Streams and Serialization Protocols: A Formal Approach

Cosmin Truta
Department of Computer Science
University of Toronto

May 5, 2000

Abstract

This paper proposes a way to elegantly approach a problem that used to be ugly by its nature: to save and restore (serialize) objects to and from magnetic support, in a portable, machine- and language-independent fashion. The solution also addresses intra- and inter-process communications, because the problems raised by interchangeability and portability and are similar in both cases. To achieve this, a high-level description language that addresses low-level issues is designed. Translation from a high-level specification to a certain programming language can be done automatically, thus considerably reducing the effort that programmers used to spend when implementing complicated, standard or non-standard serialization protocols, and increasing the code reliability at the same time.

1 Introduction

To persist is to hold to a state or a goal. In computer programs, an example of persistence is retaining information between application invocations.

Objects that are created when an application runs — windows, dialog boxes, collections, and so on — are temporary. They are constructed, used, and destroyed as the application proceeds. Objects can appear and disappear as they enter and leave their scope, or when the program terminates. By making these objects streamable, they can be saved, either in memory or file streams, so that they persist beyond their normal lifespan.

There are many applications for persistent objects. When saved in the shared memory, they can provide interprocess communication. They can be saved permanently on disk using file streams, and can be transmitted on the network to other systems. They can then be read back and restored by the same application, by other instances of the same application, or by other applications. Efficient, consistent, and safe streamability must be available to all objects that need it.

Serialization is the process of writing an object to or reading it from a persistent storage medium, such as a disk file. However, in this article, we consider serialization as being the above processes, and the process of transmitting and receiving data between two applications, no matter if this happens on a single computer and in the same memory, or on a local network, or on the Internet.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the value of its member variables, to a stream. At the same moment, or later, the object can be re-created by reading, or deserializing, the object's state from the stream. Serialization should handle all the details of object pointers and circular references to objects that are used when serializing an object. A key point is that the object itself is responsible for reading and writing its own state. Another key point is the consistence between reading and writing operations, which is, when reading an object to restore its state, the restored state must correspond to the state of the object when it was written into the stream.

Several libraries supply built-in support for serialization, and some of these libraries are designed in an object-oriented manner. As an example, *Microsoft Foundation Classes (MFC)* provide support for defining the serialization algorithm for each class, and they seamlessly take care of serializing pointers or polymorphic collections, by using their own serialization protocol.

Everything that enters into a computer via a wire from the Internet, is based on communication protocols. From the lowest physical level to the highest application level, bits and bytes have a very precise order and meaning, carefully chosen and then publicly specified by the authoring organizations or companies. In fact, the entire Internet is specified by a huge collection of articles called *Requests for Comments (RFCs)*. Usually, specifying a communication protocol means writing an informal document that describes the purpose, the content and the meaning of every single element from the file stream. Sometimes, this specification is accompanied by a reference library, written usually in C, that offers functions to read and write from the standard stream. Implementors might choose to use this library, or to implement their own. Their choice is determined mainly by the performance of this reference library and by the ease of understanding, using and extending it, but other arguments such as the difficulty of using it in another programming language with a different paradigm, or simply the fact of having too many bugs, may also apply.

This paper addresses the above problem, by proposing a method and a formal language to specify serialization protocols. They are intended to stand at the same level as the languages accepted by LEX or YACC. Those tools accept grammars presented in a well-defined fashion as input, and produce C code as output. The generated code is not quite human-readable, but it is very optimized, and, more importantly, if a correct grammar is given as input, we are sure that the output code is correct. That output C code hardly needs any other modification before being further compiled into assembler code, object code or machine-executable code. So, while the input for LEX and YACC is a formal specification written in a problem-oriented language (that accepted by these tools), it is also an executable program, since no further action from the programmer's part is needed.

The rest of this report is organized as follows: Section 2 presents a motivational example, from which we develop basic features of the language. Section 3 describes an advanced feature of the language: the ability to manage collections of elements. We describe reasoning about correctness in Section 4. Section 5 compares our approach with related work, and Section 6 concludes the paper, with summary and an outline of future research directions.

2 Designing the Method

2.1 Design Goals

It is relatively easy to make objects serializable. However, trivial approaches lack important features such as:

- *Streamability*: files can be read and written serially, thus allowing the file format to be used as a communications protocol for on-the-fly generation and display of images.
- *Portability*: complete hardware- and platform-independence.
- *Reliability*: straightforward detection of data corruption.
- *Simplicity*: developers should be able to implement this model easily.
- *Interchangeability*: any standard-conforming decoder must read all conforming files.
- *Flexibility, extensibility*: the format allows for future extensions and private add-ons without compromising interchangeability of the basic format.
- *Robustness*: the design supports full file integrity checking as well as simple detection of common transmission errors.
- *Ancillary information*: textual comments and other non-critical data can be stored within the file.

The language proposed in this paper is intended to completely separate all the details related to serialization from any other application-specific detail. The job of producing correct, compilable code (C, C++, Java etc.) is left to an automatic tool capable of doing that.

The design of this method is at its very beginning. While its final purpose is to be able to specify most, if not every existing serialization protocol, and to be helpful when designing and publishing new protocols, it can now work only with relatively simple examples. The guiding principles of designing the method are *simplicity* (keep the language syntax as simple as possible), *intuitiveness* (specifically, the similarity of the language to the existing programming languages can greatly reduce the learning curve) and *maximum performance* (the ability to generate executable code as if written "by hand" and tweaked for maximum execution speed and minimum compiled/object code size).

2.2 A Small Example

Let's consider the simple case when an application has a structure with two fields that must be saved in a file. The examples that follow are written in C.

```
struct SimpleStruct
{
    int i;    /* an integer */
    char *s;  /* a string */
};
```

Even in this simple case, two questions appear:

— How to represent the integer?

   a) As raw text, ended by a new line, by a space, or by any other non-digit character.

   b) As a binary sequence of bytes.

   In case where the answer is *b*, there is no problem if the reading and writing operations are always performed on the same machine, but otherwise, low-level aspects, namely integer size (2, 4 or 8 bytes) and endianness, must be considered.

— How to represent the string?

   a) The contained text, ended by a new line.

   b) The contained text, ended by a terminating `NULL`, in case that there can be a multi-line text stored in a single variable.

   c) The contained text, preceded by an integer number that contains the length of the text, in bytes.

   d) The contained text, not preceded or ended by any other information, since the end of the string corresponds to the end of the file.

   It is obvious that, to choose the answer *d*, the string must be the last in the structure, and if there would be another string in that structure, a different approach must be used for it.

Let's say that we decide for the simplest approach: represent both as raw text. Now, to the implementation. The most trivial code to do this would be something inside a function:

```
/* ... */
fp = fopen("output.dat", "wb");
fprintf(fp, "%d\n", s.i);
fprintf(fp, "%s\n", s.s);
fclose(fp);
```

It is easy to imagine how the code for reading the stored information will look like. Or, there can be separate functions to do this:

```
void readSimpleStruct(FILE *fp, SimpleStruct *s)
{
    fscanf(fp, "%d", s->i);
    s->s = (char *)malloc(100);  /* the biggest size is 100 */
    fgets(s->s, 100, fp);
}

void writeSimpleStruct(FILE *fp, SimpleStruct *s)
{
    fprintf(fp, "%d\n", s->i);
    fprintf(fp, "%s\n", s->s);
}
```

In this simple case there is at least one thing that the programmer must take care of, and that is allocating storage for the string. If someone wants to have an arbitrary size, the code would get fairly complex. The code above is not something to understand easily or tell other people so that they can read some data into their applications, for exchange purposes. Another problem is that different coding styles look very differently, so detecting that two pieces of code have the same functionality is non-trivial.

Instead, we can think of describing the structure in the following way:

```
struct SimpleStruct
{
    /* This is a comment. It is similar to a C comment */
    // This is another comment. It is similar to a C++ comment

    int i;
    string s;

    read  // The reading method is specified here
    {
        read i;
        read s;
    }

    write  // The writing method is specified here
    {
        write i;
        write s;
    }
}
```

To interpret the sequence, we say that there is a structure `SimpleStruct` that has two fields: an integer field named `i`, and a string field named `s`. To read the structure from a stream, `i` must be read first, then `s` must be read after it. To write the structure to a stream, first write `i`, then write `s`. The language accepts comments in the same manner as C and C++.

So our language is born. We expect from it to be able to have any number of structures, and to be able to have members of any built-in or user-defined type. `SimpleStruct` is an example of a user-defined type, and if a member of this type appears in another structure, it will be read and written according to the reading and writing methods specified in the body of `SimpleStruct`.

2.3 Low-level Aspects

2.3.1 Binary vs. Textual Encoding

Nothing was said about the size of the integer, the representation method, the endianness etc. In programming languages, all these details are transparent, because for every CPU these parameters are fixed, and the compiler uses them automatically. Storing numbers as strings satisfies our goal of portability, because those numbers are converted back and forth by routines that are separately written for every language and for every CPU type.

When numbers are stored in binary format, the problem becomes more complex. We expect to have our structure read on any kind of machine, so we cannot assume that the machine on which the structure is read has the same parameters as the machine on which the structure is written.

Why encode numbers in binary? Humans prefer to encode information as text, because that information is human-readable and human-editable. A binary encoding adds extra cost in that more programming effort is needed, for both encoding and decoding. Whenever adding new fields in a file structure, changes must be incorporated in all software that process that file structure, and those changes are not obvious and they are subject to errors. Overall, the complexity of software increases.

But textual encoding is not an option when the speed of processing is critical (for example, in remote procedure calls), and when the amount of data is very large (in multimedia streams, or in large databases). In the second case, data compression is often considered, and that certainly increases the software complexity even more. That's why an automatic way of producing executable code to process binary files would be very helpful. As a matter of fact, even if the method proposed is able to process both binary and textual streams, we expect it to be most effective on binary streams, because complex structures are usually encoded in binary.

2.3.2 Encoding Integers

The encoding of integers depends on several parameters. As with text vs. binary, there are some parameters that are hidden under the hood of each programming language and must be uncovered here for portability purposes.

The most important important parameter is the integer representation. Since almost every machine represents integers as 2's complement, we decide to do it same way. If it happens, ultimately, to read our structure on a machine with another representation (such as *BCD — Binary Coded Decimal*), that machine is responsible of converting from the 2's complement back and forth to its representation.

Another parameter is the integer size. It is well known that the size of the integer is different on different machines, and that integers can have values in a wide variety of ranges. Most of the time, the integer occupies 8 bits (*byte*), or 16 bits (a sequence of two contiguous bytes, also known as *word*), or 32 bits (*double word*), or 64 bits (*quad word*). This is very machine-specific, but it is critical. When translating our specification into a programming language, say C, we must know the machine or the machines on which the compiled sequence is to be executed. So we will translate the double word into an *int* if the machine is a 32-bit machine, or into a *long* if the machine is a 16-bit machine. We will not care about that, since the translation will be done automatically. We care about our language, and we want to make its syntax as simple as possible. We drop the word `int` since it does not tell us many things about itself, and we introduce 4 other reserved words:

`byte` — to represent 8-bit integers
`word` — to represent 16-bit integers
`dword` — to represent 32-bit integers
`qword` — to represent 64-bit integers

Endianness is pretty tricky, and it cannot be disregarded. Today's machines are divided into *little-endian* machines (Intel, etc.) and *big-endian* machines (Sparc, Motorola, etc.) *Little-endian* means the integer $(hex)ABCD$ is stored in a contiguous sequence of bytes, as $(hex)CD(hex)AB$, while *big-endian* means that the same integer is stored as $(hex)AB(hex)CD$. When transferring the integer value $(hex)ABCD$ directly from a little-endian machine to a big-endian machine and vice-versa, it will be received on the other part as $(hex)CDAB$.

One might say that considering the above machine parameters will lower the level of our language. It is true. But it will give us a better and an easier way of specifying them, as they are compulsory. In higher-level languages, reading a little-endian, 32-bit integer can look like this:

```
void readInt(FILE *fp, int *i)
{
    *i = getc(fp) + (getc(fp) << 8) +
        (getc(fp) << 16) + (getc(fp) << 24);
}
```

It is certainly not high-level. It will get uglier when taking care of some compiler optimization that could (in an incorrect way, but possible to happen) "simplify" the whole expression as

```
    *i = getc(fp) * 0x1010101;
```

and even worse when there is a need to optimize the portable code for speed:

```
#if defined(__16BIT__)
typedef long portableInt;
#else /* defined __32BIT__ */
/* in this case long can also fit, but it may lose performance */
typedef int portableInt;
/* for now, we don't take care of 64-bit machines */
#endif

void readInt(FILE *fp, portableInt *i) { ... }
void writeInt(FILE *fp, portableInt *i) { ... }

#if defined(__INTEL__)
# define READ_INT(fp, i)  fread((void*)&i, 4, 1, fp)
# define WRITE_INT(fp, i) fwrite((void*)&i, 4, 1, fp)
#else /* defined __SPARC__ etc. */
# define READ_INT(fp, i)  readInt(fp, &i)
# define WRITE_INT(fp, i) writeInt(fp, i)
#endif
```

So if we let our language accept low-level constructs, it will be possible to write binary stream specifications more clearly, and focus on important issues. The specification compiler can automatically generate source code like above, for any programming language.

We will choose the following words as reserved words in our little specification language:

`text` — to mean that integers are represented as text
`little_endian` — to mean that integers are represented as binary, little endian
`big_endian` — to mean that integers are represented as binary, big endian

Syntactically, these reserved words will stand as attributes for types, the same way as other type attributes are specified in C: `text dword` means the 4-byte integer is represented as text.

We preserve the C convention of specifying if an integer is signed and unsigned. So the sequence `unsigned word` will be very probably translated into `unsigned short` for the C language, and into `word` for the Pascal language, while the word `byte` will mean `char` for C (as it does not have other type for expressing 8-bit integers), and `shortint` for Pascal.

The above reserved words are used as attributes of a field declaration (the notion of attribute being the same as in the C language). So, in our example, the declaration for the integer field `i` would look as `text dword i`. Similar approaches can be taken to specify string fields, as string fields can also be represented in different ways.

Since it is expected for a protocol definition to decide upon a certain representation type for most or all of its elements and to use it consistently, it is useful to get rid of the syntactic burden of writing the same thing for most of the fields. We can think of a protocol that always uses unsigned integers, represented in binary, big endian convention. So to describe this protocol in a simpler way, we could allow the following constructs in the beginning of the specification:

```
default unsigned;
default little_endian;
```

The optional directive `default` can appear only at the beginning of the definition file, and its effect is to override the default attributes of a field. When a word field appears, it means that it is unsigned, and it is internally stored in binary, etc. The effect of the default directive on a certain field is void if that field has local attributes that override the global, default attributes.

2.3.3 Encoding Other Simple Types

Even if the character type is the simplest type, it still needs some special consideration. ASCII characters fit in one single byte, but if internationalization is considered, the two-byte Unicode set must be used. We use the known keyword `char` for ASCII characters, and `wchar` for Unicode wide characters.

Floating-point numbers shall be encoded according to the IEEE specifications. The keyword `float` will stand for for 32-bit IEEE floating point, and `double` for 64-bit IEEE floating point. There are also other floating-point encodings, but, for portability reasons, we stop at these. By designing separate structures with their own reading and writing methods, any kind of fixed- or floating-point numbers can be represented.

2.4 Finishing the Example

The last feature to add in order to simplify the way to express our file structure comes from the observation that the two component fields of the internal structure come one after the other, in the same order as they are read or written. So probably we wouldn't need special read or write methods. This is true for our example, but not true for non-trivial examples. But we can think of a simplification if we state that, in the case when read and write are not supplied, the existence of default read and write methods is assumed. The effect of these default methods is the reading and the writing of the entire contents of the structure, having the fields stored in the stream in contiguous bytes, in the order in which the fields are syntactically listed in the structure definition.

Finally, our first example is specified as:

```
default text;

struct SimpleStruct
{
    signed dword i;  /* equivalent to "dword i" */
```

```
    string s;
    /* the reading and writing methods
       are automatically provided by the compiler */
}
```

Note that the implementation details are hidden.

3 Managing Collections of Data. Polymorphism

3.1 Homogeneous Collections: Arrays

The concept of *array* (a homogeneous sequence of objects) is supported by the majority of the programming languages. Often, there is a need for serializing arrays. For example, if we consider a movie, we expect to have a structure to represent a frame, and then we save the movie as an array of frames. We must support variable-size arrays, as many examples need them. This must be somehow translated into languages like C, that provide native support only for fixed-size arrays.

Often, when saving an array into a file, the following approach is used: first store the size of the array, then store the array itself, element by element. When loading it, read the size n, allocate (statically or dynamically) a contiguous memory block to store at least n elements, and read those elements into that memory block.

```
struct MyStruct
{
    // ...
}

struct MyArray
{
    dword size;
    MyStruct[size] data;

    // the reading and writing methods are provided by default
}
```

The argument of the [] operator can be an expression which involves constant values, and values previously encountered in the stream. The access to a certain element of the array is specified via the same [] operator, using a 0-based index as an argument.

If there is a need to access a value outside of the structure (which can happen), then we must give such access somehow. Parameters passed directly to structures are a way to solve this problem.

The following example illustrates a fragment of almost any image format definition, provided that image format has an RGB palette:

```
struct ColorPalette(byte bit_size)
{
    struct ColorPaletteEntry
    {
        // internal structure definition, visible only here
        byte red;
        byte green;
        byte blue;
    };
    ColorPaletteEntry palette[1 << bit_size];  // the array
}
```

3.2 Heterogeneous Collections

In the previous example, we had a color palette structure which contains elements of the same type: color entries. This means, after reading the first element from the collection which is a color entry, we expect the second element to be a color entry too. Either when reading a member of a structure or a member of a collection, the type of that member is known, so it is possible to choose the appropriate reading method.

However, many times we need to deal with collections containing elements whose type can be chosen randomly, among a list of possible types. For instance, JPEG files ([7]) may contain information regarding quantization, Huffman compression, arithmetic compression etc., and we expect to represent each type of such information in a corresponding structure. But it is not possible to anticipate the exact order in which these structures will be read, and it is not possible to anticipate whether an element of a certain structure type will be present in an actual file. Even if there are some ordering constraints regarding the JPEG structures, they are not deterministic.

A *heterogeneous collection*, or a *polymorphic collection* is a collection that contains elements of various types without offering the possibility to determine the type of each element statically. Therefore, there is a need to determine the element types dynamically.

On the disk, a collection will reside in sequences of contiguous bytes. It is possible to determine the type of information stored in a sequence of bytes, if that sequence is accompanied by another special sequence called *signature*. The signature is usually short (1 to 4 bytes), to make possible the quick computation of a boolean expression which will determine whether the rest of the byte sequence represents a certain type.

The algorithm to determine the type of a certain element when reading the collection is:

```
read signature
for each type
  if type.check_signature(signature) then
    this is the type
```

```
endfor
```

The next bytes will be read and interpreted according to the found type.

When reading an element of this type, the signature must be checked, and when writing it, the signature must also be written. Each type that can be present in a heterogeneous collection must have a signature identification scheme (formula or algorithm) that uniquely distinguishes this type among the other types. This scheme will be specified as a boolean expression. Since in many cases the boolean expression is an equality check, we will add the possibility to specify signatures as constant values.

We will show later how to prove the correctness of an identification scheme.

Below are presented two structure types which can be present in a polymorphic collection:

```
struct Struct1
{
  // the signature is specified by a constant value
  const char signature = 'A';
  // the rest of the information goes here
  int info1;
}

struct Struct2
{
  // the signature is specified by a boolean expression
  char signature
  {
    signature >= 'B';  // boolean expression
  }
  // the rest of the information goes here
  int info2;
}
```

The collection items are organized in *sets* or *bags* (depending on whether the count of the appearance of each item is important) when the ordering is not important. Otherwise, those items are organized in *sequences*.

As the signature must be present, we cannot have polymorphic collections containing simple types such as integers. The syntax and the semantics used to define and access a polymorphic collection is similar to that defined by the *UML Object Constraint Language (OCL)* [10]. The main difference regarding OCL is the fact that the possible types need not be in any relation with respect to each other, while in OCL there must be inheritance relations among the contained data types.

For example, `set<Struct1, Struct2>` means that we have a set of elements which can have either the type `Struct1` or `Struct2`. At the same time, `sequence<Struct3, Struct4, Struct5>` means that we have a sequence of elements which can have any of the types `Struct3`, `Struct4` or `Struct5`.

When generating executable code, the reading algorithm will be adjusted so that it will start reading the signature bytes, then it will try to match them against the signature identification algorithm of the first type. If the signature is specified as a constant value, the read bytes are checked against that value. If the signature is specified as a boolean expression, it is checked if that expression is satisfied. In case of success, the reading of the structure will continue; otherwise, the signature of the next possible type will be tried. The reading is repeated for all the subsequent elements previously stored in the file, until no other successful reading is possible, due to either signature mismatch or end of file.

It is a real challenge to do automatic translation from a formal description to a programming language like C, C++ or Java. It is difficult to translate to any programming language that does not support polymorphism, or has limited polymorphism (e.g., restricted by inheritance relations), and it is subject to future research. On the other hand, automatic translation to Smalltalk or other untyped languages would be very simple and straightforward, as in those languages, any object can have any type, at any moment.

4 Correctness

4.1 Coping with Data Corruption: Validation and Recovery

It is known that the data stored into a file or transmitted via a stream is subject to corruption. Sometimes, software that reads data from such corruptible sources performs extra checks, to be sure that it does not access invalid addresses, and the program does not take an unexpected path.

Let's take the example of the color palette, and let's design a self-contained color palette structure. This means that the bit size, previously passed as a parameter, is now a field in the structure.

Color palettes usually have at least 2 colors and at most 256 colors, for performance reasons. When storing a color palette in a stream, we know that we store valid values, but we cannot be sure about them when we reload them. What we can do, is to enhance the description language by adding the possibility to specify *invariants*. An invariant is a property that characterizes a structure. The structure is *valid* if the invariant property holds. To specify an invariant, simply enumerate logical expressions inside the body of the new special method called `valid`. In our case, to have a valid palette with a number of colors between 2 and 256, we must have the bit size between 1 and 8.

Sometimes it is possible to have a field with an incorrect value, and to know how to correct that value. We will introduce a method called `recover` to describe the actions to be taken when the `valid` check fails. When translating the whole specification into programming languages, the translation of the `valid` method gets called right after reading. The `recover` methods can be ignored, if the application wants to have another behavior in case of errors. It is not recommended to ignore (i.e. not to translate) the `valid` methods, however.

The semantics of `valid` and `recover` is expressed by the following pseudocode:

```
// validation / recovery performed after reading
read
call valid
if (!valid) then
    // the data read from input is corrupted
    if exists recover then
        call recover
        call valid      // redo the validation again
        if (!valid) then
            error       // the recovery has failed
        endif
    else
        error           // there is no way to recover
    endif
endif
// end - successful reading

// validation / recovery performed before writing
call valid
if (!valid) then
    // the data prepared for output is corrupted
    if exists recover then
        call recover
        call valid      // redo the validation again
        if (!valid) then
            error       // the recovery has failed
        endif
    else
        error           // there is no way to recover
    endif
endif
write
// end - successful writing
```

For practical implementation, if it is possible to prove that the actual recovery method always produces valid files, the second validation test can be removed, thus optimizing the code.

The following example illustrates the use of `valid` and `recover`:

```
struct ColorPalette
{
    struct ColorPaletteEntry
    // Internal structure definition, visible only here
    {
```

```
        byte red;
        byte green;
        byte blue;
    };

    byte bit_size;  // the number of colors is 2 ** bit_size
    ColorPaletteEntry palette[1 << bit_size];  // the array

    valid
    {
        (bit_size >= 1) && (bit_size <= 8)
    }

    recover
    {
        if (bit_size < 1)
            bit_size = 1;
        else if (bit_size > 8)
            bit_size = 8;
    }
}
```

4.2 Reasoning About Specification Correctness

While it is possible to encounter data corruption, it is also possible to have incorrect specifications. The specification correctness can be checked using automated verification tools like Spin, Larch, PVS, etc. (see [2]).

Unfortunately, there is currently no way to completely automate this operation. However, with human assistance and with some guidance about how to translate from our language into an automated verification tool, some results can be achieved. This step is essential when designing complex serialization protocols.

We need to know what sort of questions can we ask the theorem prover. Here are some sample questions:

- The very first and the most important question about the correctness of a protocol definition is: given the writing methods, do I retrieve the same information when re-reading it?
- Given the reading methods, is it possible for the validation methods to be trivially true? In that case, they can be safely removed (this cannot always be detected by the compiler).
- Given the reading methods, is it possible for the validation methods to be trivially false? In that case, there is something wrong in the specification (this cannot always be detected by the compiler).

- Given the reading and the recovering methods, is valid information always obtained when reading a stream, even if it is damaged?
- Is the protocol extensible? Can a superset of that protocol be defined? Given two protocols, we can say that the first protocol is a superset of the second protocol iff it can encapsulate more information, and, at the same time, any stream that is compliant with the second protocol is also compliant with the first protocol.
- Given two protocols, can it be inferred from their definitions that they are equivalent? Two protocols are equivalent if and only if every stream that is compliant with one protocol is also compliant with the other protocol.

Please note that the question "Do I rewrite the same thing that was previously written?" is not a goal, as some protocol definitions can allow two streams with different contents to represent the same information.

4.2.1 Translating Formal Protocol Descriptions into the PVS Theorem Prover Language

PVS [4], the *Prototype Verification System*, is an interactive environment for writing formal specifications and for checking formal proofs. It has a very expressive specification language and powerful theorem-proving capabilities.

The research that is currently done regarding the verification of protocol specifications is mainly focused on how to translate certain patterns of serialization protocols into their equivalent PVS definitions. After translation of some real examples and proving their correctness, we intend to look for proving patterns, in order to get to a completely mechanical proof, at least, for a part of such protocols.

A trivial protocol having a simple structure with 3 fields of the same type, default reading and writing methods and a validation method, can be translated into PVS in the following way:

```
test: THEORY

BEGIN

  anything: TYPE

  % This is the logical file structure.
  % It is a generic structure that contains 3 fields
  %  of the same type.
  struct: TYPE = [anything, anything, anything]

  % This is the file stream.
  % It is regarded as a mapping from integer offsets
  %  to logical structures.
```

```
stream: TYPE = [nat -> struct]

% An uninterpreted constant which stands for 'nothing'.
nothing: struct

% An empty stream contains nothing.
emptystream: stream
empty_ax: AXIOM
  FORALL (n: nat): emptystream(n) = nothing

% This is the formal specification of the reading method.
% For any stream, for any offset, the object located
%  in that stream, at that offset, is equal to the object
%  that is contained by the stream at that offset.
read: [stream, nat -> struct]
read_ax: AXIOM
  FORALL (s: stream):
    FORALL (offset: nat):
      read(s, offset) = s(offset)

% This is the formal specification of the writing method.
% For any stream, for any offset, for any object,
%  the stream which results after writing the object
%  at the offset is equal to the stream containing
%  the object at the same offset.
write: [stream, nat, struct -> stream]
write_ax: AXIOM
  FORALL (s: stream), (obj: struct):
    FORALL (offset: nat):
      write(s, offset, obj) = s WITH [(offset) := obj]

% The validation method is a boolean function
valid?: [struct -> boolean]
valid: (valid?)

% The first theorem asserts that reading after writing
%  an object yields an identical object.
read_write: THEOREM
  FORALL (s: stream), (obj: struct):
    FORALL (offset: nat):
      read(write(s, offset, obj), offset) = obj

% The second theorem asserts that if a valid object has been
%  written then a valid object will be read.
read_write_valid: THEOREM
  FORALL (s: stream):
```

```
    FORALL (offset: nat):
      read(write(s, offset, valid), offset) = valid

END test
```

The theorems ensure that the correctness condition holds at every offset in the file, enabling *compositional reasoning*. Compositional reasoning allows to prove correctness of a system by proving correctness of every component of that system [12]. We apply the compositional reasoning in this case by saying that a serialization protocol is correct if and only if the contents of any file produced by that protocol is correct at every point (i.e. offset).

To verify the theorems, after invoking the prover, it is enough to use the following prover command:

```
(grind :theories ("test"))
```

This is the highest-level strategy, which automatically does skolemization, heuristic instantiation, propositional simplification (using BDDs), if-lifting (conversion of some expressions into their IF-THEN-ELSE equivalent), rewriting, and applies decision procedures for linear arithmetic and equality ([4]).

4.2.2 Checking the Specification of the Validation

Our last sample protocol showed how to describe constraints for the validity of the stream. Are those constraints well-written? Let's translate those constraints into PVS:

```
valid_palette: THEORY

BEGIN

  % The value to be validated.
  bit_size: VAR upto[255]

  % The value to be recovered.
  bad_bit_size: VAR upto[255]

  % Ensure that the validation condition can be met.
  valid_can_be_true: THEOREM
    EXISTS(bit_size): (bit_size >= 1) AND (bit_size <= 8)

  % Ensure that the validation condition is not vacuously true.
  valid_can_be_false: THEOREM
    EXISTS(bad_bit_size):
      NOT ((bad_bit_size >= 1) AND (bad_bit_size <= 8))

  % See if the recovery method produces a valid result.
```

```
  good_recovery: THEOREM
    (IF bad_bit_size < 1 THEN bit_size = 1 ELSE
      (IF bad_bit_size > 8 THEN bit_size = 8 ELSE
        bit_size = bad_bit_size ENDIF)
      ENDIF)
    IMPLIES
      (bit_size >= 1) AND (bit_size <= 8)

END valid_palette
```

We can try to see if the theorem prover handles does the work entirely by itself:

```
(grind)
```

Thus it works for `valid_can_be_true` and `good_recovery`, but not for `valid_can_be_false`. We will try to do the proof for the latter theorem by ourselves, indicating which steps to take.

Since the proofs of the first two theorems are very similar and very easy, we will consider both. Since we have to prove the existence of some values that satisfy an obvious arithmetical relation, it is enough to find concrete values. PVS will do the rest, which is, to prove that the chosen values really satisfy the property requested by the theorem.

In the first case, in the theorem `valid_can_be_true`, `bit_size` can have any value between 1 and 8. Therefore, to prove the existence, we instantiate the variable with such a value:

```
(inst 1 "1")
(assert)
```

The meaning of this proof is the following: instantiate the first variable from the theorem (which happens to be the only variable — `bit_size`) with 1. After the instantiation, the proof sequent is

```
valid_can_be_true :

% no antecedent
  |-------
{1}   (1 >= 1) AND (1 <= 8)
```

which is further proved with the simple `(assert)` command.

The proof for the theorem `valid_can_be_false` is similar. The only difference resides in the need to find a natural number that is less than 255, and it is not between 1 and 8. So either 0, or values between 9 and 255 are acceptable:

```
(inst 1 "9")
(assert)
```

After instantiation, the proof sequent is

```
valid_can_be_false :

{-1}  ((9 >= 1) AND (9 <= 8))
  |-------
% no succedent
```

The `(assert)` command ends the proof.

4.2.3 Checking the Specification of Heterogeneous Collections

Consider a heterogenous collection of internal structures: the JPEG file. A correct JPEG decoder must correctly identify the type of every JPEG structure (block) present in a JPEG file. Thus, no two signature identification algorithms accept the same structure type.

We will not try to verify the JPEG algorithms here, but will consider a much simpler case instead: there are structures of *type*1 which are identified by the leading byte which is 1, there are structures of *type*2 which are identified by the leading byte which is 2, and there are structures of *unknowntype* which are identified by the leading byte which is neither 1 or 2. The theorems `no_confusion1`, `no_confusion2` and `no_confusionN` stand for the verification of each type. The proof of correctness requires completing the proofs of all of these theorems. As these proofs are similar we will consider only the first type.

The presence of the unknown type is essential to enable adding further features to the file format, without breaking the backwards compatibility. The JPEG format has the same property. Therefore, it is possible for the upcoming JPEG2000 Standard [9] to extend the existing JPEG file structure. The extensibility of our little format is checked here by the `extensibility` theorem.

```
signatures: THEORY

BEGIN

  % The leading value, which will further be restricted
  %  to the domain 0..255 (byte)
  signature: TYPE = nat

  % The type contained by our structure
  anything: TYPE

  % This is the logical file structure.
  % It is a generic structure that contains 2 fields
  %  of the same type, along the signature.
  struct: TYPE = [signature, anything, anything]

  % These boolean mappings identify the type of an object.
```

```
  type1?: [struct->boolean]
  type2?: [struct->boolean]
  typeN?: [struct->boolean]

  % Define the actual mappings.
  type1_ax: AXIOM
    FORALL (v: (type1?)):
      v`1 = 1
  type2_ax: AXIOM
    FORALL (v: (type2?)):
      v`1 = 2
  typeN_ax: AXIOM
    FORALL (v: (typeN?)):
      (v`1 /= 1) AND (v`1 /= 2) AND (v`1 <= 255)

  % This assures that no object of type1 will be misregarded as
  %  being of another type.
  no_confusion1: THEOREM
    FORALL (v1: (type1?), v2: (type2?), vN: (typeN?)):
      (v1`1 /= v2`1) AND (v1`1 /= vN`1)

  % Continue doing the same kind of verification
  %  with the rest of the types.
  % no_confusion2: THEOREM...
  % no_confusionN: THEOREM...

  % Check the extensibility
  extensibility: THEOREM
    EXIST (v: (typeN?))

END signatures
```

The proof for `no_confusion1` is presented below. Each step of the proof is explained in lines that start with `%`:

```
(grind)

% (v1`1 /= v2`1) becomes NOT (v1`1 = v2`1)
% (v1`1 /= vN`1) becomes NOT (v1`1 = vN`1)
% trying repeated skolemization, instantiation and if-lifting
% this yields 2 subgoals:

no_confusion1.1:

{-1}  type1?(v1!1)
{-2}  type2?(v2!1)
```

```
{-3}  typeN?(vN!1)
{-4}  (v1!1`1 = v2!1`1)
  |-------
% no succedent

no_confusion1.2 :

{-1}  type1?(v1!1)
{-2}  type2?(v2!1)
{-3}  typeN?(vN!1)
{-4}  (v1!1`1 = vN!1`1)
  |-------
% no succedent

% prove "no_confusion1.1" first
% apply the axioms "type1_ax" and "type2_ax":

(use "type1_ax")
(use "type2_ax")

% the current goal becomes:

no_confusion1.1 :

{-1}  v2!1`1 = 2
[-2]  v1!1`1 = 1
[-3]  type1?(v1!1)
[-4]  type2?(v2!1)
[-5]  typeN?(vN!1)
[-6]  (v1!1`1 = v2!1`1)
  |-------
% no succedent

% in this moment we have enough information to say it is valid:

(assert)

% simplifying, rewriting, and recording with decision procedures
% this completes the proof of "no_confusion1.1"
% the proof of "no_confusion1.2" is similar
% apply the axioms "type1_ax" and "typeN_ax":

(use "type1_ax")
(use "typeN_ax")

% the current goal becomes:
```

```
no_confusion1.2 :

{-1}  (vN!1`1 /= 1) AND (vN!1`1 /= 2) AND (vN!1`1 <= 255)
[-2]  v1!1`1 = 1
[-3]  type1?(v1!1)
[-4]  type2?(v2!1)
[-5]  typeN?(vN!1)
[-6]  (v1!1`1 = vN!1`1)
  |-------
% no succedent

% in this moment we have enough information to say it is valid:

(assert)

% simplifying, rewriting, and recording with decision procedures
% this completes the proof of "no_confusion1.2"

Q.E.D.
```

We noticed that the axiom `type1_ax` is applied twice (once for each subgoal `no_confusion1.1` and `no_confusion1.2`). We can remove one extra step of the proof if we will apply this axiom before splitting the goal, i.e. before issuing the `(grind)` command.

5 Discussion

5.1 Current Limitations

An important limitation of our language is its inability to interact with the rest of the application while reading or writing information from the streams. This is not a problem when the information does not take much space, because it is easy to serialize all objects at once. But in situations like progressively displaying an image as it comes from the Internet, or instantly compressing or decompressing some information in a stream, a different strategy is needed.

Thinking about the general ability of this method of treating various serialization protocols, every stream can be encoded once that there is a way to represent various data types (both predefined and user-defined). We do not attempt to specify non-streamable file structures, such as any database that requires random file access and that cannot reveal or store all important information without performing file seek operations. In general, modern file structures are streamable, especially if they are intended to be used when transmitting information via the Internet.

5.2 Related Work

5.2.1 ASN.1

ASN.1 (ISO/IEC-8824) [13] is the acronym for *Abstract Syntax Notation One*, a language for describing structured information.

The first example presented in this paper can be modeled using ASN.1 in the following way:

```
i ::= INTEGER
s ::= OCTETSTRING
```

The difference between ASN.1 and our approach is that it can only enumerate the data types, in a manner similar to our enumeration of fields within structures. It does not provide a way to specify custom encoding, and its default encoding is not efficient in some situations.

ASN.1 is widely used for describing the lowest-level communication protocols, almost strictly related to the physical level. It is not suitable for higher-level applications.

5.2.2 SGML and XML

SGML, the *Standard Generalized Markup Language* (ISO 8879) is the international standard for defining descriptions of the structure and content of different types of electronic documents. *XML*, the *Extensible Markup Language* [18], is a markup language designed to enable the use of SGML on the World Wide Web. More precisely, it is a *metalanguage* — a language for describing other languages — which lets anyone design his or her own markup. It is an abbreviated version of SGML, that omits the more complex and less-used parts of SGML in return for the ease of use and understanding and for being more suitable for delivery and interoperability over the Web.

In the case of the previous example, the document type contains one integer field named i, and one string field named s. A concrete document to store the integer 123 and the string *"Hello, world!"* is presented here:

```
<?xml version="1.0" standalone="yes"?>
<SimpleStruct>
    <i>123</i>
    <s>Hello, world!</s>
</SimpleStruct>
```

As in our case, the document architecture is described via a special construct. Unlike our approach, the explicit description of this architecture can be missing, as it can be inferred from the contents of the actual, concrete document.

XML is becoming widely used for storing and exchange data. Its limitations are its high syntactic redundancy and its possibility to store only textual information, which don't make it suitable for encoding large binary streams.

5.2.3 Domain-Specific Languages

While ASN.1 and XML, presented above, address a large variety of problems, there are stream description methods that address very specific problems. A good example is constituted by *XDR (External Data Representation)* [14], used in protocols such as *RPC (Remote Procedure Calls)* and *NFS (Network File System)*. It is an external data representation for numbers stored in a file, so that applications can focus on the logical rather than physical structure.

ODA (Office Document Architecture) [11] is an ISO standard (8613) for representing documents containing multifont text, raster images and geometric graphics, an encoding format for use in multimedia mail and file exchange. It was designed mainly by a group from Carnegie Mellon University (Information Technology Center), and a group from the University of Michigan (Center for Information Technology Integration).

MSDL (MPEG Syntactic Description Language) [8], designed at Columbia University, is intended for describing the syntax of MPEG streams. It provides the complete syntax and semantics of the description language.

The conclusion is, even if all these are different, they all target the same issue. As an observation, none of these approaches provides support for data validation and data recovery. That's why we believe that a wider, unified view, to encapsulate all of them, would be useful.

5.3 Advantages of Our Approach

The method that we described is supposed to enable the writing of formal specifications of serialization protocols. These specifications are primarily intended to accompany the informal specifications that describe the same protocols, to increase their understanding, and to make possible the automatic creation of executable code to implement those protocols. However, there are additional advantages of formalizing streams.

- *A way to communicate between people*
  Formal descriptions can accompany the verbose, informal descriptions of serialization protocols. They can increase the level of understanding, because one is able to see, at a glance, important information about the protocol architecture.

- *Automatic generation of validation tools*
  One example of such tool is the Java validator, that takes as input a Java-compiled class (*bytecode*), and decides whether that compiled code is compliant with the Sun's specification *100% Pure Java* [15]. Another example is the HTML validation service, provided by the World Wide Web Consortium. This service can tell the same thing about any HTML file, if it is valid according to the HTML file specification. Such validation tools can be constructed automatically from the reading and the validation methods present in the formal specification.

- *Easy access to binary-encoded information*
  Once a formal description is obtained, it is easy to build a program to convert back and forth from the original binary encoding, to a textual, e.g. XML-like representation. The validation and the recovering methods play a special role after the text has been modified by a human: they ensure that the modification does not affect the validity of the information.

  There are some tools that allow humans to edit information encoded in binary, by having binary-to-text and text-to-binary converters. However, they are only a few, and each of them has its own textual description language. They all address the same problem which is generalized in this article. By having a conversion tool to support our formal language, there is a possibility to have textual description of any binary file, provided there is a formal description of that binary file format in some repository.

6 Conclusion

Can formal methods ever improve productivity as much as compilers? asked Danny House in his paper [6]. His answer followed immediately: *We claim that they can and will.* The material presented in this article is a concrete example in which the usage of a method to specify and then encode serialization protocols is very similar, if not identical, to the classic usage of a compiler. One might claim that there is nothing new in here; it is just the translation from one programming language into another programming language, so it is just the usual compilation. However, the source language is not a programming language, but a description one. It cannot be used to write programs, but to describe, in a formal way, different serialization and communication standards that used to be described and that are still being described only informally, in natural language. The formal description of such a standard is not meant to replace its informal description, but to accompany it.

However, it can replace the reference library that the authoring group used to provide. First, the effort needed to write the formal specification is significantly less than the effort spent to write the standard library. Second, code in any imperative programming language (including the language used by the former library reference) can be automatically generated, directly from the specification. But to be able to generate code at the same flexibility that is offered by today's reference implementations, more research must be done, as currently there is no possibility to interact between the generated code and the application code, during the serialization process.

There is also some effort being spent on translating formal protocol specifications into languages accepted by theorem provers, in order to do reasoning about specification correctness.

The method presented throughout this article is meant to produce executable code directly from abstract specifications, thus entering the small, but hopefully growing group of such methods, currently formed by tools like LEX and YACC.

References

[1] Thomas Boutell et. al., *"PNG (Portable Network Graphics Format) Version 1.0"*, RFC 2083, October 1996. This specification has also been published as a W3C Recommendation, which is available at http://www.w3.org/TR/REC-png.html

[2] Marsha Chechik, *"Automated Verification"*, course CSC2108, University of Toronto, 1999.

[3] CompuServe Inc., *"Graphic Interchange Format (sm), Version 89a"*, 1990.

[4] J. Crow et al., *"A Tutorial Introduction to PVS"*, presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Also available at http://www.csl.sri.com/sri-csl-fm.html

[5] M. J. Folk, B. Zoellick, G. Riccardi, *"File Structures: An Object-Oriented Approach with C++"*, Addison-Wesley, 1998.

[6] Danny House, *"Formal Methods: More Than Just Proofs"*, DCS Technical Report 374/1998, University of Toronto.

[7] ISO/IEC, *"Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines"*, ISO/IEC IS 10918-1, ITU-T T.81, September 1992.

[8] ISO/IEC, *"Coding of Moving Pictures and Associated Audio"*, ISO/IEC JTC1/SC29/WG11, March 1996.

[9] ISO/IEC, *"Information Technology - JPEG 2000 Image Coding System"*, ISO/IEC FCD, March 2000.

[10] Object Management Group, *"Unified Modeling Language - Object Constraint Language (OCL) Specification, Version 1.1"*, January 1997. For details, see the OMG home page at http://www.omg.org

[11] Jonathan Rosenberg et al., *"Multimedia and Document Translation: ODA and the Express Project"*, Springer-Verlag, 1991.

[12] Oleg Sheyner, Jeanette Wing, *"Composing Proofs of Security Protocols Using Isabelle/IOA"*, February 2000.

[13] Doug Steedman, Technology Appraisals Ltd., *"Abstract Syntax Notation One (ASN.1) — The Tutorial and Reference"*, 1990. Also available at http://www.techapps.co.uk/asn1gloss.html

[14] Sun Microsystems, Inc., *"XDR: External Data Representation Standard"*, RFC 1014, June 1987.

[15] Sun Microsystems, Inc., *"100% Pure Java Certification Guide"*, Revision 3.0, October 1999.

[16] Cosmin Truta, *"OOP — Object Persistence"*, MSc. Thesis, Babes-Bolyai University, Cluj-Napoca, Romania, 1999.

[17] W. M. Turski, T.S.E. Maibaum, *"The Specification of Computer Programs"*, Addison-Wesley, 1987.

[18] World-Wide Web Consortium (MIT, INRIA, Keio University), *"Extensible Markup Language (XML) Version 1.0"*, February 1998. This specification is a W3C Recommendation, available at http://www.w3.org/TR/1998/REC-xml-19980210

A The Language Syntax

Here is the grammar for our language in EBNF syntax. This grammar is ambiguous, and only the important parts are presented, so it is not complete. It is provided only to increase the understanding, and not to be used for implementation.

```
start ::= (default_decl)*  (struct_def)*

identifier ::= <## a C-like identifier ##>

expression ::= <## a C-like (C++-like) expression ##>

default_decl ::= "default"
    ("signed" | "unsigned" | "text" |
    "little_endian" | "big_endian") ";"

struct_def ::= "struct" identifier ("(" struct_arg ")")?
    "{" struct_body "}" (";")?

struct_arg ::= (type_decl identifier) |
    (struct_arg "," type_decl identifier)

struct_body ::= (struct_signature | struct_field | struct_method |
    struct_def)*

struct_signature ::= (type_decl "signature" "{" method_body "}") |
    ("const" type_decl "signature" "=" expression ";")

struct_field ::= (struct_field_decl ";") |
    ("const" struct_field_decl "=" expression ";")
```

```
struct_method ::= method_name "{" method_body "}"

struct_field_decl ::= type_decl | (type_decl identifier)

type_decl ::= identifier | (identifier "[" expression "]") |
    collection_type

collection_type ::= ("set" | "bag" | "sequence") "<" type_decl ">"

method_name ::= "read" | "write" | "valid" | "recover"

method_body ::= (instr)*

instr ::= expression ";" |
    ("{" expression "}") |
    ("if" "(" expression ")" instr ("else" instr)?) |
    ("for" "(" (expression | "") ";" (expression | "") ";"
        (expression | "") ")" instr) |
    ("while" "(" expression ")" instr) |
    ("do" instr "while" "(" expression ")" ";") |
    <## other C instructions, possibly necessary or useful ##>
```