Domain Specific Languages (DSLs) for Network Protocols
(Position Paper)

Saleem Bhatti, Edwin Brady, Kevin Hammond
School of Computer Science
University of St Andrews, St Andrews, UK
{saleem,eb,kh}@cs.st-andrews.ac.uk

James McKinna
ICIS/Grondslagen
Radboud University, Nijmegen, NL
james.mckinna@cs.ru.nl

Abstract

Next generation network architectures will benefit from the many years of practical experience that have been gained in designing, using and operating network protocols. Over time, the networking community has gradually improved its understanding of networked systems in terms of architecture, design, engineering and testing. However, as protocols and networked systems become more complex, it is our contention that it will be necessary for programming techniques to evolve similarly so that they better support the design, implementation and testing of both the functional and the non-functional requirements for the network protocols that will be used.

We therefore envisage new levels of programming language support that permit: (a) the design and implementation of new protocols with provably correct construction; (b) inline testing; and (c) the expression of protocol behaviour within the design. Based on our ongoing work with both network protocols and programming language design, we believe that exploiting the capabilities of recent work in Domain Specific Languages (DSLs) will allow us to meet such requirements, allowing straightforward and "correct-by-construction" design and implementation of next generation network protocols.

1. Introduction

The next generation of network architectures must encompass a wide range of existing requirements that have emerged over the four decades or so of networking experience. Additional requirements will also appear in the near future as more applications and users populate the network. As two examples, network protocols (and applications) need to be: (i) *adaptive* to variations in network conditions; and (ii) *secure* against a range of attack vectors from the network. Furthermore, certain applications (e.g. mobile *ad hoc* networks – MANETs) may require rapid prototyping of applications protocols, coupled with strong verification of protocol behaviour and properties (e.g. military MANETs, sensor networks).

This list already presents a very challenging set of requirements. The traditional approach to protocol definition, implementation and testing typically has a long lead time, requires an extensive and difficult-to-construct test suite, and can often be difficult to debug. Overall, this means that it is not easy to experiment with novel network protocols, especially for wireless and mobile environments.

While the C sockets API remains widely used and provides excellent functionality, anyone who has used it will know that it is not easy to debug, requiring careful use and programming for ensuring correct protocol behaviour. Typically, 50% or more of the code will deal with error checking or other software control functions rather than the functionality of the protocol, and it is not easy to separate these aspects in the working protocol implementation.

Meanwhile, other languages and APIs, such as the Java network API, may offer higher levels of abstraction and better error-handling paradigms. However, they offer no specific assistance with the design, implementation and testing of network protocols, and often lack the lower-level systems hooks that give the C Sockets API its power, flexibility and continued longevity.

1.1. Limitations for protocol design

Whilst many approaches have been proposed to support the definition of communication protocols (from syntactic methods to formal methods to model-based approaches), each of them typically deals well with only a subset of the problem space. For example, Abstract Syntax Notation 1 (ASN.1) allows packet and interface definitions, but cannot specify protocol states, whereas finite-state-machines (FSMs) allow specification of state transitions, but not packet and interface specifications.

There are other requirements that are difficult to describe using such approaches. For example, in wireless and mobile environments, we may need to explore novel approaches to:

- *adaptation capability:* adaptation decisions for applications and protocol operation, e.g. use of a fuzzy systems approach to deal with changes in the network conditions [1] to allow media-stream adaptation.

- *operation in untrusted communication environments:* a node may need to support communication in environments where there is a high risk that relay nodes or end-systems may be compromised. In such a situation, trust cannot be guaranteed across the network, e.g. use of routing through secure, exploratory learning of forwarding behaviour [12].

- *tuning protocol operation for improved performance:* factors such as mobility patterns, rate of mobility and node density may affect protocol performance, so the protocol needs to be tuned for optimum performance (this is particularly important in resource limited environments), e.g. adaptation of protocol timers to reduce overhead in dynamic MANET routing [5].

To define, test and build this functionality into individual protocols using current specification techniques and technology would require experienced developers to spend significant time in the "design, build, test" cycle. Attempting to integrate all the functionality listed above in a single protocol, through a single development system, would present a very substantial undertaking. Yet these are *precisely* the kind of functions that we would like to have available in a library, in order to build new protocols, as we need them for novel applications, quickly and easily.

1.2. Scope of the paper

This paper considers a new *language-based* approach that aims to produce provably-correct protocol designs and corresponding implementations, using an easy-to-use Domain-Specific Language (DSL) mechanism. For the sake of brevity, we will confine our attention to:

- Protocols that are typically defined using mechanisms that describe the 'on-the-wire' encodings of the protocol messages, and the associated protocol behaviour.

- Protocols in use on end-systems or hosts, and providing services to higher layer protocols. That is, the protocols are likely to have both a user-plane and control-plane element and are likely to interface to lower protocols, as well as higher layer protocols. This is in contrast to, say, routing protocols, that do not really have a user-plane element.
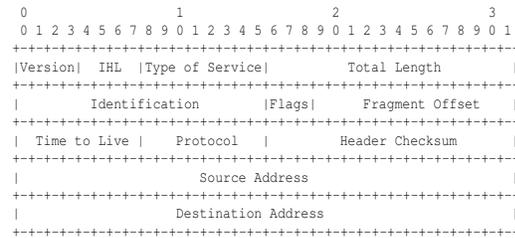


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1. IPv4 address format (from RFC791)

Our discussion and approach could equally be applied to application layer protocols; control-plane and signalling protocols; protocols used for transactional services (such as remote procedure call); protocols providing real-time streaming, and many other uses cases or application areas. However, in order to highlight the key ideas of our work, we restrict our attention to the constraints outlined above.

2. Defining protocols today

2.1. Message formats

The lower layers of message formats are still often described using 'ASCII pictures' of the byte-level, on-the-wire encoding, with bits/bytes numbered, and with conventions about bit/byte order for transmission. For example, Figure 1 shows the IPv4 packet header from RFC791. This technique has the advantage that it is easy to use, provides an easy visual understanding of the header as a data structure, and provides, effectively, a canonical view of the message. It is also machine independent, in that the message definition provides a clear notation of the bit/byte ordering of the data structure on the wire, regardless of any internal byte/word alignment and ordering preferences.

A more formal alternative is Augmented Backus-Naur Form (ABNF - Internet STD68, RFC5234, May 2008). This provides a readily machine-parseable definition but remains, essentially a syntactic notation representing the on-the-wire data structure. Another formal syntactic description notation is ASN.1, which uses abstract data types to define data structures. ASN.1 is also platform independent, and relies on the use of an associated set of formal encoding rules for the ASN.1 to define the on-the-wire encodings. The use of different encoding rules can give different on-the-wire packets for the same ASN.1 (Note that we do not consider transactional protocols, so we will not discuss RPC-like encodings here.)

2.2. Behaviour

The examples of protocol definition schemes above all have one thing in common: they are *syntactic* descriptions only. ABNF and ASN.1 both have some explicit definitions for

data-types, and therefore some semantic information, but there is nothing that describes protocol *behaviour*. Typically, protocol behaviour is described using textual descriptions which have to be translated to code by a programmer, allowing room for errors in interpreting the specification. While FSMs can be used to make the behaviour of a protocol more concrete in terms of state transitions, they still require correct translation to code by a human programmer. Systems that describe communicating processes such as Finite State Processes (FSP) and Communicating Sequential Processes (CSP) can be used to verify behaviour, but then are not related to the description of the messages. Finally, dealing with uncertainty in changing network conditions is not handled well in protocol behaviour definitions. For example, under which conditions does sufficient level of packet loss look more like a possible denial of service attack rather than the normal operation of a harsh network environment (e.g. mobile/radio)? Such concerns may affect the protocol operation and may be affected by higher-level policy. Clearly, behavioural hooks should be in place to allow such adaptive behaviour. To the best of our knowledge, no existing technique covers all the required levels of behavioural specification.

2.3. Error checking and Testing

Similarly, error checking and testing mechanisms may be distributed across the protocol definition: they may be part of the syntactic description, (e.g. numerical error code values), part of the behavioural description (e.g. states that deal with error conditions), and may also have additional behavioural input to the overall protocol operation, outside of the main syntactic and behavioural specification. Moreover, test suites are often defined outside the protocol definition, and may not always involve the protocol designer. The DSL approach described here potentially allows automatic construction of (at least some) behavioural test cases. This is a contentious point and is outside the scope of our immediate discussion: test engineers have, in the past, been totally separate teams from the development engineers, though this is changing in today's development environments and project management paradigms, e.g. eXtreme Programming (XP).

3. A DSL-based Protocol Example

3.1. Dependent Types

Where simple types express a meaning, *dependent types* allow types to be predicated on values, *so expressing a more precise meaning*. This allows the programmer both to write programs and to verify specifications within the same framework. *Simply because a program typechecks, we obtain a free theorem [16] that the program conforms to its specification*. Moreover, since a program's type is its specification, the program is, in effect, an executable proof of its specification.

Dependent types allow values to be captured within types. In a *full-spectrum* dependently typed language [10], *any* value may appear as part of a type, and types may be computed from any value. We can thus create indexed *families* of types. We require programs to be *total*, so guaranteeing that they terminate by returning a value. A key advantage of being able to index types over values is that it allows us to link *representation* with *meaning*.

A simple example concerns linked lists, which may be represented as an algebraic data type as follows, parametrised over the element type A:

$$\underline{\text{data}} \quad \text{List}\, A = \text{nil} \mid \text{cons}\, A\, (\text{List}\, A)$$

Dependent types allow us to predicate types on values, such as length, so that we may represent lists as follows, parametrised over the element type A and the list length n:

$$\underline{\text{data}} \quad \text{List} : (A : \star) \rightarrow (n : \mathbb{N}) \rightarrow \star \quad \underline{\text{where}}$$
$$\text{nil} : \text{List}\, A\, 0$$
$$\mid \quad \text{cons} : A \rightarrow \text{List}\, A\, k \rightarrow \text{List}\, A\, (k+1)$$

Note that an empty list, nil, expresses in its type that the length is zero, and a non-empty lists, cons $x\, xs$ expresses in its type that the length has increased by one over its tail. Any function over lists then carries an explicit invariant explaining the function's effect on size. For example, if we append two lists, we must add the size, and express this in the type:

$$\textbf{append} : \text{List}\, A\, n \rightarrow \text{List}\, A\, m \rightarrow \text{List}\, A\, (n+m)$$
$$\textbf{append} \quad \text{nil} \quad ys \mapsto ys$$
$$\textbf{append}\, (\text{cons}\, x\, xs)\, ys \mapsto \text{cons}\, x\, (\textbf{append}\, xs\, ys)$$

3.2. Domain Specific Languages

The ability to predicate types on values has important uses in language implementation, namely that *properties of the language we are implementing can be expressed directly in types*. Using a dependently-typed implementation language, we aim to develop a domain-specific language for communication protocol definition which integrates: i) the specification of the structure of packets and interfaces (e.g. in the style of ABNF); ii) the specification of the states and transitions in a state machine, and the conditions under which specific state transitions are valid; and iii) a means of combining and executing valid state transitions.

3.3. Protocols as DSLs

A potential benefit of dependent types is in describing data such as protocols and file formats precisely, covering *semantic* constraints in addition to more common syntactic

constraints as may be expressed in ABNF or ASN.1. We propose to take such ideas further: rather than just using types to describe protocols, we would also like to *program* in the same framework. Correctness (with respect to the stated type) can then be statically guaranteed throughout our program, and we can also exploit static information about the data to remove any need for dynamic checks, so improving efficiency. This is particularly important in implementing network protocols.

The correctness of a network protocol is often verified (if at all) by model checking a finite-state-machine or Petri Net representation [15]. This approach is, however, limited:

1. The state machine representing a protocol may have a large number of states and transitions. Verifying the protocol requires exploring the entire state space. Since this may be very large, the model may be a simplified (and so unrealistic) representation.

2. A model is needed in addition to the implementation. This model may have been simplified to reduce the state space, or there may be errors in transcription between the model and the implementation.

As an approach to verification, model-checking is therefore not *self-contained*, in that it can verify a protocol (or a simplification of a protocol), but *not the corresponding implementation*. In contrast, we will construct a *self-contained*, flexible framework in a dependently-typed language to encode and verify network protocols. This brings two main advantages:

1. Desired properties can be expressed as part of *the implementation itself*. An implementation can still be considered in terms of states and transitions, but we can ensure *at compile-time* both that only *valid* transitions can be executed (soundness), and that all valid transitions are handled (completeness).

2. Within the same framework, we can precisely express the form of data to be transmitted and *ensure* that it is validated. For example, if a message comprises some lines of text, a line count, and a checksum, we can construct a proof (a certificate) that the checksum is valid and that the line count is correct with respect to the data. Such information can also be used for optimisation; e.g. we can know statically that no bounds check is needed when looking up a bounded index from the list of lines.

3.4. A Simple Transport Protocol

We consider a simple transport protocol with automatic repeat request (ARQ), where packets consist of a sequence number, a list of bytes (the payload) and a checksum calculated from the sequence number and payload. All packets must be acknowledged by the receiver before any more packets can be sent. By using a dependently-typed host language to implement a strongly typed domain-specific language for describing protocol state transitions, we can do all of the following *in the same framework*:

1. Describe the packet format

2. Guarantee that packets are verified on receipt, and no processing occurs on unverified packets.

3. Guarantee the machine is in a valid state before executing a state transition. (e.g. timeout cannot occur if an acknowledgement has been received and acted on.)

4. Guarantee that sending a packet (or sequence of packets) ends in a consistent state, either with success or with timeout.

Packets can be described directly in our notation. They consist of a sequence number, a checksum, and the payload:

$$\underline{\text{data}} \quad \text{Packet} = \text{Pkt Byte Byte (List Byte)}$$

Data is not valid unless the checksum calculated from the data corresponds to the checksum in the packet.

$$\textbf{check} \; : \; \text{Byte} \rightarrow \text{List Byte} \rightarrow \text{Byte}$$

We represent validated data explicitly. A ChkPacket corresponds to a raw Packet, and can be computed from valid packets only:

$$\underline{\text{data}} \quad \text{ChkPacket} : \text{Packet} \rightarrow \star \quad \underline{\text{where}}$$
$$\text{chkPacket} : (seq \; : \; \text{Byte}) \rightarrow (chk \; : \; \text{Byte}) \rightarrow$$
$$(data \; : \; \text{List Byte}) \rightarrow$$
$$\text{ChkPacket} \, (\text{Pkt} \, seq \, (\textbf{check} \, seq \, data) \, data)$$

Whenever we have a ChkPacket, we have a proof that the packet data is validated — the return type of chkPacket requires that the packet on which it is predicated has a valid checksum, so the existence of a value of type ChkPacket p implies that p is valid. A secondary advantage of this (other than knowing that the data is valid) is that when a packet has been validated once, it never needs to be validated again, because the type systems ensures that we are working with validated data.

The sender can be in one of four states; ready to send, waiting for an acknowledgement, timed out, or finished. In each case, the state records the current sequence number.

$$\underline{\text{data}} \quad \text{SendSt} = \text{Ready Byte} \mid \text{Wait Byte}$$
$$\mid \text{Timeout Byte} \mid \text{Sent Byte}$$

We can describe the transitions allowed by the sending machine precisely, by relating commands to the effect they have on the machine *in the type*. Each constructor describes

how the state is affected, and carries the data required to execute the transition.

$$\underline{\text{data}} \quad \text{SendTrans} : \text{SendSt} \rightarrow \text{SendSt} \rightarrow \star \quad \underline{\text{where}}$$
$$\text{SEND} : \text{List Byte} \rightarrow$$
$$\quad \text{SendTrans}\,(\text{Ready}\,seq)\,(\text{Wait}\,seq)$$
$$| \text{ OK} : (\text{ChkPacket}\,(\text{Pkt}\,seq\,data\,(\textbf{check}\,seq\,data))) \rightarrow$$
$$\quad \text{SendTrans}\,(\text{Wait}\,seq)\,(\text{Ready}\,(seq+1))$$
$$| \text{ FAIL} : \text{SendTrans}\,(\text{Wait}\,seq)\,(\text{Ready}\,seq)$$
$$| \text{ TIMEOUT} : \text{SendTrans}\,(\text{Wait}\,seq)\,(\text{Timeout}\,seq)$$
$$| \text{ FINISH} : \text{SendTrans}\,(\text{Ready}\,seq)\,(\text{Sent}\,seq)$$

We can describe a machine parametrised by its state, which carries a list of data to be transmitted:

$$\underline{\text{data}} \quad \text{SendMachine} : \text{SendSt} \rightarrow \star \quad \underline{\text{where}}$$
$$\text{sendMachine} : \text{List}\,(\text{List Byte}) \rightarrow (s : \text{SendSt}) \rightarrow$$
$$\quad \text{SendMachine}\,s$$

Our interface to this machine is through a transition function, meaning that only valid transitions can be executed:

$$\textbf{execTrans} : \text{SendTrans}\,s\,s' \rightarrow$$
$$\quad \text{Machine}\,s \rightarrow \text{IO}\,(\text{Machine}\,s')$$

In effect, we have created a DSL (SendTrans), with the validity operations enforced by the dependent type system of the host language. The *interpreter* for the DSL, **execTrans**, guarantees through its type that the properties required of the protocol are satisfied *however it is implemented*. The return type, $\text{IO}\,(\text{Machine}\,s')$ indicates that the state machine has moved from state s to s', and executed some I/O operations in the process (possibly involving the actual transfer of data across a network connection).

So far, we have described and hopefully justified the transitions allowed in the implementation of the sender. Types ensure that these transitions are applied safely and correctly in the implementation. In our implementation, the process of sending a packet could result in the machine either being ready to send the next packet, or timed out:

$$\underline{\text{data}} \quad \text{NextSent} : \text{Byte} \rightarrow \star \quad \underline{\text{where}}$$
$$\text{NextReady} : (seq : \text{Byte}) \rightarrow$$
$$\quad \text{SendMachine}\,(\text{ReadyToSend}\,(seq+1)) \rightarrow$$
$$\quad \text{NextReady}\,seq$$
$$| \text{ Failure} : (seq : \text{Byte}) \rightarrow$$
$$\quad \text{SendMachine}\,(\text{Timeout}\,seq) \rightarrow$$
$$\quad \text{NextReady}\,seq$$

The **sendPacket** function, which implements the protocol by sending a packet and waiting for an acknowledgement, can now be given a type which ensures that the machine is ready to send a packet at the start, and expresses that it will end in an appropriate state (e.g., not still waiting for a reply):

$$\textbf{sendPacket} : (seq : \text{Byte}) \rightarrow \text{List Byte} \rightarrow$$
$$\quad \text{SendMachine}\,(\text{ReadyToSend}\,seq) \rightarrow$$
$$\quad \text{IO}\,(\text{NextSent}\,seq)$$

Any type-correct implementation of **sendPacket** has an explicit guarantee (verified by the type checker) that it ends in a consistent state. *Either the packet has been successfully sent, or the request timed out and the machine is ready to try again.*

The receiver is constructed in a similar way, but the states are simpler because there are no acknowledgements to wait for — the receiver will either accept a packet and wait for the next in sequence, or else will reject a packet.

$$\underline{\text{data}} \quad \text{RecvTrans} : \text{RecvSt} \rightarrow \text{RecvSt} \rightarrow \star \quad \underline{\text{where}}$$
$$\text{RECV} : (seq : \text{Byte}) \rightarrow (data : \text{List Byte}) \rightarrow$$
$$\quad \text{CheckPacket}\,(\text{Pkt}\,seq\,(\textbf{check}\,seq\,data)\,data) \rightarrow$$
$$\quad \text{RecvTrans}\,(\text{ReadyFor}\,seq)\,(\text{ReadyFor}\,(seq+1))$$

4. Related work

The most closely related work to that described here has been published either in the DSL community or the network community, but not both. As far as we are aware, there is no current work that addresses network programming and protocol definition in the manner we propose. In particular, none of the existing approaches deals explicitly with uncertainty in operation, e.g. with respect to security and adaptability in network protocols as explained above.

4.1. Domain specific language approaches

The work that is most similar to our own involves developing special-purpose type systems for guaranteeing resource properties, e.g. [6, 7, 13, 14, 17]. These approaches differ from ours in that they develop a new resource type system rather than exploiting an existing general-purpose type system. They consequently must also develop both new soundness proofs and a new type checking implementation. While these approaches seem promising as specialised applications, we prefer to build on a strong general-purpose type system, to give maximum flexibility, and to allow maximum reuse of existing tools, proofs and implementations. Data and packet description languages, e.g. [3,8,11] are also closely related to our work, but unlike our approach are limited to expressing syntactic constraints.

4.2. Model-checking approaches

Previous approaches to protocol verification have typically been based on developing special-purpose type systems, *post-hoc* program analysis, or a combination of these techniques. For example, Marriott *et al.* [9] use a deterministic finite-state automaton (DFA) to describe the allowed states of resources. Their approach relies on a program analysis that models the *approximate* behaviour of the program, and that then checks that this behaviour conforms to

the DFA. In contrast, in our approach, we effectively place the permissible states in the types of the domain-specific language. This allows us to relate the *real* program, rather than an approximate model, to the permitted behaviour and so to guarantee correctness *by construction* and without the limitations of a DFA. In particular, unlike a DFA, we are not limited to a pre-determined number of states.

4.3. Protocol verification approaches

We are also aware of some other relevant approaches, that, unlike our work, *separate* the definition, checking and implementation of the protocols. For example, Metarouting [4] proposes an algebra for checking correctness of *routing algorithms*, with a tool to generate `OCaml` code; there is also work in the use of high-order-logic proofs [2] for (*post hoc* and *pre hoc*) semantic analysis of protocols; and the Zebu project [1] at INRIA also focuses on some aspects of correctness in specification and code generation. Finally, work related to PRISM [2] considers probabilistic modelling, including some work on communication protocols (though this is not its main focus), but constrains the problem-space to specific Markov processes.

5. Summary

In this paper, we have taken the position that the use of domain specific languages (DSLs) based on dependent types can offer major benefits in protocol design, implementation and testing. By allowing semantic and behavioural information to be defined together within a DSL, we allow the definition of a protocol to be provably "correct by construction". We do not require the use of multiple, disparate formal methods to define the message format and behaviour: if the type checking is correct, then it follows that the definition *must be correct*. Furthermore, if an implementation is created from the DSL, then it must operate correctly, simply by the properties obtained from use of dependent type systems. We have demonstrated the efficacy of our position by giving an example of a simple automatic repeat request (ARQ) protocol: we give the specification, the behaviour and evidence of its correct operation from its definition.

5.1. Further work

This is still very much a work in progress. Given our enthusiastic and bold claims, there can be only one priority for further work: to build a system showing the use of DSLs and associated programmatic tools that will demonstrate the position we have taken in this paper. We intend to do this in the very near future.

---

[1] http://phoenix.labri.fr/software/zebu/
[2] http://www.prismmodelchecker.org/

References

[1] S. N. Bhatti and G. Knight. Enabling QoS adaptation decisions for Internet applications. *Computer Networks*, 31(67):669–692, 1999.

[2] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as Applied to TCP, UDP, and Sockets. In *SIGCOMM '05: Proc. Conf. Applications, technologies, architectures, and protocols for computer communications*. ACM, 2005.

[3] K. Fisher, Y. Mandelbaum, and D. Walker. The Next 700 Data Description Languages. In *Symposium Principles of Programming Languages*. ACM, 2006.

[4] T. G. Griffin and J. L. Sobrinho. Metarouting. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2005.

[5] Y. C. Huang, S. Bhatti, and D. Parker. Tuning OLSR. In *PIMRC2006 - The 17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Helsinki, FI, 2006.

[6] A. Igarashi and N. Kobayashi. Resource Usage Analysis. In *Symposium on Principles of Programming Languages*, pages 331–342. ACM, 2002.

[7] N. Kobayashi. A Type System for Lock-Free Processess. *Inf. Comput.*, 177(2):122–159, 2002.

[8] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML, a Functional Data Description Language. In *Symposium on Principles of Programming Languages*. ACM, 2007.

[9] K. Marriott, P. Stuckey, and M. Sulzmann. Resource Usage Verification. In *Proc. First Asian Programming Language Symposium, APLAS 2003*, pages 212–229. Springer-Verlag, 2003.

[10] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[11] P. J. McCann and S. Chandra. Packet types: Abstract specification of network protocol messages. In *SIGCOMM '00*. ACM, 2000.

[12] M. Rogers and S. Bhatti. A Lightweight Mechanism for Dependable Communication in Untrusted Networks. In *Proc. DSN2007 – 37th Annual IEEE/IFIP Conference of Dependable Systems and Networks*. IEEE Computer Society, 2007.

[13] K. Suenaga and N. Kobayashi. Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts. In *ESOP*, volume 4421 of *LNCS*. Springer, 2007.

[14] G. Tan, X. Ou, and D. Walker. Resource Usage Analysis Via Scoped Methods. In *Foundations of Object-Oriented Languages*, 2003.

[15] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, fourth edition, 2003.

[16] P. Wadler. Theorems for Free! In *Proc. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89*. ACM, 1989.

[17] D. Walker. A Type System for Expressive Security Policies. In *Twenty-Seventh ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 254–267. ACM, 2000.