Halmstad University Post-Print

A Domain Specific Approach to Network Software Architecture: Assuring Conformance between Architecture and Code

Yan Wang and Veronica Gaspes

N.B.: When citing this work, cite the original article.

©2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Wang Y, Gaspes V. A Domain Specific Approach to Network Software Architecture : Assuring Conformance Between Architecture and Code. In: 2009 Fourth International Conference on Digital Telecommunications, ICDT 2009. Piscataway, N.J.:IEEE; 2009. p. 127-132.

DOI: <http://dx.doi.org/10.1109/ICDT.2009.4>

Copyright: IEEE

Post-Print available at: Halmstad University DiVA
<http://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-96>

A Domain Specific Approach to Network Software Architecture

Assuring Conformance Between Architecture and Code

Yan Wang and Verónica Gaspes
Halmstad University
CERES
Halmstad, Sweden
yan.wang@hh.se, veronica.gaspes@hh.se

Abstract—Network software is typically organized according to a layered architecture that is well understood. However, writing correct and efficient code that conforms with the architecture still remains a problem. To overcome this problem we propose to use a domain specific language based approach. The architectural constraints are captured in a domain specific notation that can be used as a source for automatic program generation. Conformance with the architecture is thus assured by construction. Knowledge from the domain allows us to generate efficient code. In addition, this approach enforces reuse of both code and designs, one of the major concerns in software architecture. In this paper, we illustrate our approach with PADDLE, a tool that generates packet processing code from packet descriptions. To describe packets we use a domain specific language of dependent types that includes packet overlays. From the description we generate C libraries for packet processing that are easy to integrate with other parts of the code. We include an evaluation of our tool.

Categories and Subject Descriptors

D.1.2 [Programming Techniques]: Automatic Programming-Program Synthesis.

Keywords

network software, software architecture, dependent types, program generation.

I. INTRODUCTION

Network software is typically organized according to a layered architecture that is well understood. However, writing correct and efficient code that conforms with the architecture still remains a problem [1]. In other domains this problem has been dealt with using automatic code generation from specifications. The most well known case is compiler technology where lexical analyzers are generated automatically from regular expressions [2] and parsers are generated from context free grammars [3]. More recently, the same approach has gained attention in other domains like communication services [4], cryptography [5] and financial contracts [6]. We propose to use a similar approach for network software: the architectural constraints are expressed in a language that is used as a source for automatic program generation. Conformance with the architecture is thus automatically assured and knowledge of the domain allows us to generate efficient code. Further, other tools, for example for automatic testing and for evaluation of nonfunctional properties, can use our language as a source. One of the major concerns of software architecture is the

possibility of reusing not only code but also designs. With our language based approach the designs are encoded in the constructs of the language. Reuse is thus enforced *by construction*.

As part of the layered architecture, protocol specifications include packet specifications. These are written in a highly structured informal notation that describes header fields with lengths, constraints and other properties. On the other hand, programmers that implement network protocols have to deal with packets as sequences of bits that have to be interpreted according to the specification, at the same time converting between byte order in the network device and the processor. This is most frequently done in the C programming language using offsets, bit masks and dedicated functions which are difficult to relate to the specification. Also, code fragments referring to a field or to a constraint on a field can appear more or less anywhere in the code. All this makes it difficult to keep track of the correspondence between the architecture and the implementation and to make modifications to the programs that follow slight modifications in packet specifications.

In this paper we address this problem using a domain specific language based approach. We introduce PADDLE, a tool that generates packet processing code from packet descriptions made in a dedicated language. Packets are described using dependent types in a notation that also includes a construct for packet overlays. From the packet descriptions we generate C libraries for packet processing that are easy to integrate with other parts of the code. The choice of dependent types allows us to deal with semantic constraints on fields and among fields. When using our tool, packet descriptions are kept in one place, can be modified if needed, and the packet processing code is generated automatically. Both our language and its implementation as a tool are based on recent work that formalizes the treatment of ad-hoc data formats [7] that we have adapted to packet processing. Our tool is part of a larger project using the same language based approach to the development of network software.

The contributions of this work are as follows. In Section II we introduce the components of our notation for describing packets, including physical layout, dependency of fields and semantic constraints. We also show the operations for layering packets of a protocol stack. In Section III we show how we generate code for a packet processing library in C. We also discuss some of the characteristics of the generated code. In Section IV we compare the results of using our tool to generate

a packet sniffer with an existing sniffer programmed directly in C. The paper concludes with a section on related work and one on conclusions and future work.

II. PADDLE: A PACKET DESCRIPTION LANGUAGE

Our language for describing packets is a language of types that resembles structures in C: header fields are given names and types. However, it is richer than ordinary C structures. The base types can include information on the number of bits occupied by the field. The type of a field can refer to other fields, so that the value of a field can be the number of bits needed for another field. Boolean functions can be used in types to constraint the values that a field might take. To build layers of headers we provide a construct for overlaying packet descriptions. In what follows we present the language in detail.

A. Field Types

A field type is either a primitive type qualified by the amount of bits needed for its representation, a type constrained by a boolean function, an alternative between two types or an array type:

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

a) *Base Types*: Given that B is a primitive type in C and e is an integer expression in C, $B(e)$ is a field type in PADDLE. A field with this type occupies as many bits as the value of the expression. If no expression is used, the default size of the type B is assumed. The expression may refer to the value of previous fields. As an example, a field can have type $\text{int}(10)$ to indicate that only 10 bits are used in the buffer for a value of type integer for that specific field.

b) *Constrained Types*: Given that τ is a field type and e is a boolean expression in C, a field with type $\tau\{e\}$ is a field of type τ whose value satisfies the condition e . References to other fields can be made in the boolean expression. As an example, a field can be typed

```
ihl : int(4){ihl>=5}
```

The field `ihl` uses 4 bits in the buffer and its value should be an integer greater or equal than 5.

c) *Sum Types*: Given that τ and τ' are types, a field with type $\tau + \tau'$ is a field with type τ , alternatively τ' . Using this type we can describe fields of a variety of forms, as for example

```
f : char(4){f=='a'}+int(4){f==97}.
```

d) *Array Types*: Given that τ is a type and e is an integer expression in C, a field with type $\tau[e]$ is a sequence of length e of elements of type τ . The expression e may refer to other fields. For example, a field with type $\text{int}(4)[100]$ is a sequence with 100 integer elements, each occupying 4 bits. If the empty `[]` is given, the field can be a sequence of any length.

In PADDLE packets are described putting together fields in records. As an example consider the definition of a packet for IP version 4:

```
ipv4 = {
  version : u8(4){version==4};
  ihl     : u8(4){ihl>=5};
  totalen: u16;
  id      : u16;
  flags   : u8(3);
  fragoff : u16(13);
  ttl     : u8;
  protocol: u8;
  hdchksum: u16;
  srcaddr : u8[4];
  dstaddr : u8[4];
  option  : u8[ihl*4-20];
  payload : u8[totalen-ihl*4];
};
```

where `u8` and `u16` are just abbreviations for unsigned char and unsigned short.

B. Overlays

In addition to field records, PADDLE provides overlays as a way of describing packets. Overlays are used to encapsulate a packet within another and they can be nested. In this way packet specifications can be made following the layered architecture, in a modular way. Given packets *pname* and *pname'*, a new packet can be defined by placing *pname'* within one field of *pname*:

```
pname.fname <-> pname'{e1, ..., em}
```

The overlay includes a list of conditions that have to be satisfied. For example, assuming that the packet type `tcp` has been defined, the overlay that describes `tcp` over `ipv4` is

```
ipv4.payload<->tcp {ipv4.protocol==6}
```

As anticipated, the software architecture, in this case a layered architecture, is encoded in the constructions of the language. More domain specific constraints are also part of the language. In the case of PADDLE this is the fact that packet headers are records of header fields. This language based approach provides us with a source for code generation. The kind of code that is generated is also domain specific. In the case of PADDLE, we generate packet processing libraries, with fragments that can be used to parse packets, to marshal packets and to do some simple processing like filtering. These libraries can then be integrated with the rest of, for example, a protocol implementation or other network software written in C. The resulting program is improved in that there is a localized packet description, making the program more maintainable. The programmer in turn avoids dealing with the low level details involved in the implementation of packet processing. From a software architecture perspective, two central concerns are guaranteed by construction:

- conformity between architecture and code,
- reuse of code and designs.

III. CODE GENERATION

The C programs we generate from packet descriptions include in-memory representations, parsing functions and

marshaling functions. The parsing functions are used when receiving a packet while the marshaling functions are used when sending a packet.

A. The representation types

To each packet description we associate a C structure with the same fields as in the packet specification, the *representation type*. The types of the fields in the structure are obtained from PADDLE field types by erasing type dependencies and constraints. The translation from PADDLE field types to C types is as follows:

- A base type $B(e)$ is translated to the C type B .
- A constrained type $\tau\{e\}$ is translated to the translation of the underlying type τ .
- A sum type $\tau + \tau'$ is translated to a C union of the translations of τ and τ' .
- An array type $\tau[e]$ is translated to a C array with fixed size e whenever e is present and does not contain free variables. Otherwise, it is converted to a pointer.

Overlays are translated to C unions. If at some point in a program we were interested in retrieving the value of a complete packet we would get a value in its representation type. As an example consider the PADDLE packet description fragments

```
ethernet = {
  dstadd : u8[6];
  srcadd : u8[6];
  ptype  : u16;
  payload: u8[];
};

ethernet.payload<->arp
{ethernet.ptype==0x806};
ethernet.payload<->ipv4
{ethernet.ptype==0x800};
```

The representation types are

```
typedef union{
  arp *arp;
  ipv4 *ipv4;
  u8 *payload;
}ethernet_payload_u;

typedef struct{
  u8 ethernet_dstadd[6];
  u8 ethernet_srcadd[6];
  u16 ethernet_ptype;
  ethernet_payload_u *ethernet_payload;
}ethernet;
```

However, when parsing or marshaling a packet we are only interested in identifying the fields and checking that they comply with the constraints expressed in the PADDLE description. The packet will reside in some buffer and we will try to avoid copying the contents of the buffer. In order to go through the buffer we introduce a number of auxiliary types

that help in the implementation of the parsing and marshaling functions.

To begin with we introduce *field handles* that are used to keep track of the fragment of the buffer where a given field resides. Field handles have type

```
typedef struct{
  char * buffer;
  u_int index;
  u_int offset;
}field_h;
```

including a pointer to the buffer where the packet resides, an index to the buffer bit being read or written, and a count of the number of bits representing this field.

If needed, for example to test whether some condition holds, the value of a field can be easily extracted from the buffer via its handle using a predefined function `FieldRead`.

With these field handles, we can associate a type to each packet description, a *packet handler* type, that is a structure of field handles. For overlays we use unions. Translating a packet description to its packet handle type is straightforward. As an example consider Ethernet packets again. The packet handle has type:

```
typedef union{
  arp_h *arp_h;
  ipv4_h *ipv4_h;
  field_h *payload_h;
}ethernet_payload_h_u;

typedef struct{
  field_h *ethernet_dstadd_h;
  field_h *ethernet_srcadd_h;
  field_h *ethernet_ptype_h;
  ethernet_payload_h_u *ethernet_payload_h;
}ethernet_h;
```

B. The parsing function

For each packet description in PADDLE we also generate a parsing function with prototype

```
packet_h *parse_packet(char *buffer,
                       u_int bitIndex);
```

where

- `packet_h` is the packet handler type we have generated for the given packet description. For example, for Ethernet packets it will be `ethernet_h`.
- `buffer` is the memory area where the incoming packet is stored.
- `bitIndex` is an index indicating at which bit parsing should commence.

The code generated for the parsing function depends on the PADDLE type. Some illustrative cases are:

- For fields of a base type an offset has to be moved forward as many bits as required. For example, the field `ptype` of an Ethernet packet is described as `ptype:u16` in PADDLE. This is translated into

```
offset = 16;
p->ethernet_ptype_h =
    FieldMake(buffer, index, offset);
index += offset;
where the default bit length of u16 is 16 and
p->ethernet_ptype_h is the field handle initialized
by a predefined function FieldMake.
```

- For overlays, the conditions have to be evaluated and the corresponding parsing functions have to be called. For example, an Ethernet packet is an Ethernet_ARP packet or an Ethernet_IPv4 packet depending on the value of the field ethernet_ptype. The value of ethernet_ptype is read from the buffer and this guides further parsing:

```
FieldRead(p->ethernet_ptype_h,
          &ethernet_ptype);
if (ethernet_ptype==ARPTYPE) {
    arp_index =
        getIndex(p->ethernet_payload_h
                ->payload_h);
    payload_h->arp_h =
        parse_arp(buffer, arp_index);
};
if (ethernet_ptype==IPv4Type) {
    ipv4_index =
        getIndex(p->ethernet_payload_h
                ->payload_h);
    payload_h->ipv4_h =
        parse_arp(buffer, ipv4_index);
};
```

- For constrained field types, the value of the field has to be extracted and the condition has to be tested. For example, if the field version of an Ethernet_IPv4 packet has a value other than 4 it should be discarded:

```
FieldRead(p->ipv4_version_h,
          &ipv4_version);
if (!(ipv4_version==ARPTYPE))
    return NULL;
```

If something goes wrong during parsing, the function returns NULL.

C. The marshaling function

For each packet description we also generate a marshaling function with prototype

```
int *marshal_packet(
    char *buffer,
    u_int bitIndex,
    packet_inmemory *packet);
```

where

- buffer is the memory area where the outgoing packet is stored,
- bitIndex is the index indicating at which bit marshaling should commence.
- packet is the in-memory representation of the outgoing packet.

The return value of type int will be non-negative on success.

For example, the marshaling function for the Ethernet packet has the following prototype:

```
int *marshal_ethernet(
    char *buffer,
    u_int bitIndex,
    ethernet *p);
```

The marshaling function converts an in-memory representation to a sequence of bytes for transmission. It has to do the following tasks.

- Feed the value of each field to the packet buffer using a predefined function FieldWrite. For example, the following code is used to write the value of the field ptype into the buffer (assuming u16 in PADDLE):

```
offset = 16;
FieldWrite(buffer, index, offset,
           p->ethernet_ptype);
index += offset;
```

- In case of overlays, choose an adequate marshaling function to proceed. For example, ethernet_payload is written into the buffer using

```
if (p->ethernet_ptype==ARPTYPE)
    marshal_arp(buffer, index,
                p->ethernet_payload);
if (p->ethernet_ptype==IPv4Type)
    marshal_arp(buffer, index,
                p->ethernet_payload);
```

where the value of ethernet_ptype is checked first to guide the further function calls.

D. The generated code

We have put some effort in generating code that makes efficient use of resources. The parsing functions work directly on the buffer that stores the packet. Only when the value of a field is needed, for example for checking a constraint, do we use a variable in the program to store the field. The parsing and the marshaling functions are bit oriented, instead of byte oriented, meaning that fields can use less than a byte and that fields can cross byte boundaries. With this, headers can be very compact, an important issue in protocols for sensor networks.

IV. A PACKET SNIFFER

We used PADDLE to produce a TCP/IP packet sniffer, a program used to intercept and display TCP/IP packets being received over the Ethernet network. In this section we compare the resulting program with Sniffex, the TCP/IP sniffer that follows with the packet capture library libpcap [8].

Figure 1 shows the execution time results where the X-axis shows the number of packets and the Y-axis shows the execution time in seconds. The difference in performance is accounted for by the time needed to construct the handle for each field.

It is also interesting to compare the size of sources because it provides a hint on the time needed for programming and on the possibility of understanding the complete program.

Figure 1. Execution time results

	Lines of code	Generated C code size	Executable size
PADDLE	39	297	10720
Sniffex	279	-	9072

Table I
LINES OF CODE RESULTS

Table I shows the lines of code for PADDLE. The lines of code of Sniffex have been adjusted to take into count only the functions that are used in the example. The lines of C code generated from PADDLE include both .h and .c files. We are not so satisfied with the size of the executable and we think there are opportunities for optimizations.

V. RELATED WORK

We are not the first ones to be interested in using high level formal descriptions to generate programs that deal with tedious tasks. In this section we mention some of the earlier work using types as specifications for program generation.

The use of types to describe packets with the purpose of generating packet processing code was introduced in [9]. Types are used externally and a compiler generates parsing functions that can be rapidly adapted to implement packet filters. To our knowledge [9] were the first to use types for packet specifications. There is only one basic type, *bit*, and type constructors for repetition and sequencing. In order to cope with data dependency, fields are allowed to have *attributes* that can be referred to in restriction clauses. There are also ways for overlaying a packet specification in the field of another specification — typically in the payload field for layered protocols. In our work we replace the ad-hoc notion of attributes from [9] with a richer system of dependent types. We also allow for a richer set of basic types instead of just bits. In this way we can deal with more semantic constraints and consistency conditions that are beyond the scope of [9].

The idea of using dependent types for expressing constraints and physical representations was introduced in [10], in the more general setting of ad-hoc data processing. There is, however, no way of expressing overlays. More recently, in [7], the semantics of dependent types as a formalization of ad-hoc data formats was presented. In our work we adapted the semantics for the specific case of network packets.

Types have also been used in [11] to describe and manipulate binary data formats. They address however another domain, has been tested to deal with Java byte code, and does not provide mechanisms for layering data in fields.

For dealing with the contents of payloads, [12] introduces a notation for describing data types in a language independent way and comes with tools to convert these descriptions into binaries that can be used to store or transmit instances of these data structures. In our context it would typically be used to describe the kind of data an application wants to send.

VI. CONCLUSION AND FUTURE WORK

We have shown that taking a domain specific, language based approach we can address two of the main concerns of software architecture:

- conformity between architecture and code,
- reuse of code and designs.

We have illustrated this with PADDLE, a tool to assist in the implementation of protocol stacks, a domain where software is organized in layers. Using our tool packets are described in language of dependent types. In PADDLE types are used to describe both the physical layout of packets and semantic constraints on their fields. These descriptions are then the source for program generation. We generate C code for both interfacing the network to the host formats but also for parsing and writing packets from and to the wire. This processing is bit oriented allowing for very compact packet formats suitable for embedded systems. We think that the descriptions in PADDLE are closely related to packet descriptions in protocol specifications and that the resulting programs are modular and thus easy to assess correct, to maintain and to modify.

We are keen to do more extensive experiments with our tool, both to evaluate performance of the resulting code and to identify limitations that might lead to improvements. In the long term we intend to incorporate the packet types of PADDLE as part of the type system of a domain specific programming language for the implementation of protocol stacks. When for efficiency reasons packets should not be read into a data structure, the program inspects the buffer where the packet is placed by the network adapter. In making our type system internal we still want to be able to deal with the binaries storing packets. We plan to use techniques borrowed from [13] and adapt them to a language with types.

REFERENCES

- [1] M. Shaw and P. Clements, "The golden age of software architecture," *Software, IEEE*, vol. 23, no. 2, pp. 31–39, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1605176
- [2] M. E. Lesk and E. Schmidt, "Lex - a lexical analyzer generator," AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Tech. Rep.
- [3] S. C. Johnson, "Yacc: Yet another compiler-compiler," AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Tech. Rep. 32, 1975.
- [4] C. Consel and L. Réveillère, *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, ser. Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004, no. 3016, ch. A DSL Paradigm for Domains of Services: A Study of Communication Services, pp. 165 – 179.
- [5] J. Lewis, "Cryptol: specification, implementation and verification of high-grade cryptographic applications," in *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*. New York, NY, USA: ACM, 2007, pp. 41–41.

- [6] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2000, pp. 280–292.
- [7] K. Fisher, Y. Mandelbaum, and D. Walker, "The next 700 data description languages," *SIGPLAN Not.*, vol. 41, no. 1, pp. 2–15, 2006.
- [8] "Pcap," <http://www.tcpdump.org/pcap3man.html>, [Online; accessed 10-Nov-2008].
- [9] P. J. McCann and S. Chandra, "Packet types: abstract specification of network protocol messages," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 321–333, 2000.
- [10] K. Fisher and R. Gruber, "Pads: a domain-specific language for processing ad hoc data," *SIGPLAN Not.*, vol. 40, no. 6, pp. 295–304, 2005.
- [11] G. Back, "Datascript - a specification and scripting language for binary data," in *GPCE*, 2002, pp. 66–77.
- [12] CCITT, "Specification of abstract syntax notation one (ASN.1)," International Telegraph and Telephone Consultative Committee, Tech. Rep., 1988, recommendation X.208.
- [13] P. Gustafsson and K. Sagonas, "Efficient manipulation of binary data using pattern matching," *Journal of Functional Programming*, vol. 16, no. 1, pp. 35–74, 2006.