The Agilent Protocol Encoder (APE)

Kevin Mitchell, *Member, ACM*

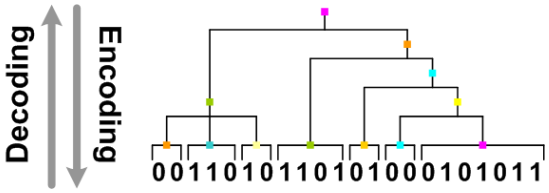
***Abstract*—The Agilent Protocol Encoder (APE) was designed as a test-bench to explore various aspects of protocol specification. In addition to designing a domain-specific language for specifying communication protocols, the project built an integrated development environment (IDE) to assist in the construction and debugging of such specifications. The concept of action-directed decoding was developed to allow a range of different decoders to be constructed from the *same* specification. The compiler can generate code for the APE virtual machine (APE VM), with an instruction set tailored to the task of protocol decoding. A firmware implementation of this VM has also been developed, along with a more traditional back-end that generates C++ code.**

***Index Terms*—protocol decoding, specification languages.**

I. Introduction

THE APE framework, developed within Agilent Technology's research labs, consists of a new language for specifying communications protocols, a tool chain to support the manipulation of specifications written in this language, and translators into high-quality decoders, both in software and firmware. This paper describes the APE language, the motivation behind its design, the principle achievements of the project, and the areas that have been identified for future exploration.

Protocol messages are typically structured in a hierarchical fashion, but the communication links on which they are transmitted are linear in nature. The protocol specification defines how such messages are encoded into a linear stream, and decoded back into the original representation by the receiver.



Some protocols, such as SMTP[1] and SIP[2], are text-based. They are distinguished by being human-readable, and in many cases free-format in the sense that white-space is not significant. The task of decoding such protocols, at least at the syntactic level, is very similar to that of parsing a conventional programming language. Existing compilation tools, such as YACC[3], Bison[4] and ANTLR[5], can be readily adapted to the decoding task. Other protocols are XML-based, with SOAP[6] being a good example of this style. Many tools exist for manipulating XML documents, and these can be used for manipulating XML-based protocols. Where space is important, for example, when transmitting data across a link where you wish to conserve bandwidth, a binary protocol is usually more appropriate. Such protocols attempt to compress the data into a small number of bits, and are usually not human-readable without mechanical assistance. Bit-level protocols are widespread, ranging from the various IP stacks used in the Internet to the commonly used encodings for ASN.1-based protocols. The APE is intended to address the problem of describing, encoding and decoding bit-level protocols.

K. Mitchell is with the Measurement Research Laboratory, Agilent Technologies UK Limited, Edinburgh EH12 9DJ, UK. Email: (see http://member.acm.org/∼kevin_mitchell).

II. Background

In the beginning protocol decoders were typically written by hand. Conceptually this is quite a simple task, but it is time-consuming and error-prone. Over time people developed support libraries to assist in this task. The structure of the code naturally mimicked the structure of the protocol being decoded, and it quickly became apparent that this task could be mechanized if the protocols were specified in a machine-readable fashion. Over the years a number of such tools have been developed, so why introduce yet another language? The majority of the previous examples have been procedural in nature. They describe *how* to decode a protocol, rather than simply describing the abstract structure of the protocol's data units. This tends to lead to overly verbose specifications where the user is forced to specify details that could be inferred by a compiler. Furthermore, generating encoders from such specifications can also be problematic. One of the initial goals of the APE project was to develop a more declarative way of specifying protocols. This should enable the same specification to be used for both encoding and decoding. By constructing a domain-specific language for the task, coupled with a compiler incorporating techniques like type inference and generic dispatch, the resulting specifications should be more concise and flexible than their procedural counterparts.

Writing the protocol specification is only part of the story. Most people develop programs using interactive development environments. These IDEs provide a number of advantages over traditional command-line tools. These advantages include on-the-fly parsing to detect syntax errors, language-aware editors that perform code-folding and syntax highlighting, powerful navigator windows to quickly navigate around the different declarations within a file, and between files, language-specific wizards to automatically generate code, integrated debuggers, with source-level debugging, and so on. Unfortunately, when developing protocol specifications many of the existing tools require the users to go back to the old command-line style of program development. All the advantages of using an IDE are equally applicable to the task of protocol development; we just need an IDE that supports this task.

Traditional tools typically adopt a one-size-fits-all approach to decoding. Their decoding actions are largely independent of the intended use of the decoder. But decoding a stream of Ethernet packets to determine the mix of protocols encapsulated within these packets potentially involves a lot less effort than decoding the same packet stream to display in something like a WireShark browser[7]. Another goal of the APE project was to develop techniques to allow the decoders produced by the compiler to be tailored to the task at hand.

III. Language

In this paper we just give the reader a flavor of the APE language that should be sufficient to understand the main ideas behind the design.

A. PacketTypes

The original inspiration for the APE language was the work done on PacketTypes by McCann and Chandra at Bell Labs in 1999-2000[8]. Their language was both expressive and concise, although this conciseness often led to ambiguities within the specification. The language had the following salient features:

- Packet descriptions are expressed as types
- The fundamental operation on packets is checking their membership in a type.
- Layering of protocols is expressed as successive specialization on types.
- Refinement of types creates new types, a facility useful for packet classification.

The starting point for the PacketTypes work was the C programming language. The hand-written decoders that this work intended to replace were written in C, and so the typical protocol implementor was already familiar with this language. Building on the notation for C structures was therefore a natural first step. However, the type system of a language such as C is not expressive enough to capture all of the structural dependencies in a typical bit-level PDU. For example, in the case of an IP PDU we might start by defining

```
nybble := bit[4];
short := bit[16];
long := bit[32];

IP_PDU := {
  nybble version;
  nybble ihl;
  byte tos;
  short totallength;
  ...
  byte protocol;
  short cksum;
  long src;
  long dest;
  ip_option options[];
  bytestring payload;
} ...
```

This type can be interpreted as imposing a structure on packets, but without additional constraints it allows many sequences of bits that are not valid IP packets. The necessary constraints appear in a **where** clause following the sequence, as in:

```
IP_PDU := {
  ...
} where {
  version#value = 0x04;
  options#numbytes = ihl#value * 4 - 20;
  payload#numbytes =
    totallength#value - ihl#value * 4;
}
```

The **overlay** ... **with** refinement constraint allows us to merge two type specifications by embedding one within the other, as is done when one protocol is encapsulated within another, lower-layer protocol. For example, the specification of the encapsulation of a User Datagram Protocol (UDP) frame within an IP packet may be specified as:

```
UDPinIP :> IP_PDU where {
  protocol#value = 17;
  overlay payload with UDP_PDU;
}
```

assuming that the UDP_PDU type was appropriately specified elsewhere.

The use of constraints leads to natural and concise specifications in many cases. However, the language is also very ambiguous. One of our initial goals was to remedy some of the deficiencies of the language, whilst retaining the flavour of this approach.

B. APE Packet Types

The philosophy underpinning the design of the APE language is similar to that introducted in PacketTypes, although the syntax is less influenced by the C language. So in the APE language the previous example would be written as follows.

```
IPv4 =
( header:   (
    version:      UInt4<4>
    ihl:          UInt4
    tos:          Type_Of_Service
    totallength:  UInt16
    ...
    protocol:     UInt8 as IP_Protocol
    cksum:        Bit[16]
    src:          Bit[32]
    dest:         Bit[32]
    options:      IP_Option[]
  ) where num32 == ihl
  payload: IP_Payload<header.protocol>
) where numBytes == header.totallength;
```

The details of this definition are not important at this point; we are simply trying to give the reader a flavor of the language. Just as in PacketTypes, protocols are described by types where the structure of each type mimics the structure of the PDU it is describing, and where constraints are used to resolve ambiguities. The simplest packet type is Bit which matches a single bit in the PDU. Compound types can be constructed using sequential composition (indicated by juxtaposition), alternation, and iteration. In this respect the syntax has a lot in common with regular expressions. But types can be named, and referenced from other types, allowing recursive types to be constructed. Subcomponents can also be named, and then referred to elsewhere within the type.

C. Constraints

Each type reference has a number of attributes associated with it. Given a type reference T the expression T#*attr* denotes the value of the *attr* attribute of this reference. Type references are matched against bit sequences, and so the most frequently used attribute is size which indicates the number of bits that matched, or should match, against the reference. A constraint can restrict the value of this attribute to a range of values, most typically a single integer, or can use the value of this attribute to constrain other parts of the specification. In our example the (user-defined) expression numBytes is interpreted as an abbreviation for numBytes(**this**) which, in turn, is interpreted as idiv(**this**#size, 8). When used in the expression numBytes == header.totallength this has the effect of constraining the type to which the constraint is attached to have a size which is an integral number of bytes, whose length (in bytes) is equal to the value of the totallength field. This example also introduces another attribute, value. By default, the value attribute for a type reference is the bit sequence against which the reference was matched. However, the user can also specify an explicit value, as in

```
UInt8 = Bit[8] where value is unsigned;
```

Such a type matches eight bits, but the value of a reference to such a type is the unsigned integer represented by these bits. Other constraints include

address	the bit address of the start of the type reference in the PDU.
count	the number of iterations in an iteration type reference
choice	the index of the alternation chosen in an alternation type
remaining	the number of bits remaining in the current context at the start of the type reference.

One way of viewing these constraints and attributes is as an attribute grammar[9]. We attempt to find a derivation tree from the grammar such that the fringe of the tree matches the bit sequence constituting the PDU being matched, the attributes of all nodes are defined by their positions within the tree, and where all the constraints over these attributes are satisfied. Whilst such a view encapsulates what it means for a bit sequence to be accepted by such a specification, it doesn't help us find valid derivation trees in an efficient fashion. The challenge, from a compilation viewpoint, is to find a **deterministic** decoding strategy that will accept all bit sequences that satisfy the specification, and reject all those that don't.

D. Generic Types

Consider trying to specify the IPv4 protocol. The *type* of the payload field clearly depends on the *value* of the protocol field. But this mapping from value to type has to be extensible to allow new protocols to be added later. Ideally we should not have to alter the file containing the definition of the IPv4 protocol just because we want to handle a new IPv4-encapsulated protocol.

The approach adopted by the PacketTypes language was to use overlays. In our view that has a couple of disadvantages. First, in the original definition of the IP_PDU type there is no indication that the protocol and payload fields are linked in any way. It is only when the reader encounters the UDPinIP type that this linkage becomes apparent. In addition, the overlay mechanism can be viewed as decoding a bit sequence twice, the first time as a simple bit sequence, and then subsequently as a UDP frame, or whatever type is imposed by the overlay. As we will discover later, the APE allows user-defined actions to be embedded within our packet types. If we allowed some form of overlay mechanism it would significantly complicate the definition of what actions get executed during a decode, and in what order.

In the design of the APE we decided to take our inspiration from the design of the multimethod-dispatch mechanism in languages such as Dylan[10] and Cecil[11]. We believe our approach makes the relationship between value and type clear, and naturally supports extensible mappings, both at linking and runtime. We start by defining

```
generic IP_Payload<protocol: Int>;
```

This declares IP_Payload as a generic type, i.e. a type whose definition depends on the value of the argument, an integer in this case. We then define

```
IPv4 =
( header:
  ( ...
    protocol:       UInt8
    ...
  ) where ...
  payload: IP_Payload<header.protocol>
) where ...;
```

In this specification the structure of the payload field is constrained by the value of the protocol field. However, the mapping from integers to payload types is still not specified at this point. To define this mapping the language allows us to declare specializations of a generic type. These specializations constrain the argument(s) of the generic to subtypes of the original argument type(s), and then provide a definition of the type to associate with these arguments. We view primitive types such as Int as sets of values, and then treat subtypes as subsets, and constants as singleton sets. With this interpretation we can now define

```
IP_Payload< 6> = TCP;
IP_Payload<17> = UDP;
```

and so on. Here we view 6 as an abbreviation for the set $\{6\}$ which is a subset, and hence subtype, of the set/type Int. Although fairly meaningless in this particular example, we could also define specializations such as

```
IP_Payload<1000..2000> = T;
```

This would be used whenever the protocol value was in the range 1000..2000 and there were no other specializations that were more specific.

E. Most-specific specializers

Clearly for such an approach to be deterministic our specifications must be written in a way that for every possible argument to the generic type there must be a unique most-specific specialization. Therefore if we defined

```
IP_Payload<30..50> = T1;
IP_Payload<40..60> = T2;
```

then we would have an ambiguity unless there was also a specialization such as

```
IP_Payload<40..50> = T3;
```

to deal with the overlap between the two ranges.

If the protocol field had the value 41 then we would view the payload as having type T3. Why? Because 40.50 is a subtype, when types are viewed as sets, of both 30..50 and 40..60, and so is more specific than either of them. In the most common case where the specializations are singleton values then it is easy to ensure unique most-specific specializers. But in more complex cases, involving ranges and multiple arguments, it can be more challenging to ensure uniqueness.

The APE language supports a simple module structure. Types are defined within the context of a module. An import declaration allows modules to be imported into other modules, and the corresponding export declaration allows selected types to be visible outside the module. The development environment compiles each module independently. However, a decoder is generated for a project, which contains a set of modules. The specializations for each generic type can be spread across multiple modules within the project. The dispatching code required to support each generic type can therefore not be generated on a per-module basis. The compiler invokes a linker phase after all the modules in a project have been compiled. The linker is responsible for analyzing all the available specializations for a generic, determining the most-specific specializations, and generating dispatching code for the generic type.

There is one further complication that should be noted at this point. In some cases the mapping from values to types may need to be modified at run-time. A good example is where dynamic port-mapping is used. An exchange of signaling messages may establish a relationship between an IP port and a particular protocol. To support such dynamic behavior the generic dispatch mechanism needs to allow the runtime system to alter its behavior. A simple strategy is to attach a map from values to types (or pointers to their runtime representations) to the dispatching code. The map is checked for entries first, and if no entry is found then the statically-compiled dispatching code is then executed as before. To avoid this overhead when there is no need to support dynamic modifications the generic type must be tagged with a **dynamic** keyword modifier when such behavior is required.

F. Extensible Enumerations

In our previous example the protocol field contained a value in the range 0..255, and the generic type IP_Payload also took an integer as argument. This is perhaps not as clear as we would like. A neater approach might be to define an enumeration type that allowed us to use symbolic names instead of constants such as 6 and 17 in our specifications. The protocol field could then be defined to provide values of this type, and the generic could be specialized on this type.

The basic idea of the enumeration mechanism is similar to the enumerations in languages such as C++ and Java, i.e. it defines a type/set with named values. However, where it differs is that in some cases we want the set of named values to be extensible. So, for example, in the IPv4 module we define an IP_Protocol enumeration. We could, at that point, specify all the possible values of this enumeration. But this would embed into this module some information about lots of other protocols, i.e. their IP protocol number. Now in some cases this might be want you want. For example, it is natural to define all values for the IP service precedence field within the IPv4 module, in which case you could define the type as[1]

```
datatype IP_Precedence = {
  routine = 0,
  priority = 1,
  immediate = 2,
  flash = 3,
  flash_override = 4,
  critic_ecp = 5,
  internetwork_control  = 6,
  network_control = 7
};

Type_Of_Service = (
  precedence: UInt<3> as IP_Precedence
  delay:      ...
  ...
);
```

This approach works well when all the elements of the enumeration can be specified at the point where the type is first introduced. However, in the case of the protocol field an alternative approach might be to define the type within the IPv4 module, but allow enumeration bindings to be added to it in other modules. We refer to such declarations as *extensible enumerations*. We need a mechanism for defining which enumerations are extensible, some way of constraining the range of permitted values, and a syntax for adding new bindings to the enumeration. We deal with each of these in turn.

To indicate that an enumeration is extensible we simply attach an ellipsis to the end of the definition, as in

```
datatype IP_Protocol = { ip = 4, ... };
```

In most cases we wish to constrain the permitted values of the enumeration, and we can use a constraint to perform this task.

```
datatype IP_Protocol = {...} where in 0..255;
```

To add an additional binding to the enumeration we use the following syntax.

```
datatype IP_Protocol += { tcp = 6 };
```

Note that this definition can appear at the point where the TCP protocol is defined, for example in a TCP module. It does not

[1] APE types form two distinct classes. Packet types are used to define the structure of protocol elements. The attribute values range over a disjoint class of data types, including `Int`, `Bool`, and user-defined enumerations.

have to be defined in the IPv4 module. When the compiler builds a decoder it collects together all the modules in the current project. Depending on the mix of modules chosen the set of bindings for each enumeration can potentially vary.

Now whether such an approach is preferable to the non-extensible version is partly a matter of taste. Does the knowledge that TCP is mapped to value 6 in the IPv4 protocol field "belong" to the IPv4 module, to the TCP module, or to an auxiliary TCP_over_IP module, for example? If a new protocol comes along that can also be carried over IP, do we need to go back in and change the IPv4 module, or should we just be able to add ourselves to this enumeration from the new module? We need to gain more experience of writing APE specifications before coming to any firm conclusions about which approach works best in practice.

G. Higher-order types

Most APE types either take no parameters, or are parameterized on abstract types such as integers, Booleans or bit sequences. There are occasions, however, when we require a bit more flexibility. Consider the following examples:

```
TA = num: UInt8
     values: A[num];

TB = num: UInt8
     values: B[num];

TC = num: UInt8
     values: C[num];
```

There is clearly a common pattern being repeated here. It might be clearer to define a single type parameterized on the type being iterated over. We can write such a definition using a higher-order type, i.e. a type that takes another type as parameter.

```
NV<T> = num: UInt8
        values: T[num];

TA = NV<A>;
TB = NV<B>;
TC = NV<C>;
```

What if the type being passed as argument is also parameterized? The TLV type in the WiMAX specification[12] illustrates this need, and the APE syntax required to specify it.

```
WiMaxTLV< WiMaxTLV_Value<t:Int> > = (
  type: UInt8
  len:  WiMaxTLV_Length
  val:  WiMaxTLV_Value<type>
        where size == (len * 8)
);
```

We can parameterize the WiMaxTLV type on any type that takes an integer as argument.

H. Coercions

In the APE we can coerce between different types using the e **as** T syntax, for example to explicitly coerce from an Int value to an enumeration type. Normally the type T is a datatype. But we can also make sense of such a coercion in the case where e is a value of type Bits, and T is a packet type. Indeed you can imagine the whole decoding process as starting with the execution of the expression e **as** T, where e is the bit sequence representing the PDU and T is the root packet type to be used to decode the PDU. The interesting thing about this construct is that there is nothing stopping you evaluating such an expression in the middle of another decode. Of course the challenge is then to define what such an expression does. But let's delay worrying about that for now, and just consider how we might use such an expression in a specification.

Suppose we have a protocol consisting of a collection of blocks, where each block has some data and then two "pointers" to other blocks. We assume the blocks can be in an arbitrary order, possibly with gaps between them, and with no distinguishing bit sequence at the start of each block. Furthermore, we assume that the first block is at the start of the PDU. Decoding a packet containing an encapsulated TIFF file[13] might require us to handle a layout similar to this one. How can we express such a protocol in the APE?

We define a new module, plus either import or define some basic types such as Byte and UInt8. We then add the following definitions to the module, where the type PDU is the root type.

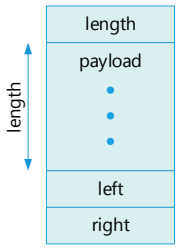
```
Pointer<bs: Bits> =
  (ptr: UInt8) where value = bs as
       T_at_offset<ptr, bs>;

T<bs: Bits> =
  length  : UInt8
  payload : Byte[length]
  left    : Pointer<bs>
  right   : Pointer<bs>
;

T_at_offset<offset: Int, bs: Bits> =
  prefix: Byte[offset]
  t: T<bs>
  suffix: Byte[]
;

PDU = (bs: Bit[])
      where value = bs as T_at_offset<0, bs>
```

What is going on here? We start by decoding the PDU as a sequence of bits. This decode will obviously be very quick as we aren't trying to impose any structure on the bit sequence. It will essentially just skip over all the bits in the PDU in a single leap without processing them in any way. However, it also allows us to assign the name bs to the PDU bit sequence. We then evaluate the expression bs **as** T_at_offset<0, bs>, which will result in the decode of this bit sequence against the type T_at_offset<0, bs> at some later point.

The type T_at_offset is intended to capture the structure of an instance of type T at some specified byte offset in the PDU. It imposes the following structure on a bit sequence.

If the offset is 0, as it is in our case, the prefix will be empty, the type T will be matched against the start of the bit

sequence, and the suffix will soak up any bits that remain after T has been decoded. The type T has the following structure:



The field left is of type Pointer. It matches against a byte, interprets this as an unsigned integer, and also evaluates an expression that will result in the decode of the original bit sequence against the type T_at_offset<ptr, bs>. In other words we will decode another instance of T, but at a different offset in the PDU to the original one. We could obviously decode a different type at this offset as well with a small change to the specification.

To summarize, the basic trick at the specification level is to use the bs **as** T expression to repeatedly process a bit sequence to "follow" the pointers and decode each individual block in turn. At the level of the specification the solution is reasonably concise. Ideally we would like to parameterize the types Pointer and T_at_offset on the type T so we can express this pattern once, and then reuse it in similar situations elsewhere. At present the APE compiler cannot cope with mutually-recursive types that contain type parameters, and so we cannot do this.

Whilst, syntactically, we have a mechanism for describing "non-linear" decoding problems, the challenge is to define the execution semantics of bs **as** T in a way that provides a satisfactory solution to the problem of building decoders from such a specification. Imagine a decode engine as having a queue of pending decode tasks, where a decode task consists of a bit sequence to be decoded, and a packet type to decode it against. We start by adding our PDU and root type as a new task and start the decoder engine running. When we encounter an expression such as bs as T we create a new task, add it to the queue, and then continue our current decode. When the current task is finished we pick the next task off the queue, if there is one, decode it, and keep going until the queue is empty. To avoid loops we record which tasks have been completed, and skip a task if it is a duplicate of a previously queued task. "Duplicate" in this context means the same bit sequence, the same packet type, and identical arguments to this type. Each new task would result in a new decode tree being added to the browser at the top-level of the decode tree display, as described in Section IV. The result, in our example, would be a sequence of decodes of type T. The APE IDE uses properties (Section V-C) to hide selected fields in a decode tree. Using this ability we would hide the initial decode to clean up the display, and also the prefix and suffix fields. But ideally we want to do better than this. What value should be returned from an expression such as bs **as** T? Suppose we introduce a new opaque datatype that represents a decode task ID of some form. When we add a new decode task to the queue, as a result of the bs **as** T expression, we return the ID of this task as the value of this expression. Such values in the decode tree view could potentially be interpreted as hyperlinks, allowing you to navigate around the structures that have been decoded.

There is one problem associated with this technique that we don't currently have a good answer for, namely how would we construct encoders from such specifications? That challenge is left for future work.

I. Type-checking

As mentioned previously, the APE type world is split into packet types and abstract data types. APE expressions are built from constants and packet type attributes. These attributes have types, and these types are constrained to be abstract data types. In other words the language does not allow an attribute to have a value whose type is an instance of a packet type. For those readers expecting the decode process to produce some form of decode tree, with packet type fields having as value the corresponding decode tree, this behavior might be unexpected. However, to support action-directed decoding (see Section V) such behavior is essential. The types of some attributes are fixed. For example, the size, count and choice attributes will always have values of type Int. But the value attribute does not have such restrictions. By default the value will be the bit sequence that was matched against this type reference, and will have type Bits. But the user can override this behavior by explicitly defining an expression for the value in a constraint. The type of this attribute can therefore be any datatype, either built-in or user-defined. The compiler uses unification-based type inference[14] to deduce the appropriate type for such attributes and the expressions that use them. The compiler also uses typing information to allow some expressions to be simplified. For example, an expression of function type is assumed to be implicitly applied to **this**, the field to which the constraint containing this expression is attached. Similarly if an expression is deduced to have a packet type, for example a field reference, then the compiler converts this to a value attribute expression for this type. Such heuristics were illustrated in Section III-C. The expression numBytes == header.totallength would be expanded by the compiler to numBytes (**this**#value) == header.totallength #value for example. The compiler could also use type inference to deduce the types of the arguments to packet type declarations. However, at present we do not do so, preferring the clarity of explicitly typed parameters to the conciseness of implicitly typed ones.

Type-checking bit sequences presents a particular challenge. The type Bits is used for bit sequences of arbitrary length. However, in many cases we can deduce the length of a bit sequence at compile time. Furthermore, during the decode process the representation used to hold a bit sequence may depend on the size of the bit sequence, and the alignment in the PDU. For example, a field that is always sixteen bits in length, and guaranteed to start on a byte boundary, can be simply represented by a pointer to the data. In contrast, a field whose length or alignment in the PDU is unknown at compile time will require a more complex representation. To capture such information the compiler introduces a family of specialized bit sequence types of the form Bits$n$. For example, the type

Bits5 is used for bit sequences whose length is guaranteed to be always five bits long.

The user does not have to use the more refined bits types in a specification. Instead the compiler adopts a two-pass approach to type-checking bit sequences. In the first pass all bit sequences are assigned the type Bits. Once the compiler has propagated size information through-out the specification, and has deduced the size constraints for each field, a second type-checking pass is performed. This allows the compiler to assign more specific fixed-length bit types to some expressions, together with the appropriate coercion functions between expressions. A good example of the need for such coercions is where the value of an alternation is required, and the values of the branches of the alternation are all bit sequences but of differing lengths. The refined bit sequence typing information is then used by the various back-ends to generate the appropriate representations for each bit sequence.

J. Alignment analysis

The compiler also performs an alignment analysis. By default the language assumes that a type can be matched against a bit sequence starting on an arbitrary bit alignment. Whilst this allows very flexible decoders to be produced, they will not be particularly efficient, and in many cases this is overly general. For example, do we really want to decode an IPv4 frame that starts three bits into a byte? The language allows the user to specify alignment constraints on types, using a similar notation to that adopted by Click[15]. Specifying the starting alignment for every type would be very tedious. Instead the user attaches alignment information to the "top-level" types in a module, typically those that are exported, and the compiler then propagates this information through-out the specification. The refined bit sequence typing, when used in conjunction with the alignment information, allows each back-end to choose the most appropriate representation of each bit sequence it manipulates.

IV. The APE IDE

Gone are the days when a simple command-line compiler interface was all that the average developer wanted. Integrated development environments such as Visual Studio, Eclipse and NetBeans provide a whole suite of tools to increase developer productivity. Unfortunately protocol specification tools have not kept pace. There is no fundamental justification for this, other than it being a much smaller market. Almost all the facilities that help you develop mainstream programming language applications are also applicable to protocol specification development. These include on-the-fly syntax-checking, navigation tools, language-aware editors, integrated source-level debuggers and so on.

One of the aims of the APE project was to explore ways of increasing productivity. Companies spend a lot of money employing people to write and maintain protocol specifications. Anything we can do to speed up this process has the potential to save a lot of money. The APE language was designed to reduce a lot of the syntactic clutter and overspecification present in many specification languages, pushing more of the work onto the compiler. But there is a limit to how much speed-up in development time you can get by changing the language alone. Equally important, in our opinion, is the development environment in which you create these specifications. It was our intention to provide a modern development environment for the APE, with facilities similar to those encountered in conventional IDEs such as Visual Studio and NetBeans.

Writing an IDE from scratch is a large task. Indeed to begin with we started down this path, intending to write a small and simple environment to support APE development. Unfortunately it quickly became apparent that users would expect and demand far more features in the IDE than we had the resources to provide. Many modern IDEs are extensible, allowing support for new languages to be quickly added. But this then raised the question of which IDE to support. Our intention was to provide a number of different back-ends to the compiler, targeting different languages. So there was no single IDE we could assume all our users would be using. Furthermore, we envisaged the decoders produced by the APE as forming only part of a much bigger application. Switching backwards and forwards between two full-blown IDEs could quickly become irritating and confusing to users. Fortunately there was a middle-ground option available. Some IDEs, including NetBeans and Eclipse, are built on top of a platform, a collection of modules providing the core functionality of the IDE. Users are free to build their own applications on top of this platform, allowing them to decide the mix of features they require to support their application.

The APE IDE is built on top of the NetBeans Platform[16]. An example layout of the IDE is shown in Figure 1. The main pane in this view shows an APE module describing an Ethernet frame being edited. The pane is tabbed, allowing the user to easily switch between modules. Furthermore, the pane can be "undocked" from the main window into one or more auxiliary windows, a useful feature when you wish to exploit multiple displays. Indeed, thanks to the support provided by NetBeans Platform, the arrangement of all the panes within the IDE can be altered to support the user's preferences.

The top-left pane show the list of opened projects, and the modules within these projects. It also shows the currently selected primary task and the back-end to use for generating code, in this example the DecodeTree task and the APE VM back-end. The pane on the bottom-left of the window displays a navigator view of the currently selected module. It displays the types and values defined in the module, groups generic types and their specializations together, and highlights those entries that are exported from the module. The toolbar at the bottom of this pane allows the entries to be sorted alphabetically or by source location, and can also hide private declarations and actions.

As described in Section III-J, the compiler propagates alignment information through-out the declarations. Alignment errors, where the user expects a type to be decoded starting on a particular boundary, but where the compiler fails to deduce this fact, can be hard to spot. The decoder will function correctly, albeit slower than expected. Types that are inferred to start on arbitrary bit boundaries are highlighted in the navigator view, allowing the user to quickly identify problems,

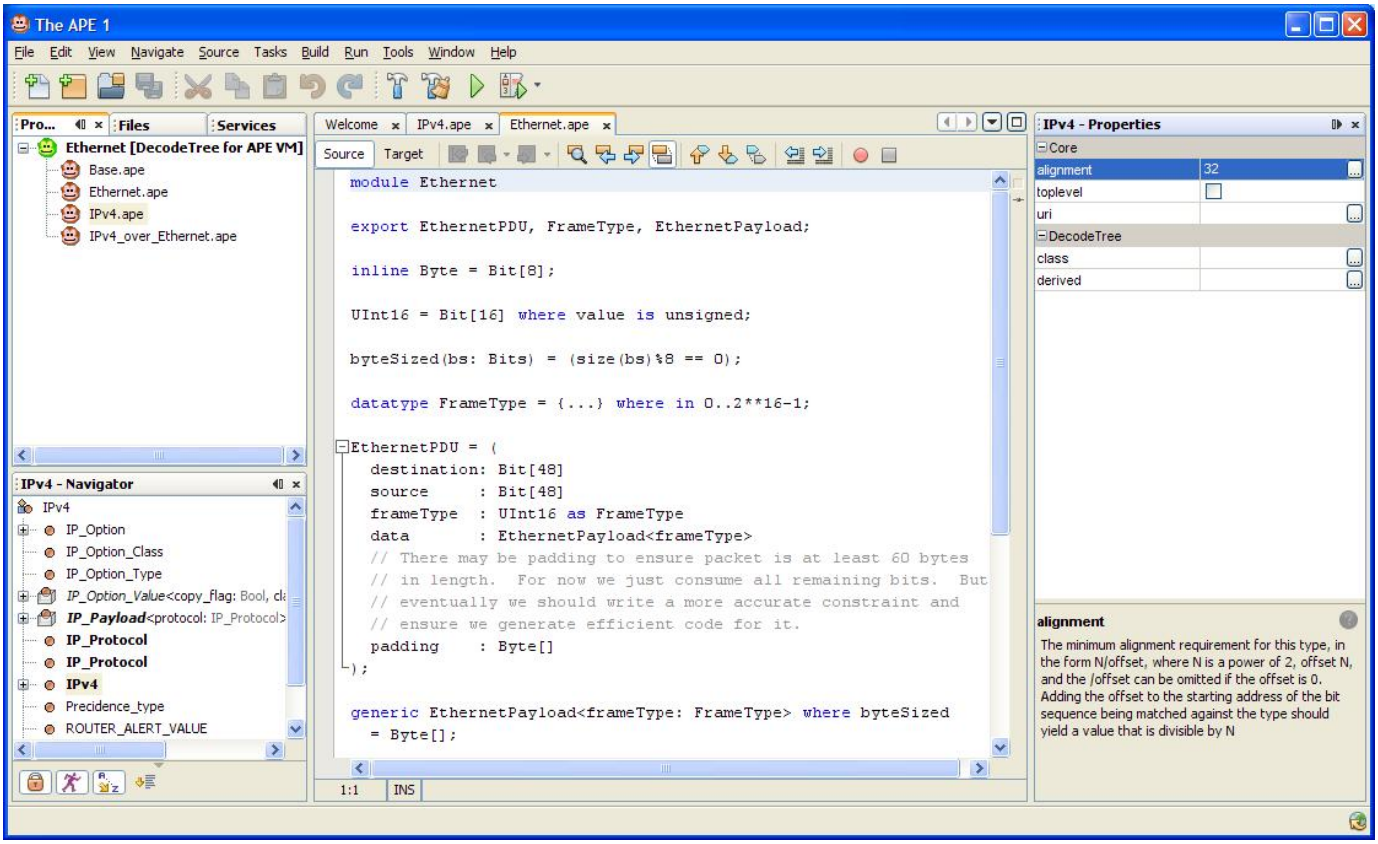


Fig. 1. The APE IDE

and add additional alignment constraints when appropriate. The entries for packet type declarations in the navigator view can be expanded to navigate the internal structure, the fields and subfields, of the type. Selecting an entry in the navigator view displays the properties associated with the entry, see Section V-C, in the properties inspector shown on the right-hand side of the window.

In addition to providing the usual editing functions, including seach/replace and control key short-cuts, the source editor is aware of the APE syntax. The system performs on-the-fly parsing of the APE code whenever the user pauses typing. Syntax errors are displayed in the editor panel, allowing errors to be quickly identified and fixed. Code folding and syntax coloring are also supported. When generating code for the APE virtual machine the generated code can be displayed in the Target tab of the source view.

Developing a specification that compiles without errors is only part of the challenge. We must also check that the specification can decode our packets successfully, and imposes the expected structure on these packets. This is another area where the provision of a IDE can really help. The IDE contains an integrated source-level debugger. The user can load a file containing a sequence of packets, and then run the decoder against each packet. Individual packets can be selected and the execution of the decoder stepped through, both at the source level and the virtual machine code level. The user can also set breakpoints both in the source code and the VM code.

One of the intentions of the APE, which we expand on in the next section, was to encourage greater sharing of specifications between different groups. The IDE provides a subversion-based interface to a specification repository[17], allowing the user to browse the available specifications, and load them into their projects.

V. Action-directed Decoding

The purpose of decoding within the APE system is to execute *actions*. These are fragments of application-specific code that are interleaved with the fields defining a protocol specification. Conventional parser generators, such as YACC[3] and ANTLR[5], adopt a similar approach. They allow actions to be embedded within a language grammar, and then arrange for these actions to be executed as part of the parsing process. By analogy, an APE decoder could construct a tree, representing the hierarchical structure of the PDU, and execute the actions as they were encountered during the decoding process. However, this is not the only possible execution strategy, or even the most desirable one. To understand why we must examine the differences between text-based parsing and the decoding of "bit-level" communication protocols.

A parser must usually process all of the input text to construct a parse tree. Even where only parts of this tree may be required by an application, the nature of text-based input can make it hard to skip over portions of the input text that are of no relevance. In contrast, bit-level protocols

frequently employ fixed length fields, or type-length-value (TLV) encodings for the fields. In such a setting it becomes much easier to jump over subcomponents that are of no interest to the application. Furthermore, applications that only require a partial decode of packet data are surprisingly common. Packet filtering, classification and indexing are typical examples of such tasks.

If every decoder had to construct a complete decode tree, simply executing actions as they were encountered during the construction process, then this would impose an unacceptable overhead on those applications that required access to only a small subset of each PDU. For this reason the APE adopts a rather unusual approach to protocol decoding. We take the view that the *only* purpose of performing a decode is to execute the actions embedded within a specification. If a part of the PDU does not need to be decoded to support the current action set then it should be skipped over. In the extreme case, where the specification contains no embedded actions, the packet may be skipped in it's entirety. We use the term *action-directed decoding* to describe this approach to protocol decoding.

The action-directed approach can yield very different decoders from the same specification depending on the actions embedded within it. For example, our IPv4 example might be augmented with a single action that uses the value of the protocol field to gather statistics about the relative frequencies of the different protocols carried by a link. An action-directed decoder for such an example could skip the analysis of almost all the fields within the IP packet. At the other extreme, a PDU browser that displayed the detailed structure of a packet would require a full decode of the PDU. Such behavior could be driven by actions that accessed all the fields within the PDU.

Although it might seem like action-directed decoding is simply a different way of viewing the decoding process, its influence permeates to the heart of the language design. To allow the "thinest" of decodes, where we only perform the minimum amount of work to support the actions, it is important to avoid the construction of any form of decode tree. This, in turn, requires careful definition of field visibility within a specification to avoid creating a situation where part of the decode has to create a partial decode tree just in case another part of the specification contains an action that needs to reference it later. If the language design is too liberal then we may not be able to avoid creating such structures. If the design is too restrictive then it may be hard, or even impossible, to express the necessary constraints to define some protocols of interest. The language design therefore requires a delicate balancing act between these two extremes. One point perhaps deserves further clarification. Whilst the decoder itself is designed to avoid the construction of a decode tree, this does not imply that an APE decoder cannot construct such a tree for a PDU. It simply implies that such construction must be performed by actions, perhaps mechanically generated, rather than being an essential and implicit part of every decoder.

That different applications might require access to different parts of a PDU seems quite natural. But if each application had to take a copy of the protocol specification, and then augment it with application-specific actions, we would quickly encounter problems. Our assumption is that a protocol specification would be developed, and debugged, prior to the insertion of application-specific actions. But bugs may be encountered later, and specifications naturally evolve over time. Incorporating such changes in multiple application-specific versions of the protocol source would be both tedious and error-prone. Although we envisage actions to be relatively small chunks of code, particularly when designed for real-time execution, they can quickly obscure the structure of the underlying protocol, further hindering the maintenance of the protocol specification.

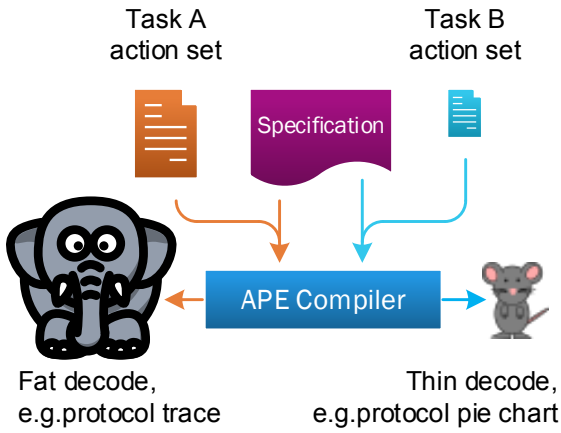


Fig. 2. Fat and thin decodes from the same specification.

The APE addresses this problem by storing the actions and the specification in separate files. The compiler is passed a protocol specification and the application for which we require a customized decoder. The compiler takes care of merging the specification and the actions for the selected application at compile time. The APE development environment (IDE) allows the user to edit actions within the context of the merged document. However, when such a hybrid document is saved the actions are split and stored independently of the specification. Furthermore, the editor can mark the specification portion of the document as being read-only, avoiding accidental changes to a shared specification during the application development process. The merging process is illustrated in Figure 2, which also emphasizes how the resulting decoder can vary in size and speed depending on the selected action set. The view of the merged document, as it appears within the APE IDE, can be seen in Figure 3. The grey portions of text in this example are read-only, showing that only the code for the actions is editable.

A. *Tasks*

As noted previously, there are potentially many different decoders that could be constructed from the same protocol specification, each one differing in the amount of detail that is extracted from the same PDU. We could take the view that there is a set of actions associated with each application, but this course-grained approach will prevent valuable opportunities for sharing. For example, an application might require

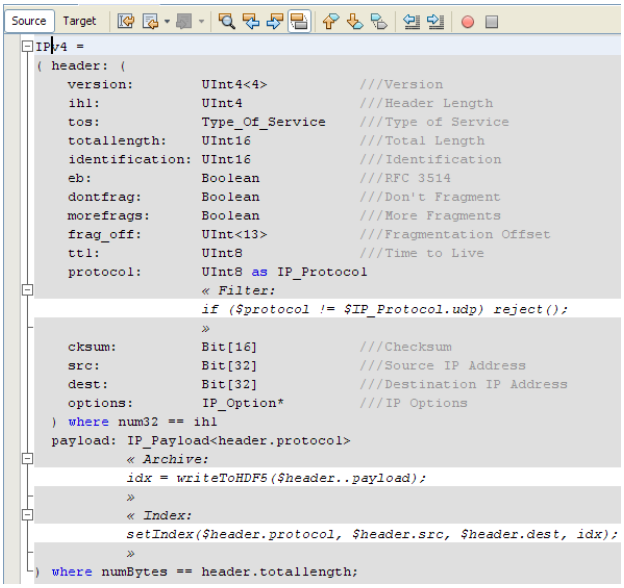


Fig. 3. A merged document within the APE IDE.

packets to be filtered before further processing. But a task like filtering may be common amongst multiple applications. We therefore associate actions with tasks, and an application then uses one or more tasks to build a customized decoder. Tasks may depend on other tasks, forming a directed acyclic dependency structure. This allows us to construct tasks whose purpose is simply to name, or group, another collection of tasks. This grouping allows us, without loss of generality, to specify a single task when building a decoder. We refer to the task we use for this purpose as the *primary task*. We can view the primary task as analogous to an application, but in practice the decoder + actions is likely to form only a small part of the application.

Clearly actions are expected to be executable fragments of code. But what language should these fragments be written in? To answer this question we must first understand how the decoders themselves will be implemented. The APE development system supports multiple back-ends. By this we mean that we are able to generate decoders for a variety of different languages. We might generate code for a low-level language like C, or for an existing specification language. By default the APE implementation generates code for an abstract machine, the APE VM. This is similar in spirit to the Java Virtual Machine[18], but specialized for the task of protocol decoding. The APE debugger uses this virtual machine to build a source-level debugger, a crucial component when developing a new specification. Whilst a software implementation of this VM is naturally slow, at least when compared to a decoder written in C, we have also developed an implementation in firmware. The current APE implementation can also generate decoders written in C++.

Just as there are multiple different back-ends for the APE, the intention is that there will also be multiple languages the actions can be written in. For example, when using a C back-end it may be natural to embed actions written in C within the specification. These actions would then be interleaved with the code generated for the decoder. If the decoder takes the form of code for the APE VM then we could also write actions in C. In this case, in a software implementation of the APE VM, we might use something akin to Java's JNI interface[19] to call the actions. In the case of an FPGA implementation of the VM, for example in a Xilinx Virtex 6, we might implement the actions on the embedded PowerPC.

For small actions, and in most cases we expect the actions to be small particularly when decoding at high speed, the overhead of calling between the APE VM processor and an auxiliary general-purpose CPU may be unacceptably high. In such cases we might be better off writing the actions as VM code sequences. This situation is analogous to embedding C actions in a C back-end, except in this case we are embedding VM code sequences representing actions into a VM code sequence performing the decoding. Of course it would be unrealistic to expect the user to write such sequences directly, just as users rarely write in assembler any more. However, we can imagine defining a variety of simple domain-specific languages, along with translaters into APE VM code. Typical examples might be languages for packet filtering, populating flow-specific data structures, and adding indexes to a database. Taken to extremes, it might even be useful to define actions that perform further decoding. For example, it might be possible to define a translator from an existing protocol decoding language to APE VM instructions. Given such a translator it would be possible to embed such code, as actions, to perform decoding of fields that were treated as opaque bit sequences by the APE. Why might this be useful? Well, it would allow existing decoding languages to exploit high-performance implementations of the APE VM. It might also allow the APE to support a wider range of protocols.

For an action to perform a useful task it must be able to access the current decoding context and retrieve the attributes of fields decoded within the scope of the action. Given the wide range of action languages we would like to support it seems unreasonable to expect the APE compiler to understand the detailed structure of each of them. Our approach is to embed simple metavariable expressions, such as $protocol and $payload#size, within the action text. Each occurrence represents a reference to part of the decoder state, for example the value of a decoded field. The compiler searches for metavariable occurrences and replaces each one by an expression that accesses these field attributes at runtime. The exact form of the replacement expression will depend on the action language and the currently selected back-end. Clearly not all combinations of back-end and action languages will be supported, partly because of time constraints, and also because some combinations are of little practical use. One of the properties of a task is the language in which the actions associated with the task will be written in, and the compiler ensures that the selected tasks are compatible with the chosen compiler back-end.

When prototyping applications using the VM back-end it would be tedious to have to keep writing domain-specific languages for all the different tasks we might wish to perform. And writing actions in a language like C would create its own problems in a prototyping environment due to the requirement

```
IPv4 =
  ( header:  ( ...
      ihl:          UInt4
      ...
      totallength: UInt16
      ...
      protocol:     UInt8 as IP_Protocol
      « Hello:
        System.out.println("Hello " + $protocol);
      »
      ...
    ) where numBytes = ihl*4
    payload: IP_Payload<header.protocol>
  ) where numBytes = header.totallength;
```

Fig. 4. A simple BeanShell action.

to write a separate file with the C actions, load a C compiler to convert them to a DLL, and then load the DLL into the VM environment. An alternative approach is to use a scripting language, such as BeanShell[20], to write our actions. Whilst clearly not practical for high-speed real-time action-directed decoding, it can be useful for offline analysis of packet data. Using BeanShell actions we can write a simple "Hello Protocol" example, shown in Figure 4, that illustrates the use of actions and metavariables.

B. Thin and Fat Decodes

The "Hello Protocol" application was an example of a thin decode. In theory we just needed to access a single field, at a fixed offset, to be able to execute the action. In practice we may need to do a bit more work. For example, the IPv4 PDU would typically be carried in the payload of an Ethernet packet, and so we would have to decode part of the Ethernet frame to determine the type of the payload, and to decode the IPv4 type if the Ethernet type field had the value 0x0800. Furthermore, without further hints, the compiler cannot guarantee there won't be any further actions embedded within the IP payload field. So the decoder may need to perform additional steps to explore this possibility. The exact behavior of the decoder is therefore not quite as simple as we might at first think. But we can safely assume that many of the fields within the IPv4 header would be skipped over by such a decoder. Another rather more plausible example of a thin decode is a packet filter where we examine one or more fields within the PDU and reject the packet if certain criteria are not met.

In a fat decode all, or most, of the fields need to be decoded. An example might be conformance testing where we wish to check that all of the packet fields are well-formed. A Wireshark-style packet browser also requires a full-decode of a packet, although in this case it can be done lazily. Some applications may require multiple decoders to be constructed. For example, we might use a packet filter to reduce the incoming packet rate to an acceptable level, and then couple this to an offline decoder that performs a more detailed inspection of the remaining packet data. Our case study in Section VI is an example of such an approach.

As mentioned earlier, the APE IDE encourages the clean separation of specification development from application-specific task development. The initial "Core" task does not allow actions to be inserted. When this task is selected as the primary task the user is allowed to edit all parts of the specification. The intention is that the user will start in this mode, developing the specification and checking it's correctness using the APE browser and debugger. Once the user is confident the specification is correct the user can then start developing application-specific tasks. The tasks editor is shown in Figure 5. When the user selects a new task as their primary task then the editor switches to a mode where the specification part of the document becomes read-only; only the text of the actions can be edited. The intention is to allow the same specification to be shared between multiple application task developers without any danger of the shared specification being inadvertently altered.

C. Properties

Actions drive the decoding process. However, this does not mean that these actions must be written explicitly by the user. Indeed in many cases writing actions is a tedious task that could better be performed automatically. For example, imagine annotating a specification with actions to construct decode trees. The structure of these decode trees is based on the structure of the specifications themselves. A compiler can therefore generate suitable class definitions to represent such trees, and then automatically insert actions to generate instances of these classes. In practice, however, the result is unlikely to be very satisfactory. A user may wish to suppress some fields from the decode tree as they contain no semantic value; their presence in the PDU is simply to help drive the decoding process. Length and choice determinants are good examples of such fields. A user may also wish to change the class names of the generated classes.

Fortunately the APE provides a simple but powerful mechanism to support such requirements and many others. A set of properties can be associated with each task. For predefined tasks these properties are also predefined. For user-defined tasks the user is able to define custom property sets. A property consists of

- a name,
- a type,
- a description,
- a scope,
- and a default value

The name, description and default value should be self-explanatory. The property type can be a primitive type such as String, Integer or Boolean. The type can also be defined as a simple enumeration. The property editor uses the type information to select the appropriate property value editor. For example, a property of Boolean type will display a check box that can be used to alter the value. An enumeration will use a Combo Box editor to allow the user to select the required value from amongst the permitted values.

Some properties only make sense when attached to an individual field. Other properties are designed to be attached to types, whilst others are intended to be attached to each module. The scope of a property indicates which contexts, module, declaration, or field, a property value can be attached to.

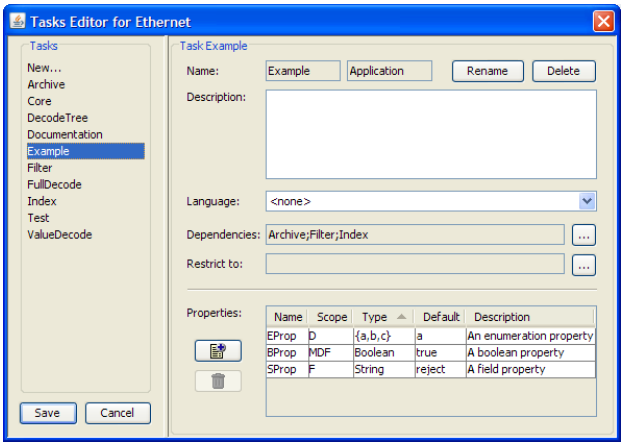


Fig. 5. Defining task properties

Figure 5 shows some properties being defined for the Example task. When an element such as a module, declaration or field is selected in the navigator view, the corresponding properties are displayed in the property view, grouped by task. Tasks may depend on other tasks, and every task will (implicitly) depend on the Core task, so in general there may be multiple groups of properties displayed in the property panel.

Some properties are predefined, namely those associated with predefined tasks. For example, the Core task has an alignment property defined on types that allows the user to specify the desired starting alignment for the type (see Section III-J). There is also a uri property to allow the user to provide a link to some external documentation relevant to the module, type or field. This will typically be a link to a section in the formal specification of the protocol being defined. A menu attached to the navigator panel allows the user to display such documentation if the URI is defined.

Some properties are automatically defined. For example, the APE supports "JavaDoc" style comments of the form `/** ... */` and `/// ...`. These comment strings are extracted from the source and made available as description properties associated with the Documentation task. This task also defines a format property, allowing the user fine-grained control of the display of each field. The intention is for property values to be available to actions, although this is not implemented in the current prototype.

VI. An Example

By this stage the reader should have developed some familiarity with the APE language, the IDE that supports it, and the concepts of task and action-directed decoding. However, what might be less clear is how it all fits together; the "bigger picture". This section describes a small example that will hopefully help to remedy this situation.

A. Preprocessing

The first phase of our example takes a stream of packets, and discards those that are of no interest to us. The remaining packets are written to an HDF5 packet table[21], [22]. To allow

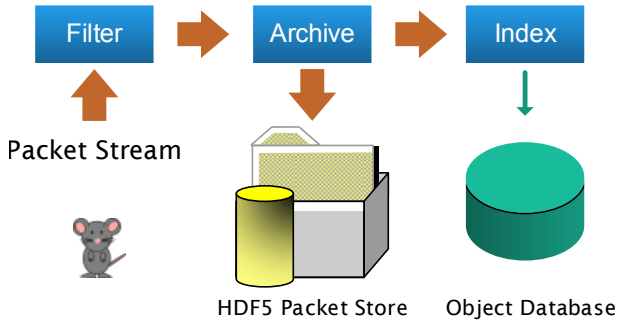


Fig. 6. The filter/archive/index task.

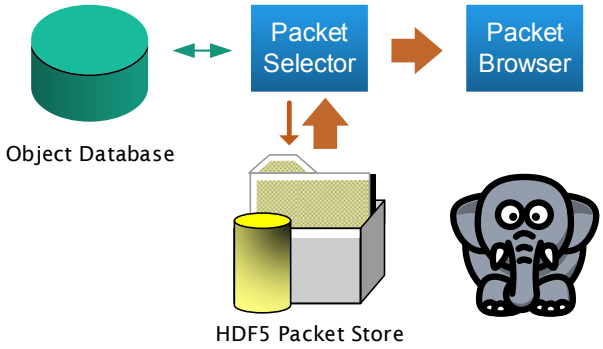


Fig. 7. The fat task.

these packets to be efficiently queried at a later time we also write some details of each packet to an object database. We will generate a thin decode for the component that performs the filtering and indexing, illustrated in Figure 6. Having archived the packet data we can use the database to query for packets of interest and perform more in-depth analysis of them. In our example the "in-depth" analysis simply consists of viewing them in a packet browser. This process is illustrated in Figure 7. Note that the example illustrates the use of both thin and fat decodes.

We assume we already have specifications for Ethernet and IPv4; our two decoders will be driven from these specification, but use different tasks. We start our example with the design of the thin decoder. We split the problem into a primary task, and three dependent tasks. Whilst unnecessarily complex for a simple example like this, it provides an opportunity to illustrate how larger tasks might be segmented into separate shareable subtasks.

The Preprocessor task depends on the Filter, Archive, and Index tasks. Invoking the APE compiler with Preprocessor specified as the primary task will merge in all the actions from these subtasks. In this example we will not associate any actions with the Preprocessor task itself. Its role is simply to group the other tasks. Although in our example we expect the archiving actions to be invoked after the filtering has taken place we choose to not create a dependency between these tasks. After all, in another application we may wish to archive an unfiltered packet stream. In this example we use the APE VM as our build target with the actions written using

BeanShell. A similar example could be constructed using the C++ back-end, with actions written in C++.

B. Filtering

To keep the example simple we use a trivial filtering criterion. We only want UDP packets; anything else should be rejected. The APE BeanShell environment contains a reject method that can be used to halt decoding of the current packet. Using this method our filter can be written as

```
...
protocol: ...
« Filter:
  if ($protocol != $IP_Protocol.udp)
      reject(); »
...
```

C. Archiving

The next step in our processing is to write any packets that pass the filter to an HDF5 packet store. In this example we assume the environment that initiated the decoder has already established a suitable database connection, and created a packet table. Our action simply has to add a selected portion of the decoded packet to the table. The following action achieves this task.

```
...
payload: ...
« Archive:
  idx = writeToHDF5( $header..payload ); »
...
```

The compiler will replace the metavariable $header..payload by an expression that, at runtime, will represent the bit sequence from the start of the header field to the end of the payload field.

D. Indexing

A filtered stream of packets can now be written to a packet store, but how do we get them back again? In some cases we may just want to access them sequentially. Examining them using a WireShark-style offline browser would be an example of this style of retrieval. In more interesting applications we may need to search for packets that meet certain criteria. To address this need we augment our application with a task that writes a précis of each packet to an Object Database, in this example db4o[23]. The exact form of the précis is unimportant. In practice it would depend on the nature of the queries we intended to perform. For example, we might include a reference to the previous précis in a flow in applications that required efficient traversal of packet flows. In our example we simply write the source and destination addresses, and the value of the protocol field, to the database. For such records to be useful we must also record a link to the corresponding entry in the HDF5 packet table containing the packet from which this précis was constructed. The Archive task stores a packet ID as part of the HDF5 packet insertion action. The Index task can add this value to the précis. However, for this to be well-defined it is important that the packet is inserted into the packet table prior to writing the précis to the database. If the two actions are attached to different fields then the field ordering can be used to ensure the actions are executed in the correct order. However, if we wish to attach the actions to the *same* field then we must record the fact that the Index task depends on the Archive task in the definition of this task. The compiler will then ensure the actions associated with these tasks are executed in an order that respects the dependency constraints. The task dependencies can be set in the Tasks Editor.

```
...
payload: ...
« Index: setIndex($header.protocol,
     $header.src, $header.dest, idx); »
...
```

To complete the first part of the example we construct a primary task, Preprocess, that depends on the three tasks we have just defined. Running the compiler with Preprocess selected as the primary task will result in the construction of a decoder that filters, archives and indexes a packet stream. The resulting decoder is optimized for the actions it has to support. The actions have no interest in the details of the payload, the IP options, or most of the other fields within the IPv4 header. The decoder will therefore skip over these fields without attempting to interpret them. At present it can be hard to visualize what must be decoded, and what can be skipped, without examining the assembly code generated by the compiler, a task not to be recommended for the feint-hearted! Eventually we plan to add support for color-coding a view of the original specification so we can quickly see what fields are directly used within the actions, what fields must be decoded to determine the sizes and offsets of those fields, and the decoding of these may lead to further dependencies and so on. It would also be useful to color-code the decoded PDUs in a similar fashion. In complex cases referring to a single field within an action may cause an avalanche of dependencies that result in the complete PDU having to be decoded. In practice such extreme behavior is unlikely to occur, but providing feedback on the relationship between actions and the resulting decoding optimizations should help avoid unexpected performance issues.

E. Analysis

We now consider the second part of our example. The APE development environment provides a browser for displaying the structure of packets, as determined by an APE decoder. The browser is similar in nature to a WireShark-style browser, but it can also handle partial decodes produced by thin decoders. In practice this just means the browser must be able to cope with bit sequences representing sequences of fields that have not been decoded. If we attached the browser to the decoder produced by the Preprocessor task we would find that many of the fields, for example the payload and options fields, would not have a value in the displayed decode tree. Contrast this behavior to a WireShark display, where every field that was present in the packet would be visible in the display.

What if we want to see all the fields within the PDU? The IDE contains a predefined task, FullDecode, whose sole purpose is to touch every field within the PDU. In other words

it creates synthetic actions that access every attribute of every field within the PDU, which has the side-effect of forcing all the fields to be decoded. We can build a decoder using this task, and then use the browser on this decoder to see a full decode of each packet. This decoder is an example of a fat decoder, in contrast to the thin decoder we generated for our preprocessor.

The browser can examine packets from a variety of sources. In addition to being able to examine conventional packet dump files it can also retrieve packets from an HDF5 packet table. We can therefore use the browser, coupled with our fat decoder, to examine the packets written to the packet table by our preprocessor. However, there may be millions of such packets, although this is unlikely given the interpretive nature of our current VM implementation. Trying to find packets of interest within such a table by sequential scanning is not very practical. Fortunately the IDE includes a BeanShell console window that can be used to write and execute BeanShell expressions. Furthermore, the browser is scriptable, and so we can write a script to retrieve the packet indices of all packets that meet a selected criterion from our db4o database. These indices can then be used to retrieve the raw packets from the HDF5 packet file for further decoding and analysis. In our example the subsequent process simply consists of performing a full decode and viewing the results in the browser. But it should be easy to see how we might extrapolate the same ideas to more realistic applications, and more efficient implementations of these ideas.

Our example, whilst simple, illustrates many of the key ideas of the action-directed approach to decoding. BeanShell actions show great promise for offline exploratory browsing and packet processing. However, support for other action languages is also important as BeanShell is not a practical solution in a real-time or firmware setting. The C++ back-end supports actions written in C++. It would also make sense to define a few small domain-specific action languages for common tasks such as filtering. This would allows the actions to be embedded directly within the decoding code rather than having to be called from it. In addition, the use of such languages allows actions using them to be easily retargeted at different back-ends. In some cases it may be more appropriate for a task to specify task-specific properties and then let the compiler convert these settings into actions. A good example is the accumulation of field information within some form of "breakout" structure. Different tasks may wish to collect the values of different fields, and some tasks may require access to the same fields. Using properties to generate the actions indirectly allows the requirements of all the tasks to be merged without the individual tasks having to collaborate, or even be aware of each other.

Although not obvious from our example, the decoder produced for the Preprocessor task performs slightly more decoding than might be expected. The cause of this additional work is the generic type defining the IP packet's payload. Modules such as TCP and UDP add their own specializations to this type, enabling the decoding of the payload to vary depending on the value of the protocol field. To enable a scalable solution modules must be compiled without knowing all the potential specializations of generic types that might be added. So when compiling the IPv4 type we cannot simply assume that we can skip decoding the payload. Other modules may define some additional specializations of the IP_Payload type that have embedded actions for our Preprocessor task, or one of its subtasks. To ensure we don't accidentally skip executing such actions we must perform the generic dispatch on the payload type. Of course if no other modules are loaded then the payload will be decoded as an uninterpreted sequence of bytes. And if other modules do add specializations but contain no embedded actions for these tasks then they will also treat the payload as an uninterpreted sequence. So the overhead is small compared to performing a full decode, but still larger than we might have expected. To avoid this additional overhead the compiler needs more information. We could perform a "full-program" analysis of all the specification modules making up an application, allowing the compiler to deduce all the static specializations of each generic type. Even this approach is not guaranteed to be safe when modules can be loaded dynamically. Other approaches are also possible. For example, in the current implementation we can add additional constraints to a task, using the Tasks Editor, to limit the modules the task can add actions to. In our example if the compiler could deduce that the Filter/Archive/Index/Preprocess tasks could only be added to the IPv4 module then it could safely assume the generic dispatch could be optimized away.

VII. Future Work

Whilst the APE project has covered a lot of ground there are still many areas requiring further exploration. In this section we briefly describe some of them.

A. Other build products

The APE could potentially produce a large number of different build products. These products fall into two main categories, dynamic and static. By dynamic we mean build products that are driven by the decoders generated by the APE. The simplest dynamic build products are probably filters. For example, we can simply insert actions at various points in a specification that check predicates on one or more decoded fields and then either reject or accept the packet. Our HDF5 example illustrated this process. Clearly if an existing application used a two stage approach of online filtering followed by offline analysis of the filtered stream then it would be reasonably easy to integrate the APE into such an application. Dynamic build products use actions to interact with the decoder. But these actions do not have to be written explicitly. For example, the APE supports task-specific properties as well as actions. We could therefore imagine attaching properties to selected fields indicating that we want to filter on them. The compiler could then automatically generate the appropriate actions along with a "driving" application that allowed patterns of acceptable field values to be specified for example. This choice between explicitly specified actions vs. implicitly generated ones will probably turn out to be a common theme. Initially, for a new type of build product, we would write explicit actions to prototype the application.

If the build product is likely to be used repeatedly, on lots of different protocols, then it will probably make sense, at some later point, to automate the generation of these actions, where this generation may be driven simply by the task itself or by task-specific properties. Given we have BeanShell, a Java scripting language, embedded in the APE we could even imagine allowing the users themselves to write scripts that generate actions.

Another common style of application is likely to be one where a decode tree is constructed for a PDU, and then application-specific methods are used to traverse this decode tree. The simplest form of such an application is probably a PDU browser, such as Wireshark[7]. This application uses a plug-in architecture for adding new protocols, and an APE backend could potentially generate code for such plugins.

Some applications require a decoder to produce a summary of fields of interest in some form of structure. In many cases it should be feasible to generate simple actions to populate elements of such structures. Indeed one might go one step further. You could imagine tagging various fields of interest, using properties, and the compiler could then potentially determine the best structures to hold the values of all these fields, together with the actions to populate these structures. Once again the best migration strategy seems to be to write actions explicitly initially, and to then automate the process once the pattern becomes clear.

B. Encoding

In the case of decoding we start with a bit sequence and create a hierarchical structure that matches this sequence. Of course, in the case of the APE we avoid building this structure explicitly, unless we write actions to do this, instead traversing the types in a hierarchical fashion. Or, looking at it another way, the call graph will end up mirroring the structure of the associated implicit decode tree. Given that encoding is the inverse of decoding, our starting point for an encode is to construct a hierarchical representation of the desired semantic content of the packet. Now whether we actually build an explicit tree, or simply execute a call graph that mimics this structure, is just one of our choices. For some protocols it is possible for the calls to directly populate the bits in the PDU, avoiding the need to build an explicit hierarchical structure prior to encoding. But in the general case it is difficult to do this, and we will need to build a tree and then traverse it to construct the PDU. The APE can already generate classes to represent decode trees, and these could be extended to support the encoding task. This sounds deceptively simple, but there are a number of issues which complicate this process.

The first question is which fields in a type should be represented as children in such an explicit decode tree. For example, consider something really simple such as an IPv4 header. We could demand the user builds a node for each field within this header. But there seems little point in constructing a node representing the version field, and then checking it has the correct value. Instead the system could simply synthesize such a node. The header length field is another good example. Requiring the user to provide an explicit length would be both unnecessary and error-prone. The run-time should be able to synthesize the appropriate field value at the point where the tree is encoded. Another good example is where a field is optional, and another bit field records whether the optional field is present or not. In that case we might simply use a null value, in the case of C++, to indicate an absent field. The encoder should then use this value to set the bit field appropriately. A CRC field can also be synthesized. These are all examples of a general process. Given a “full” decode tree we should be able to analyze the constraints, and structure of the types, to determine which fields can be derived from the structure, or values, of other fields. We can imagine pruning such elements from the decode tree, and it is the resulting tree that we want the user to build. As with many APE-related things, the analysis required to determine whether a field is derivable can be very simple in some cases, but almost intractable in others. However, such analysis is essential if we want to provide a natural interface to building PDUs.

So far we have been considering things that make the system easier to use. But is there anything akin to action-directed decoding that can be applied in such a setting? Suppose we wanted to generate a lot of test packets, at high speed, where each one had a lot in common with the rest, but where a few fields changed between each packet. This might be something as simple as a sequence number, or perhaps as complicated as the payload. In such cases we can imagine the tree representation of a PDU as having two kinds of nodes. The majority of them will be treated as before, but a few, the transient ones for want of a better term, will be modifiable. So we can imagine building a partial tree, with holes where the transient nodes will be. We can then use this as a template. We fill in the holes and encode the packet. We then fill in the holes with different values and encode the packet again. And we keep doing this until we get bored, or want to build a slightly different template. Now, of course, there is a simple way of doing this. We could wait until the holes have been filled in, and then just perform a normal encoding step on the complete tree. But this would obviously be very wasteful of resources, as in some cases almost all the PDU will remain unchanged. If a hole represents a field that will always be filled in using the same representation, with the same size, then we could simply build a PDU with gaps, and then when a hole gets filled we would simply fill the gap. The cost of instantiating the template each time would then be greatly reduced. Of course the situation might not be as simple as that. For example, the hole might represent an optional field. Assigning a non-null field to this hole might require setting a bit elsewhere in the PDU to record the presence of the field. So, in general, instantiating the template may require concatenating bit sequences that were computed at template creation time, shifting them, and so on. The challenge would be to analyze which fields were transient ones, and then maximizing the work that gets done prior to hole instantiation to minimize the cost of specializing the template each time. There is plenty of scope for heuristics and other optimizations in this area.

The APE IDE allows the user to attach task-specific properties to fields (and types and modules). We could imagine

attaching a new property to fields to tag the 'transient" ones, and then use this information to guide the construction of the encoders. Implementing such a *partial encoding* scheme would provide a nice complement to the action-directed approach to decoding, and would fulfill one of the original aims of the project, namely to generate encoders and decoders from the same specification.

C. Flows

The APE has largely focused on the syntactic aspects of protocol decoding. However, in reality most packets cannot be analyzed in isolation. Sequences of packets form flows, and state machines at each end of the communication pipeline control which packets are sent and resent. To fully analyze a packet sequence an application frequently has to be aware of such flows and state machines. This raises the question of what support, if any, the APE should provide in this area. We could take the view that the work of the APE stops after syntactically decoding a packet. At the other extreme we could extend the APE to allow the state machines to be defined, and the results incorporated into the generated decoders. Unfortunately the presence of timeouts in such state machines, and the fact that we may have to start monitoring a link part-way through a session, makes this task rather challenging to support in an automated fashion. There is, however, an intermediate stage that may be worth implementing in the APE. We could extend the constraint language to indicate which fields in a protocol should be used to represent a "flow". In some cases the order of the fields is significant, and in other cases it is not, and so we would need functions to capture both behaviors. To identify flows that span protocol layers we would introduce a notion of flow nesting. For example, at the IPv4 layer we would define flows using the source and destination addresses. In the TCP layer we would then derive a subflow based on the current IPv4 flow augmented with the source and destination ports. The intention would be to map flows to flow IDs, and then to allocate flow-specific areas of memory for use by the application code. Whilst the application would still have to manually perform such tasks as state tracking, at least some of the memory management chores could be offloaded to the APE.

VIII. Conclusions

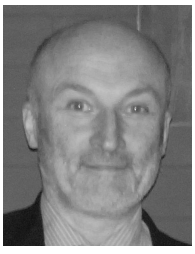
In this report we have described many of the accomplishments of the APE project. The resulting system can be used to support a variety of decoding tasks. The next step is to gain experience of applying these tools to some real examples, enabling us to evaluate their utility in practice. Furthermore, as described in Section VII, there is still plenty of scope for developing the language and tool chain further. We leave those tasks as an exercise for the reader.

Acknowledgment

The author would like to thank Tony Kirkham for his help in developing the APE virtual machine and firmware implementation.

References

[1] J. Klensin, *Simple Mail Transfer Protocol*, IETF Std. RFC 2821, April 2001. [Online]. Available: http://www.ietf.org/rfc/rfc2821.txt
[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol*, IETF Std. RFC 3261, June 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt
[3] S. C. Johnson, "YACC: Yet another compiler-compiler," in *Unix Programmer's Manual Vol 2b*, 1979.
[4] C. Donnelly and R. Stallman, "Bison: The Yacc-compatible parser generator," http://www.gnu.org/software/bison/manual/pdf/bison.pdf, May 2006.
[5] T. Parr, *The Definitive ANTLR Reference*. Pragmatic Bookshelf, May 2007.
[6] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2," http://www.w3.org/TR/soap12-part1/, 2007.
[7] U. Lamping, R. Sharpe, and E. Warnicke, "Wireshark user's guide," http://www.wireshark.org/download/docs/user-guide-us.pdf, 2008.
[8] P. McCann and S. Chandra, "PacketTypes: Abstract specification of network protocol messages," in *Proceedings of ACM SIGCOMM*, 2000.
[9] D. E. Knuth, "The genesis of attribute grammars," in *Proceedings of the international conference on Attribute grammars and their applications*, 1990.
[10] A. Shalit, *The Dylan Reference Manual*. Apple Press, 1996.
[11] C. Chambers, "Object-oriented multi-methods in Cecil," in *ECOOP'92*, 1992.
[12] IEEE, *IEEE 802.16 (WiMAX) Specification*, http://wirelessman.org/pubs/80216Rev2.html, Std., 2009.
[13] Adobe, "Tiff," http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf, June 1992.
[14] B. Pierce, Ed., *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.
[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, August 2000.
[16] T. Boudreau, J. Tulach, and G. Wielenga, *Rich Client Programming: Plugging into the NetBeans Platform*. Prentice Hall, 2007.
[17] C. Pilato, B. Collins-Sussman, and B. Fitzpatrick, *Version Control with Subversion*. O'Reilly, 2008.
[18] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
[19] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1999.
[20] P. Niemeyer, "Beanshell scripting: Lightweight scripting for java," Web site, 2005. [Online]. Available: http://www.beanshell.org/home.html
[21] hdfgroup.org, "Introduction to HDF5," Web site, 2007. [Online]. Available: http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html
[22] ——, "HDF5 packet tables," Web site, 2007. [Online]. Available: http://www.hdfgroup.uiuc.edu/HDF5/doc_1.8pre/doc/HL/RM_H5PT.html
[23] R. Grehan, "The database behind the brains," *db40 White Paper*, 2006.



Kevin Mitchell received the B.Sc. (Eng) degree in Computer Science from Imperial College, London in 1979. He received the Ph.D. degree in Computer Science from the University of Edinburgh, Scotland, in 1986. He was a lecturer at Edinburgh University until 1996, before moving to Harlequin to work on their Dylan and ML compilers. Since 2000 he has worked in Agilent's Measurement Research Laboratory. His research focuses on compiler and tool development for domain-specific languages, and parallel systems.