow the distinction between functional requirements
and non-functional requirements [Wie03]. Functional ("system-shall-do") re-
quirements are always explicitly defined in the protocol specification and can
be mapped intuitively to an implementation. Non-functional ("system-shall-
be") requirements are of a much more subtle nature, yet they can be just as
important as functional requirements. For example, a crucial non-functional requirement for a protocol implementation in an embedded system can be that the implementation should not require more memory than the device can provide. Software quality for embedded systems involves many non-functional properties related to limited resources. New methods are researched in order to reduce power consumption or to minimize the binary size and manage dynamic memory in small device software. Typically, proposed solutions have impact on respective other properties, so there is no overall best implementation, and implementations and algorithms are chosen depending on the particular application scenario and priority.

Implementation of embedded network software should be guided by a clear distinction between the two kinds of complexity, and minimize the impact of the implementation techniques (accidental complexity). Focusing on essential complexity in the implementation means structuring the code to directly reflect the essence of the problem domain. Implementations are potentially more efficient (only including the necessary parts), and the resulting system can be both general and small. This avoids implementing any features that are not part of the protocol's core value — these features might change during implementation or because of the platform. Any extraneous feature makes the core implementation look more complicated than it should be, and in the worst case, optional features become an obstacle when porting code to another platform. In more detail, the advantages of distinguishing and separating the two kinds of complexity are:

- **Conformity:** By separating the essential part from the accidental part, implementations are closer to the specification (which only contain essential parts). Implementations should therefore use additional abstraction layers where implementation details would add unwanted accidental complexity.

- **Maintainability:** Embedded network software is expected to change often, e.g., refining some implementation details to meet performance requirements or resource constraints. These changes should be clearly limited to lower-level implementation; essential complexity should remain clear and readable.

- **Portability:** The same holds for changes to the hardware platform. Essential complexity does not depend on, and should be separate from, OS or hardware support.

- **Transparency:** Accidental complexity is introduced by implementation choices, and should be hidden inside libraries. The libraries should be implemented by specialists to ensure the quality via a uniform interface provided by OS-dependent back-ends, and accommodate the code shared by all protocols. For example, timer support for various operating systems can be used by different protocols.
We continue by analyzing common implementation practice for protocols in order to separate essential and accidental implementation complexity. For the essential parts, proven good implementation practices should be enforced and later captured by language constructs. Accidental aspects play an important role that should be separated and hidden in a lower layer of library support. The implementations for accidental aspects later constitute the runtime system of the language.

4.3 Improving Code Structure for Embedded Network Software

Implementing protocols is a challenging task, especially in embedded systems where particular platform architectures and resource constraints have to be respected. As a step towards automatically generating code from a DSL, this section illustrates common practice in protocol implementation that avoids errors and makes code platform-independent to some extent. From this, we derive a number of suggestions for how implementations can be structured to improve maintainability and provide a good basis for code generation.

4.3.1 Packet Processing

Observations According to the packet specification, protocol implementors write packet processing code that includes parsing and marshaling packets. The incoming packets are dealt with as sequences of bits in the raw buffer that have to be interpreted. The outgoing packets have to be assembled and their header fields have to be filled.

We illustrate our view using the Modbus packet (Figure 4.1) again, and show code for parsing only; marshaling is similar. The code shown in Figure 4.3 extracts contents from the raw modbus packet header by mapping it to a modbus header structure modbus_hdr declared in lines 1-5. In line 9, the address field is calculated from the packet, where it is stored in a contiguous two-element array. Please note that address is declared as an array of two unsigned char; and explicitly converted into its destination type unsigned short in big-endian style. Traditionally, network packets are big-endian encoded (network byte order), i.e. the most significant byte occupies the slot with the lowest address. Depending on the host architecture and its endianness, it can be incorrect to use unsigned short or to simply copy bit patterns.

Usually, network libraries provide helper functions to convert between network byte order and host byte order (here, ntohs and correspondingly htonl for the short int type). For instance, the code to calculate the checksum in line 24 uses an explicit conversion to network byte order to achieve the same bit pattern as the packet. This holds for both CRC and LRC checksums, a configurable option when using Modbus.
4.3. IMPROVING CODE STRUCTURE...

    /* Header fields as struct */
    struct modbus_hdr {
        unsigned char address[2];  /* unsigned, big-endian */
        unsigned char functioncode;
    };
    /* Attention to alignment! */

    /* extracting fields from packet buffer */
    #define BUF ((struct modbus_hdr *)raw_buffer_ptr)
    address = (BUF->address[0] << 8) + BUF->address[1];
    flag = (BUF->functioncode & 0x80) >> 7;
    if(flag=0) { /* nc error happens */
        functioncode = BUF->functioncode & 0x7f;
        ...
    } else { /* error processing */
        ...
    }

    /* checksum calculation */
    if(flag_CRC)
        sum = checksum_CRC(raw_buffer_ptr, upper_layer_len);
    else
        sum = checksum_CRC(raw_buffer_ptr, upper_layer_len);
    /* convert endianness and test field */
    checksum_error = htons(sum) - raw_buffer_ptr[upper_layer_len];

    Figure 4.3: Parsing code (fragments) for Modbus packets

One more complicating aspect for protocol implementations is that the host platform could have alignment constraints and thereby prohibit accessing unaligned parts of the buffer. For instance, the reserved size for modbus_hdr will usually be four bytes on a 32-bit Linux platform, but three on an 8-bit microprocessor. Even worse would be if an optimising C compiler records fields with larger types in a structure to optimise memory layout while maintaining alignment constraints (relevant in larger structure definitions). The code shown in Figure 4.3 relies on the modbus_hdr structure being laid out in contiguous memory exactly as declared, which is why its contents have to be defined as unsigned char arrays, i.e. in units of one byte.

The field extraction continues in lines 10-12 by extracting an error flag and the function code from the respective field functioncode in the packet. The error flag is the first bit in this field. Its value is extracted by applying a bitmask and a bit-shift operation to the field BUF->functioncode in the packet in line 10. Modbus specifies execution errors in a reply packet by setting the highest bit in the function code, which is otherwise identical to the function code in the request packet. As a result, the function code itself can only occupy 7 out of 8 bits — a refinement of the Modbus packet specification which becomes clear only with its protocol logic description. Depending on the respective flag
value received, the code either continues normal processing or branches into an error processing part.

Writing packet processing code is time-consuming and error-prone due to all low-level operations, using offsets, bit-masks, dedicated functions and other low-level operations. This lack of abstraction is a common source of errors. Differences in the byte order between the network device and the processor is another major source of errors and incompatibilities. As pointed out, other, even microscopic, differences between different C compilers or microprocessor architectures may lead to platform related incompatibilities. Finally, packet processing is often mixed with protocol logic implementation, which makes it difficult to trace the implementation from the specification. All of the above problems make the packet processing code hard to read by anyone other than the original authors, hard to modify in case of slight modifications in packet specifications, and hard to check and debug.

**Suggested implementation disciplines** To a large extent, the demonstrated problems are caused by byte order differences, and alignment and packet layout subtleties. To eliminate such problems, we propose to use a uniform bit-oriented interface for accessing packet contents. Each field should be described as a field handle, a structure field_t consisting of the memory location array where the packet is stored, the index bitStart indicating the bit at which packet processing should begin, and the number of bits bitLen to be consumed for this field. An API based on this type will then provide methods FieldRead/FieldWrite to read and write values for fields of arbitrary length in the memory buffer.

```c
struct field_t {
    unsigned char *array; /* pointer to storage bit buffer */
    int bitStart;          /* start point of bits in bit buffer */
    int bitLen;            /* length in bits */
};
```

```c
int FieldMake(field_t *handle, unsigned char *array,
              int bitStart, int bitLen);
int FieldRead(field_t *handle, void *bits);
int FieldWrite(field_t *handle, void *bits);
```

Packet processing code using these interfaces is easy to write and read. As an example, the code below reads 1 bit for the error flag and 7 bits for function code. Fields are processed in sequence as they appear in the packet layout. A running index variable indicates the current position in the buffer, and field structures are created and released from a local memory pool as needed while reading. No low-level bit-masks and bit-shifts are required.

```c
unsigned char flag=0;
unsigned char functionCode=0;
... ...
/* behind address field, index is currently 16 */
```
4.3. IMPROVING CODE STRUCTURE...

offset = 1;
flag_handle=
    (field_t *)malloc_from(bytepool, sizeof(field_t));
FieldMake(flag_handle, incoming_packet_buffer, index, offset);
index += offset;
FieldRead(flag_handle, &flag);
free_trace(bytepool, (unsigned char *)flag_handle, sizeof(field_t));

offset = 7;
functioncode_handle=
    (field_t *)malloc_from(bytepool, sizeof(field_t));
FieldMake(functioncode_handle, incoming_packet_buffer, index, offset);
index += offset;
FieldRead(functioncode_handle, &functioncode);
free_trace(bytepool, (unsigned char *)functioncode_handle, sizeof(field_t));

Code in this style is furthermore easy to modify. For instance, if the specification of the flag field is changed to take 2 bits instead of 1, the only required change is to replace offset = 1 by offset = 2. Code also becomes more robust and portable. Memory is not accessed through C structure mappings, so the microprocessor’s alignment constraints and compiler policies do not interfere with the parsing code. The conversion of the byte order is implemented by an underlying library which is transparent in the FieldRead and FieldWrite interfaces. With small modifications of the underlying library, e.g., header files, implementations can easily be ported to a wide range of embedded systems. The above code example for parsing two fields can be used directly both on a little-endian host architecture and on one using big-endian (i.e. the network byte order.)

With such a field access library and interface in place, packet processing code only varies in offset and index variables. It is very easy to generate the repetitive code shown above by a tool. In Paper II [WG09] attached to this thesis, we report about such a code generation tool that uses bit oriented packet specifications and automatically generates C code for field access.

Our proposal has some additional advantages particularly relevant in small embedded systems. Parsing and marshaling functions using our approach can extract and construct packet fields across byte boundaries, and quantities can be aligned arbitrarily. Packet content can be compressed, using all available bits in every transmitted byte, and thereby considerably reducing the size of the transmitted header. For example, this can be used in small wireless sensor nodes to reduce energy consumption for sending and receiving packets. On the downside, bit-oriented packing in this manner obviously implies some overhead. A few cycles are lost initializing the structure for every field, and to compute byte positions from the respective bits. Memory consumption will increase as well, but only to a minor extent, since a single field structure can be reused in an entire parsing sequence. Please note that access and conversion itself should not be considered overhead: robust platform-independent implementations have to apply the same conversions (as illustrated in the example
above), It is an advantage to encapsulate all these low-level details, bit selections and byte-order conversions, in the library. Such implementation structure will lead to much better portability and easier maintenance.

As another point, packet processing and protocol logic should not be mixed in the implementation. The main goal of our Modbus example in Figure 4.3, is to illustrate subtleties of robust packet processing code. However, it also shows another undesirable practice. After decoding the error flag from the functioncode field, control flow branches into normal and exceptional processing. This implementation style is very common and straightforward at first, but will quickly lead to problems for a more involved protocol logic. Ideally, packet processing and protocol logic should be completely separate in the implementation, to guarantee that code can be easily extended and maintained when protocol standards evolve. Another advantage of this approach is code reuse for the packet processing code. For instance, a simple packet filter for an entire protocol stack can be implemented just by composing all packet processing code and checking the extracted fields against a set of allowed values.

In order to carry out this separation, the salient properties and features of protocol logic have to be captured at a suitable abstraction level. Pure protocol logic code usually cannot be looked at as easily as packet processing code, so we derive our view of concepts and features from specifications.

4.3.2 Protocol Logic

Observations As mentioned in the previous section, procedure rules are normally formalized using finite state machine (FSM) transition diagrams. Each rule represents a transition from one protocol state to another, which includes: the current protocol state, the trigger event, conditions that must be satisfied, the new protocol state, and follow-up actions of the protocol.

The trigger events can result from external interaction with the protocol, for example by an arriving packet or when a timer fires. Other trigger events are created internally, for example, after a protocol receives a request message (from the outside), it sends out an acknowledgment packet, changing its protocol state from “receive a request” to “wait for the next request”. This internal trigger event is actually part of the follow-up action for the first event (receiving a request message). It is possible, and common, that trigger events take place asynchronously and concurrently, for example receiving packets and timeout.

Trigger events fall into three distinct categories: involving an incoming packet, involving an outgoing packet, or involving no packet. This distinction enables code sharing and semantic checks for trigger events. The same packet processing code can be potentially called simultaneously by different trigger event implementations. As an example, retransmission of a packet requires the same marshaling functionality as sending a packet. Because of the distinct categories, trigger event processing code may refer to packet header fields if the event involves a packet. Name ambiguities are resolved according to the
event category, and access to a header field when no packet is involved can be signaled as an error.

Finite state machines (be it for a protocol or otherwise) are usually implemented using a large switch statement over the machine states in C, with the code under each case being the follow-up protocol action in our case. A protocol specification can include several FSMs, which is however unusual because separate FSMs are normally collapsed into a single one when designing a protocol. One trigger event might then change states in more than one FSM. Some protocols are simple and do not use any FSM.

**Suggested implementation disciplines** Trigger events can take place concurrently, so they should be implemented as individual methods which allow access by multiple threads (or processes). FSM transitions should be grouped by trigger events at the top level. The protocol FSM state is then shared and accessible by all trigger events. It is implemented as a protocol object variable (self->state). The implementation of a trigger event covers all possible previous states, includes transition condition checking, and may also call other triggers.

Recall the FSM state diagram of Figure 4.2 for the Modbus client. Applying our implementation style to this example, sending out a packet is a trigger event corresponding to one of two possible transitions: A broadcast leads to the Waiting turn around delay state, a unicast to the Waiting for reply state. Figure 4.4 shows the respective code. The two transitions are implemented as two branches under the Idle state (case 2) in the implementation. The send function send includes packet marshaling as the final part of its implementation.

Figure 4.5 shows two other cases of trigger events. Expiry of the turnaround delay timer corresponds to the transition back to the Idle state, after the delay time. The implementation of this event, in timer1_expire_function, is simple: only one state is possible, Waiting turnaround delay (case 3). There are no condition to be checked (if (1)), and no packet processing is involved. The trigger event processing error (process_error) invokes a transition from Processing error state to Idle state. Part of the task of this function is to fill values into the packet fields and to send the packet out as an error message, so this function includes a marshaling part, identical to the one in the send function. As we see, categorizing the trigger event implementations according to the kind of packet processing they exhibit (parsing, marshaling, or none) creates opportunities for code reuse and automatic code generation.

4.3.3 Layering

**Observations** Protocols are usually designed in context of a layered architecture such as the seven-layer OSI model [Ber92]. Only protocols in adjacent
/* Trigger1: sending function */
int send(Protocol self, ...){
    ...;
    /* Implementation */
    ...;
    switch (self->state){
        case 2: //Idle
            if (modbus_serial->upward_variables.u8v0!=0){ //broadcast
                self->state = 0; //go to Waiting for Reply
                timeout(kmodbusserial_timer1, self->local_variables.intv1, &timer1_expire_function, self, ...);
            } else {
                if ((modbus_serial->upward_variables.u8v0==0)){ //unicast
                    self->state = 3; //go to Waiting turnaround delay
                    timeout(kmodbusserial_timer0, self->local_variables.intv0, &timer2_expire_function, self, ...);
                }
            }
            break;
        ...;
        /* Packet marshaling */
        ...;
    }
}

Figure 4.4: FSM code for trigger event send (send function)

layers can communicate directly by exchanging packets, e.g., sending and receiving packets. The protocol information at each layer is included in the packet, and encapsulated when passing to the next lower-layers. Higher-layer protocols thus use only the services of lower-layer protocols described by the layered architecture, without being concerned about the details of how the services are implemented. No other information is shared, so the protocols do not need any other interfaces than the send and receive functions.

In reality, protocols are violating the layered architecture for information sharing by allowing protocol information access between protocols in adjacent or nonadjacent layers, a so-called cross-layer architecture [SM05]. A higher-layer protocol can then directly use information from a lower-layer protocol at runtime. For instance, TCP can be notified about congestion errors from the lower-layer wireless link layer protocols. A lower-layer protocol can require some information from the higher-layer protocols as well. As an example, the Modbus application protocol can inform the ModbusSerial link layer protocol about its physical parameter requirements, such as baud rate and parity policy. Violation of a layered architecture leads to tight protocol implementation dependencies. Such an implementation makes assumptions and poses conditions on how other protocols are implemented. For example, we must know the names of shared variables to be able to access the cross-layer shared information. To avoid such implementation dependency between layers, a new way of
4.3. IMPROVING CODE STRUCTURE...

/* Trigger2: timer1 expire function */
int timer1_expire_function(Protocol self, ...){
    ... ...
    /* Implementation */
    ... ...
    switch (self->state){
        case 3: //Waiting turnaround delay
            if (1){
                self->state = 2; //go to Idle
            }
            break;
        ... ...
        /* Without any packet processing */
    }

    /* Trigger3: processing error function */
    int process_error(Protocol self, ...){
        ... ...
        /* Implementation */
        ... ...
        switch (self->state){
            case 4: //Processing Error
                if (1){
                    self->state = 2; //go to Idle
                }
            break;
        ... ...
        /* Packet marshaling */
        ... ...
    }

    Figure 4.5: FSM code for timeout and error processing trigger events

    interfacing and information sharing between protocol stack layers at runtime is needed.

    **Suggested implementation disciplines**  A protocol implementation should provide a standardized access interface to upper and lower layer protocols. Such an interface should include:

    - General accessing methods for sending and receiving packets.
    - Downward interfaces, i.e., shared variables in higher layer protocols that can be accessed from the lower layer protocols.
    - Upward interfaces, i.e., shared variables in lower layer protocols that can be accessed from the higher layer.

    Furthermore, for reasons explained subsequently, all protocols should share a common standard type that is a superset of every individual protocol's inter-
typedef struct protocol_block *Protocol;
struct upward_variables_block {
    unsigned char u8v0;
    unsigned short u16v0;
};
struct downward_variables_block{
    unsigned char u8v0;
    unsigned short u16v1;
    unsigned short u16v0;
};
struct protocol_block {
    struct upward_variables_block up_variables;
    struct downward_variables_block down_variables;
    MethodConcurrent send;
    MethodConcurrent receive;
    int state;
    struct local_variables_block local_variables;
    ...
    Protocol up;
    Protocol down;
};

Figure 4.6: Modbus protocol data structure

face. We illustrate the idea with a small example. Assume that Modbus application protocol has 1 unsigned char upward variable (u8v0), and 2 unsigned short downward variables (u16v1 and u16v0). ModbusSerial has 1 unsigned short upward variable (u16v0), and two downward variables: 1 unsigned char variable (u8v0) and 1 unsigned short variable (u16v0). Then, we merge both interfaces into the common data structure shown in Figure 4.6. We can see that protocol_block directly includes the two variable_blocks, so little space is wasted for variables that are not present in the respective layer (e.g. ModbusSerial requires only one upward variable instead of the two that are defined). The reason for this is memory management: using an interface like this, one can allocate a protocol block including all necessary information statically, thereby making it possible to predict dynamic memory usage at compile time. The resulting variable block size has maximal size for all protocols concerned in a particular implementation.

The presented interface methodology makes all dependencies between protocols explicit, and allows for highly modular implementations and a regular, object-oriented, construction of protocol stacks. For example, the Modbus application protocol can potentially have both ModbusSerial and ModbusTCP as its lower-layer protocols. The protocol graph is shown in Figure 4.7, and can be built up following a general method, as exemplified in Figure 4.8.

Each stack is built up from top to bottom. Stack 0 is the stack which overlays top-layer protocol ModbusApp (upper-layer protocol NULL) and lower-layer protocol ModbusSerial. For this case, build_modbus_app will call build_
modbus\_serial to create and initialize its down protocol field – a new protocol object from a pre-allocated pool. ModbusApp can now send packets through ModbusSerial by calling the self->down->send function, and access the two lower-layer variables, for example self->down->upward\_variables.u16v0. In the same way, stack 1 is built using another top-level instance and calling build with a different stack ID, which leads to a down protocol constructed by build\_modbus\_tcp. The send and receive function calls remain the same, thanks to the homogenous interface.

Notice that the programmer can make a semantic mistake by using variables that exist in one, but not in another down protocol. For example, ModbusTCP does not define variable self->down->upward\_variables.u8v0. Since every protocol has the same amount of memory reserved for its upward interface, accessing an undefined field like this will not lead to program failure, but it returns a random value. It would be possible to detect these cases using a special checking tool. However, the final purpose of fitting every protocol with a common interface is again code generation. A compiler will in no case generate code to access an undefined field because the respective source code does not have a corresponding variable bound to this field.

### 4.3.4 System Related Facilities

A protocol stack runs with kernel privilege and may use a number of highly platform dependent system services such as timers and threads. The use of these resources is a major obstacle for the portability of a protocol stack implementation. One solution is to use an interfacing subsystem located between the protocol implementation and system, which transforms different underlying system services into a uniform service interface that is exported to the protocol implementation. When the hardware platform changes, only this subsystem needs to be modified. The solution of such a subsystem is not novel; the approach is widely used in runtime systems for programming languages. For example, the Java Virtual Machine (JVM) includes this kind of subsystem for protocol implementation as part of its runtime system. In this section,
void build(Protocol p, int stackID, ...){
  ...
  switch(stackID)
  {
    case 0: build_modbus_app(p, NULL, 0);
            break;
    case 1: build_modbus_app(p, NULL, 1);
            break;
    ...
    case 5: build_otherproto(p, NULL, 5);
            default: break;
  }
  ...
void build_modbus_app(Protocol self, Protocol up, int stackID){
  ...
  self->up = up;
  ...
  /*build lower-layer protocol*/
  if(stackID==0){
    self->down = new_protocol(&protocolpcol);
    build_modbus_serial(self->down, self, stackID);
  }
  if(stackID==1){
    self->down = new_protocol(&protocolpcol);
    build_modbus_tcp(self->down, self, stackID);
  }...

Figure 4.8: Constructing a data structure for layering (example)

we discuss requirements and design of these service interfaces with a focus on timers and threads, and how system design decisions could affect them.

Timers

Observations  Timers are essential system support needed by protocol implementations to implement timeout events, i.e. actions triggered after a given time interval has elapsed. Many protocols use timeouts extensively. For instance, TCP has a retransmission timeout, Modbus has response timeout, IP has a reassembly timeout. The range of timeout could be from microseconds to tens of seconds, so in an implementation, timers are usually realized by different threads according to their granularity. A timer is often an external hardware clock to reduce the hardware interrupt penalty where a kernel timer ticks regularly as the clock. Timer implementations based on data structure timing queue and timing wheel are two common and efficient schemes. Efficiency of implementation concerns both the space required for the timer data structures, and the latency, i.e., the time between invoking a routine in the timer module and its completion.
4.3. **IMPROVING CODE STRUCTURE...**

**Suggested implementation disciplines**  Every timer implementation provides its service through an interface of roughly four routines: start timer, stop timer, set timer tick interval, and expire timer. The first two routines are invoked by protocols, the tick interval is a configuration option, and the last routine is invoked on timer ticks. For example, in Figure 4.4, timer `modbusserial_timer1` is started with interval `self->local_variables.intv1` microsecond, and its handler function is `timer1_expire_function`. The runtime subsystem should provide those four routines for protocol implementations as the uniform service interface to system timers. The subsystem is free to decide which implementation scheme is suitable for timers. Determining implementation requirements are how many timers are needed and how fine the timer granularity should be.

**Threads**

**Observations**  Real world embedded systems normally allow concurrent execution of multiple tasks. Classical multi-threading, e.g., POSIX threads, is one approach to schedule the concurrent tasks, where fully fledged OS threads and blocking primitives work together. Every concurrent task is expressed as an independent flow of control by one thread which appears linear. However, multi-threading causes memory and time overhead because each thread needs its own stack to manage its local variables and to be independently scheduled, and threads are managed by an operating system or a separate scheduling module.

Operating systems for memory-constrained embedded systems, especially tiny embedded systems, are commonly based on an event-driven model, as found in e.g., TinyOS [LMG+04] and Contiki [Dun07]. Event-driven programming does not support fully independent threads, but only a reduced model of specific event handlers. Therefore, an event-driven model does not need to allocate memory for per-thread stacks, and thus it has lower memory requirements. However, it also does not support an implicit blocking-waiting abstraction, and thus blocking actions have to be explicitly implemented. Doing this in terms of a control flow state machine by switch/case constructs in C code is in general difficult [AHT+02]. Research has been done on how to capture it in more abstract constructs and avoid the low-level programming. By offering a level of abstraction above, it is possible to write event-driven programs in a thread-like style. The event-driven Contiki system uses Protthreads [DSVM06], a lightweight stackless type of thread designed for protocol implementation with an abstraction for explicit blocking. This mechanism is implemented using pure C language constructs without any architecture specific machine code, and requires almost no additional memory. Another variant is used in TinyTimber [LEAN08]: the concurrent operations are arranged as reactive objects, and their non-blocking methods as handlers. This relieves objects from explicit blocking and protects state consistency automatically. A task dispatcher
is implemented explicitly, using the low-level C functions setjmp and longjmp, and an OS-dependent manipulation of the jump buffer to implement a context switch operation.

Protocols naturally execute concurrently. For example, a Modbus server may have several different connections to clients, and may simultaneously sends replies for received requests from these different connections at the same time. The traditional way to handle concurrent tasks in a layered protocol stack is to use fully-independent "threads" (even though these might be reduced, see above), in one of several possible schemes: one-thread-per-packet, one-thread-per-layer, one-thread-per-protocol and multiple-threads-per-protocol. Every variant has its pros and cons, so the choice depends on the particular application area. For example, one-thread-per-packet is used in x-kernel [HP91], and in Foxnet [BHL01]. Each packet is handled by one thread. When processing of one packet stalls, processing for the next thread is resumed. This model requires extensive thread synchronization. Without suitable interlocking and thread shepherding, threads can overtake each other, which leads to out-of-order message delivery. In contrast, one-thread-per-layer implementations such as Promela++ [BME98] typically induce a performance penalty from queuing packets between the layers. Bottlenecks in the protocol stack quickly lead to large time- and memory-consuming queues, avoided by the one-thread-per-packet approach. Like concurrency mechanisms, choosing a suitable thread policy is a design decision that has a close connection with the entire system design.

We can see that protocol implementations have to follow traditional concurrent schemes which are based on threads. At the same time, protocol implementations have to be integrated with the rest of the systems software and have to follow the system's concurrency programming model and central control mechanisms, which are not always based on threads in embedded systems.

Suggested implementation disciplines To make a protocol stack implementation more easily fit with different concurrency models, one solution is to implement protocols in an object-oriented style. Object orientation provides natural synchronization points at object boundaries and thus suits both threading, protothreads, and reactive objects. It is also completely in line with our previous discussion of homogeneous interfaces: wrapped in the structure protocol_block shown in Figure 4.6, we find an object with its state as data (up_variables, local_variables) and a set of methods to describe its behavior, all of type MethodConcurrent.

Concurrent operations should be implemented in thread-transparent style. This means, no concurrency constructs should appear anywhere in the code; only the underlying runtime synchronizes concurrency. Such an implementation principle matches the protocol specifications, which also completely ignore potential concurrency between different behaviors. An example is the
send function previously shown in Figure 4.4. This kind of function can be executed by one of several threads, or be scheduled as a method for a reactive object. To make this possible, methods use a first parameter self to provide access to the wrapped data, making e.g. send and receive effectively methods of a protocol object.

This structure is furthermore universal enough to support any of the usual task division schemes for protocol stacks mentioned above. The main computation part for a send function is the same, both when dividing tasks as, for example, one-thread-per-packet or one-thread-per-protocol. In the one-thread-per-packet scheme, adjacent protocols connect with each other by successive send and receive function calls. That is, the last statement of an upper layer protocol's send function is always a call to its lower layer protocol, down->send, and likewise for receive with respect to the upper layer protocol. Therefore, one packet can go up and down the stack, protocol by protocol, always within the same thread. In a one-thread-per-protocol scheme, adjacent protocols connect with each other by send and receive message buffers. Each protocol polls messages from the buffer shared with its upper protocol for processing, computes according to its functionality, and pushes the message onwards into the buffer shared with the next lower protocol.

The coding style using object orientation and common interface structures can support either scheme without any modification to the code. While reflecting precisely the functionality specified in the protocol specification (essential complexity), the implementation is independent of the particular concurrency model and task division scheme, which are handled by the common runtime system.

The complete set of strategies proposed by this chapter is: bit-oriented packet processing using a library to hide endianness and avoid alignment incompatibilities; an event-centric protocol logic implementation to achieve clean structure and readability; standardized access interfaces for protocols to automate layering and decouple it from concurrency; and in general an object-oriented programming style to support an event-based concurrency model with a reduced number of threads. Runtime system support complements these principles with a standardized interface for timer support and one of several task division schemes. In summary, we have suggested a number of implementation disciplines that all relate to one common goal: to cleanly separate parts of a protocol stack implementation into parts exposing essential and accidental complexity. This enables our actual goal: to generate code for the protocol stack implementation from a description in our language Protege. The suggestions made in this chapter have been utilized for the Protege implementation. Protege is described in detail in Paper III [WG11b].
Chapter 5

Summaries of Attached Papers

The attached papers I-IV constitute our main scientific contribution. In this chapter, we give short summaries of their contents.

5.1 Embedding a Data Description Language into Haskell

The first paper [WG08] describes how we implemented the Data Description Calculus (DDC) [FMW06] in Haskell. The result is a library for processing ad-hoc data, which generates a parser and a pretty-printer from specifications using dependent types. Based on the DDC implementation in Haskell, we later built the packet layout specification in Protege.

Despite the existence of standards like XML and JSON for information exchange, many electronic systems in scientific data processing and in everyday business use semi-structured formats that expose irregularities, commonly called \textit{ad-hoc data}. Such ad-hoc data formats often expose inner dependencies between data fields, and also evolve over time, leaving some fields unused or reusing them for different purposes, thereby creating incompatibilities between different versions. Conventional parsers and parser generators based on context free grammars are by design unable to check such contextual dependencies. Additionally, it is hard to recover from a parse error, but ad-hoc data often has to be processed even in the presence of format errors.

Therefore, ad-hoc data needs more advanced tools for specification and processing than parser generators can deliver. Domain-specific data description languages [MC00, Bac02, FG05] have been developed that compile a data description based on dependent types to a processing library in a target lan-
guage, and also generate other tools (for example, pretty-printing) from the
same source. As a generalization, Fisher et al. [FMW06] have proposed the
Data Description Calculus (DDC), which defines essential data description
combinators and a dependent type system, and formalizes properties of the
generated tools as a tri-fold semantics. Figure 5.1 shows the specification syn-
tax of DDC, which includes familiar features from compositional data types,
but can also describe layout information (for instance, a field can be recog-
nized by a field length or by a field terminator), constraints (as a predicate e
in \{ \alpha : \tau \mid e \}), and dependencies between fields (by the \Sigma-construct).

\[ \tau ::= \text{bottom} \mid \text{unit} \mid C(e) \mid \lambda x.\tau \mid \tau e \mid \{ x : \tau \mid e \} \mid \alpha \mid \mu \alpha.\tau \mid \lambda \alpha.\tau \mid \tau \tau \\
\mid \Sigma x : \tau.\tau \mid \tau + \tau \mid \tau \& \tau \mid \tau \text{ seq}(\tau, e, \tau) \mid \text{compute}(e) \mid \text{absorb}(\tau) \mid \text{scan}(\tau) \]

Figure 5.1: The syntax of DDC

DDC uses three different semantic interpretations of a type \( \tau \). First, \( [\tau]_F \)
denotes a parsing function in a suitable target language, generated from
description \( \tau \). Second, \( [\tau]_{\text{REF}} \) denotes the type in the target language, into which
successfully parsed data (agreeing with format \( \tau \)) will be parsed. Finally, a
parse descriptor \( [\tau]_{\text{FD}} \) collects meta data about the parsing action, for in-
stance error information. Especially the last interpretation is an important
and unique feature of DDC. The structured handling of parse errors allows
to process ad-hoc data that contains errors, and even to develop applications
that explicitly target error analysis.

Our contribution [WG08] is a DDC implementation as an monadic parser
library in Haskell. With our approach, both the data description \( \tau \) and the
resulting tools are Haskell programs, therefore it is a domain-specific language
embedded in Haskell. The built-in error detection and collection does not halt
the parsing process; this feature is useful for generating precise error messages
and statistics, including positions and causes, both syntactical and semantic.
Our DDC implementation provides a rich collection of parser combinators for
ad-hoc data formats, which can check physical layout, field dependencies and
value constraints, and handles failures externally by using a parse descriptor –
clearly beyond conventional parser combinator libraries.

Our embedding of DDC is straightforward, taking advantage of existing
approaches for parsing, error handling, and state encapsulation. The library
implementation uses the technique of monadic parser combinators [HM96]
combined with an explicit state monad for the parse descriptor, and an error
monad. The primary implementation unit is a parsing function (interpretation
\( [\tau]_s \)) for each DDC type \( \tau \), which computes a value of the representation type
\( [\tau]_{\text{REF}} \), and updates a parse descriptor state \( [\tau]_{\text{FD}} \) carried along while doing
so.

In the context of this thesis, the work on embedding DDC in Haskell has
laid the grounds for a concise syntax and clear semantics of packet layout
specifications in Protege. Besides, it gave us the opportunity to experiment with different DSL embedding techniques. Our implementation of DDC is entirely shallow embedding into its host language Haskell and does not involve any compilation step. As such an embedding severely restricts the use of a language, we have turned to a compilation based approach when subsequently working on the Protege language.

5.2 Generating Packet Processing Code from Data Descriptions

The second attached paper [WG09] describes a variant implementation of DDC which is more focused on packet specification and protocol overlays. The specification language is reduced to commonly needed features in packet specification, and allows for conditionally nesting packets in protocol overlays. Instead of providing a Haskell parser, C code for parsing and marshaling is generated from the packet specification.

The proposed language PADDLE provides constructs to declare fields of a packet structure. Aside from basic types, which optionally declare a storage size in bits, fields can have array or sum type, and constraints can be specified. Constraints and array sizes may depend on the values of other fields, and therefore field types are dependent types. Primitive types $B$ for field types and the syntax of expressions $e$ are taken from the target language C.

$$\tau ::= B \mid B(e) \mid \tau\{e\} \mid \tau[e] \mid \tau + \tau$$

PADDLE supports overlaying packets in a protocol stack by a combinator that declares one packet type as a field of another packet, depending on conditions. In this way, packet processing code can be generated that optimizes for the entire stack, for example, compressing the fields declared in several layers without wasting space for alignment.

PADDLE specifications are compiled to C data declarations and code for parsing and marshaling the declared packets. Packets are translated into C structures, unions represent sum types, and arrays of statically unknown length are represented as pointers. The generated parsing and marshaling functions use a bit-oriented library to access a packet buffer and convert to and from a C representation of the packet.

Performance and code size of the generated code has been evaluated using a packet sniffer application as a testbed, and were comparable to those of a hand-written packet parser. Packet specification in the data description language is considerably shorter and less error-prone by using the automatic code generation. In subsequent work for the language Protege, we have used concepts of the C code generator and (partially) the specification language, and most importantly the bit-oriented field library.
5.3 The Protege Language and its Implementation

The third attached paper [WG11b] summarizes our main contribution: we present the domain-specific embedded language Protege (Protocol implementation generator) for protocol stack implementation. In this paper, we describe Protege's general architecture and syntax, and give details on the particular embedding techniques that we have used. We also describe some features of the code generation. Prior to this paper, we have given an overview of the language in a short application paper [WG10], without going into details.

Protege code retains a close correspondence to the hierarchical and modular style of protocol specifications, to encourage code reuse and achieve good readability and maintainability. Each protocol is described in isolation and then integrated into a stack using combinators. In turn, each protocol is also described in a modular way: packet specifications are separated from protocol logic. This similarity with how protocol specifications are organized makes our language accessible to network engineers. Since packet processing code is automatically generated, protocol logic implementation is liberated from low-level data manipulation related to the wire format of packets, which substantially reduces the complexity of such implementations.

In Protege, packet specifications are an integrated part of the entire protocol specification. Packet header fields are declared for the whole protocol, with their type and size. The header field is accessible to all other parts of the protocol. In the following Protege example, the packet format for protocol p1 is specified as a sequence of two header fields (upper_p and address), followed by the payload.

```plaintext
p1 :: Protocol
p1 = protocol{ name = "P1",
    packet = packetheader<:upper_p:<:address,
    send = p1_send,
    receive = _donothing
} where upper_p = header 0 (intn bit_2) |* constraint
    constraint x = (x,==.0)|*(x,==.1)
    address = header 1 (intn bit_6)
    p1_send = do address :::* Cx0272
```

Figure 5.2: Protocol specification code

Protege uses bit-oriented packet descriptions, and constraints on the field values can be specified as predicates. In the example, upper_p has type int, id 0 and size 2 (bits). The 2 bits allow for 4 possible protocols to be located on top of p1, but the specification constrains its possible values to 0 or 1. The data description language included in Protege makes it possible to specify various dependencies among header field contents and constraints over the values of the header fields. Following the methods described previously, the
5.3. THE PROTEGE LANGUAGE AND...

Protege compiler automatically generates packet processing code from the data description.

A protocol description (like the one for p1) has to specify how to transmit its packets via its lower-layer protocols in the case of a send operation, and how to pass incoming packets to the upper-layer protocols in the case of a receive operation. In Protege, code can refer to packet fields by names in these operations, for instance, when protocol p1 adds the sender’s local address as a header field (address *:=* 0x0272).

Protege uses a set of combinators to overlay individually developed protocols into protocol stacks (graphs) in a structured manner. The basic combinator <\> overlays several protocols into a top-down stack. In the following example, a protocol stack is constructed which overlays protocol p3 above p2, and p2 above p1. Additional assumptions made about the protocols organised in the stack can be declared explicitly, using combinator <&&>. Finally, combinator <> combines two of these protocol stacks, merging duplicate instances of the same protocol (in our example, p1 at the bottom).

```
stack0 = stack 0 (protocol p3)
  <\>
  (protocol p2)
  <\>
  (protocol p1) <&&> [(P1.upper_p.==.0)]

stack1 = stack 1 (protocol p4)
  <\>
  (protocol p1) <&&> [(P1.upper_p.==.1)]

graph = stack0 <> stack1
```

Figure 5.3: Protocol stack overlay example

In the paper, we describe the implementation techniques used for embedding Protege into Haskell, and features of the Protege runtime system. Protege follows an approach of embedded compilation [EFM03], using state-of-the-art techniques, for instance, phantom types to provide type safety. Protege is embedded using a hybrid approach. The packet specification and expression sublanguage are deeply embedded (as syntax), suitable for generating C code. Protocol logic parts and overlay combinators follow a shallow embedding approach, which is easier for semantic checks and later extensions. The runtime system of Protege incorporates a number of established techniques for time and space efficient protocol implementation. System-independent libraries provide bit-oriented packet processing and relevant operating system features (especially concurrency and timer support), and we use region-based memory management to avoid garbage collection and prevent memory leaks.

Our evaluation of Protege with the Rime protocol stack [DOH07] for sensor networks indicates competitive performance with the hand-crafted C implementation, drastically reduced number of lines of code, and a minor increase
in object code size. In subsequent work, we have used Protege for a real-world industrial application scenario.

5.4 Modbus, a Case Study for Protege

Paper IV [WG11a] presents a case study for Protege. The language is used to produce a modular implementation of the Modbus [Mod06a] protocol for industrial process controllers. We show that Protege is an excellent tool to produce customized subset implementations, a commonly used technique to reduce software size and complexity in small industrial controller units.

Modbus is one of the most widely used network protocols in industrial automation applications, and is a typical example of an industrial protocol with rich functionality, relatively simple data structures, and several communication layer variants in practical use. The original ModbusSerial protocol uses legacy serial communication protocol standards (RS232 or RS485) for communication between Fieldbus-enabled equipment, e.g. micro-controllers and Programmable Logic Controllers (PLCs) within an industrial controller network. ModbusTCP is a more recent Modbus variant that offers Modbus messaging services over TCP/IP networks, to connect modern devices such as intelligent sensors or advanced PLCs to a Modbus network.

As already described in Sections 4.1 and 4.3, the definition of Modbus is divided into several specification and implementation documents [Mod06a, Mod06c, Mod06b]. The core functionality of Modbus is given as the Modbus application protocol [Mod06a], which is independent of the underlying communication layer variants and specifies a number of relatively simple functions to read and manipulate device state. Separate specifications for the communication layer [Mod06b, Mod06c] describe how the Modbus application messaging service should inter-operate with the serial line or the TCP/IP stack, respectively resulting in either ModbusSerial or ModbusTCP.

Good maintainability, modularity and code reuse are key features for quick time-to-market, and are especially attractive properties in the area of industrial protocols, characterized by long-lived standards and ongoing integration work. In our paper, we show the advantages of our compilation-based DSL approach Protege. We exemplify how to systematically decompose an industrial protocol such as Modbus, and propose a modular approach to implement a protocol by using Protege. Not only are the communication layers separated as in the specification, but our implementation also decomposes the Modbus functionality of the application layer into separate modules. As depicted in Figure 5.4, a Modbus protocol implementation can be decomposed into the two underlying communication layer variants and a number of application layer functions. These application layer functionalities can be seen as independent modules of an entire application layer, acting as small sub-protocols of their own and sharing only the common communication infrastructure below. Fur-
thermore, every protocol in the picture splits into a client and a server part, which operate on the same packet layouts as a sender and a receiver.

The compilation-based Protege approach provides the necessary setting to reflect this multi-level modularity faithfully in the code. By defining each Modbus function code as a separate (sub-)protocol of its own, functions can be freely combined into custom implementations tailored towards small controller devices with limited functionality. In addition, the compilation-based high-level approach of Protege enables code reuse for the packet processing code. Parsing (receiver) and marshaling code (sender) are generated from the same Protege source code, imported into the client and server modules. And from the separation of runtime system features for Protege, our implementation gains a large degree of platform-independence: Modbus can thus be integrated into various platforms with specialized embedded operating systems.

In total, our case study implementing Modbus demonstrates that Protege increases flexibility in several aspects and thereby considerably reduces implementation cost. New customized Modbus implementations can be produced very quickly using our approach, which makes integration and maintenance much easier and drastically reduces time-to-market – a key feature for success especially in an industrial setting.
Chapter 6

Conclusions and Future Work

We have developed the domain-specific embedded language *Protege* for network software development in embedded systems. With Protege, protocols can be programmed using domain-specific abstractions for packet specification, protocol logic, and protocol overlays, at a level of abstraction comparable to protocol specifications.

Protege helps solve the particular difficulties of protocol implementation in embedded systems described in Section 2.2. Bit-oriented packet layout, a flexible concurrency model and automatic memory management are distinctive features of the Protege runtime system to meet the resource constraints of embedded systems. Bit-oriented packet layout leads to cross-layer packet compression and thereby helps save energy (Paper III), because power consumption for sending is usually orders of magnitude higher than for computation. Thanks to its good support for modularity, Protege implementations can be easily customized (Paper IV), by decomposing complex protocols into smaller separable sub-protocols and leaving out selected modules for a reduced implementation. The Protege language constructs and runtime system create a homogeneous interface to relevant services of the operating system (Chapter 4), such as packet processing and timers. Finally, the code generated by Protege is modular standard C code which is human-readable and can be easily integrated with other system software (Paper III and Paper IV).

The first important step towards Protege was to automatically derive code for packet processing from packet format descriptions. We have developed a stand-alone tool that generates C libraries for packet processing from packet format descriptions. A packet format description is dually compiled into a packet processing library and an internal data representation that can readily be used for protocol logic implementation. More generally, we have implemented a DDC-based data description language for processing ad hoc data formats, which is a generalization of our language for describing packets. This language is entirely embedded in Haskell, as a library.
Subsequently, we have integrated this packet format description language as part of the language Protege. Protege uses packet description mechanisms and code generation strategies similar to those of the stand-alone tool, but it is implemented as a DSEL, using the technique of embedded compilation. Protege was first evaluated by implementing the Rime protocol stack, a protocol suite for small sensor nodes. The case study demonstrates that the generated C code can easily be integrated with other system software, and that the modular compilation-based approach of Protege provides good performance (competitive with hand-crafted C code) and drastically reduces source code size. As a second case study, we have implemented the industrial automation protocol Modbus, to evaluate the specific advantages of Protege in a real-world application area. Our implementation demonstrates that using the Protege language greatly facilitates protocol stack implementation, increases code sharing and reuse, and makes maintenance much easier. Our implementation has been successfully tested together with a real-world integration device, and it fully conforms to the Modbus standard.

Our implementation of Protege as a compiled DSL embedded into Haskell confirms that Haskell is a convenient host for embedding a domain specific language. Specifically, Haskell's bracket-free layout-based syntax and its higher-order functions and currying provide flexibility to experiment with several versions of language constructs, without spending much time on the parser. Other advantages are Haskell's good support for modularity and its static type system, which carries over directly to the embedded language (using phantom types). The technique of deep embedding, which we use for the packet handling sublanguage of Protege, is particularly well-suited for a compilation based approach. Other parts of Protege (protocol types and overlay specification) use shallow embedding which is more flexible for later extensions. Protege uses phantom types for its deeply embedded expression syntax, and systematically derives corresponding C types during code generation. Polymorphic Haskell helper functions can be used and, depending on the calling convention used by the programmer, will be either inlined or compiled to C functions, also allowing (limited) recursion. Embedded compilation exhibits two essential advantages over stand-alone implementations with parser and code generator: ease of implementation and increased type safety.

So far, we have used bit-oriented packet processing to allow for a compressed packet format across the abstraction barriers between layers. Other cross-layer optimizations to reduce the performance penalty for modular layering can be explored as future work, for example tail call optimizations for send and receive functions, and delayed processing.

In general, an important area of future work for Protege is to further optimize its runtime system and generated code for power and memory efficiency. The runtime system should also be extended to support a larger set of operating systems, and compiler options should be provided to control the choice between different sets of runtime primitives. So far, Protege has been coupled
with Contiki and Linux. The runtime system has its own memory and timer management, but its concurrency model relies heavily on the underlying OS. A lightweight architecture-independent kernel to provide basic concurrency should be added for cases where the target platform does not use an OS at all.

Work should continue by implementing more protocols in widespread use, in order to explore the expressiveness of Protege and guide further language improvements and extensions. We believe that the most promising area for extensions is to add more controlling features for overlay specifications. So far, overlaying in Protege focuses largely on packet handling and generates a dataflow engine. Future work could explore how the protocol logic can allow optimizations at the overlay level.

Last but not least, another area for future work is to design and develop tools which build on the Protege DSL. These could be purely supportive development tools (like visualizing protocol stacks and packet layout across layers), or more "intelligent" software for automatic testing and verification. For instance, a tool could extract finite state machines from a specified protocol stack and analyze the possible interaction sequences, including cycle and deadlock detection [BME98]. It would also be possible to simulate interaction between protocol stacks within Haskell in a visual development environment, to assist programmers in debugging protocol specifications.
Bibliography


[BHL01] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in
356, 2001. 23, 44

for constructing correct and efficient protocols. In *Proceedings of

[BRLM09] Y. Bremberg, L. Réveillère, J. Lawall, and G. Muller. Auto-
matic generation of network protocol gateways. In *Proceedings of
the 10th ACM/IFIP/USENIX International Conference on Middle-
Springer. 21, 22

based approach for network protocol message processing. *IEEE
Transactions on Software Engineering*, 2010. 22


[CHR+03] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and
C. Pu. Spidle: A DSL approach to specifying streaming applications.
In Frank Pfenning and Yannis Smaragdakis, editors, *Second ACM
Conference on Generative Programming and Component En-

[Con04] C. Consel. From a program family to a domain-specific language.
In Christian Lengauer, Don S. Batory, Charles Consel, and Martin
Odersky, editors, *Domain-Specific Program Generation*, LNCS 3016,

a new generation of protocols. In *ACM SIGCOMM Symposium on
Communications Architectures and Protocols*, pages 200–208, New
York, NY, USA, 1990. ACM. 11

[DOH07] A. Dunkels, F. Österlind, and Z. He. An adaptive communication
architecture for wireless sensor networks. In *Proceedings of the 5th
International Conference on Embedded Networked Sensor Systems
(SenSys ’07)*, pages 335–349, New York, NY, USA, 2007. ACM. 11,
51

[DSVM06] A. Dunkels, O. Schmidt, T. Voigt, and A. Muneeb. Protothreads:
Simplifying event-driven programming of memory-constrained em-
bdedded systems. In *Proceedings of the 4th International Conference
on Embedded Networked Sensor Systems (SenSys ’06)*, pages 29–42.
ACM, 2006. 43
BIBLIOGRAPHY


[Hud97] P. Hudak. Modular domain specific languages and tools. Technical report, Department of Computer Science, Yale University, 1997. 2, 13, 14, 18, 19


[LMG+04] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions
and techniques in TinyOS. In First Symposium on Networked System Design and Implementation (NSDI04), pages 1–14, San Francisco, California, USA, 2004. 10, 43


[Mod06c] Modbus Organization. Modbus over serial line, specification and implementation guide, 2006. 27, 52


[Mot05] MoteIV Corporation. Tmote brochures and data sheets, 2005. 10


Publications in the series
Örebro Studies in Technology


