Data description language FlexT : flexible types for description of static data.

ALEXEI HMELNOV, STANISLAV VASSILYEV
Institute of System Dynamics and Control Theory
Russian Academy of Sciences
134 Lermontov St, Irkutsk, 664033
RUSSIA

Abstract: - The problem of binary data format specification is considered. This problem is further limited to simpler subproblems. First, the concept of *specification of data interpretation* as opposed to *specification of data modification* is introduced, i.e. we shall concentrate on the interpretation of already existing static data, and will not take into account different aspects of memory allocation, correct sequence of data elements and so on. Next, we shall consider the case of *data type identification*, which is a special and most basic kind of data interpretation. Specification of the data type identification for some data format enables to identify every bit of data of this format as a part of representation of some data element of definite type.

The language of **Flex**ible **T**ypes **FlexT** was developed for description of the data type identification. It uses rather simple extensions of the set of type constructors traditional to common procedural languages. These extensions are very efficient for the problem under consideration.

On the one hand, the use of specification when developing a program enables to increase the safety of the program due to increase of the level of abstraction of the specified program part. On the other hand, the specification of format of some data enables to increase the reliability and safety of processing of this data, because it makes them transparent. I.e. specification of data format enables to consider the data not as a "black box" - a sequence of bytes of unknown purpose, but as a collection of data elements of definite types, which makes it possible to control the contents of the data. We can also consider a machine code as a specific type of data and describe file formats like that of *EXE* and *OBJ* files.

Key-Words: - binary format, formal specification, data format specification, specification of data interpretation, specification of data modification, data type identification. *CSCC'99 Proceedings*, Pages:1371-1376

1 Introduction

During the past few decades, which constitute the history of computers, vast amount of information was accumulated in electronic form, and to store and transfer these data almost as many data representation formats was developed. The number of formats used for representation of data of some class, e.g. raster images, may run into dozens or even hundreds.

The information about data formats can be represented by natural language descriptions, which very often contain errors or ambiguities. It can also be encoded into some libraries or conversion programs designed for some definite way of data processing which, as a rule, differs from the required one. All the information about the format is presented in its description or in the source code of the library but in implicit form: in phrases of natural language or in statements of the program written in some definite programming language *for some definite way of data processing.*

So the programmer often has to write his own code for data input/output. The most essential disadvantage of this approach is not that it requires repetition of work already performed many times before, but that it may cause programming errors as a result of inattention or misunderstanding of specification.

How can this situation be improved?

1.1 Universal exchange formats

In the fields, where some traditional set of universal enough exchange formats has not been established yet, main efforts are focused on the development of such a format. An example of such a field is Geographic Information Systems (GIS) and Spatial Data Transfer Standard (SDTS). It was supposed that all programs were to support the universal exchange format, which would solve the problem of data

exchange. The example of SDTS demonstrates that such a universal format will necessarily be very complicated, because it should support all the capabilities of all the other existing formats. Thus it will be rather easy to store data into this format, because it will require to use only limited subset of its capabilities corresponding to the capabilities implemented in the program under development, but it will be much more difficult to read data from this format, because the data could be written according to capabilities of another program, which could have no correspondence in the program under consideration.

Another much more serious complication is that it is impossible to predict in advance the evolution of the considered field and it would be necessary to revise in the future even the most universal by now format. Yet another essential disadvantage of universal exchange formats is that they are as a rule very cumbersome and ineffective in comparison with specialized ones and generate too large files. A good illustration of this statement is PostScript format.

1.2 Self-documented formats

To solve the problem of extending capabilities of exchange format this approach suggests to include into data file all needed for its interpretation metainformation. Thus to read such a file a program should be able to use this metainformation for interpreting, i.e. loading into internal representation, that part of data for processing of which it was designed. For example this approach was offered for exchange of results of physical experiments.

The disadvantages of self-documented formats are that it requires converters to transfer already existing files into this formats and also that the metainformation is duplicated in each self-documented file even if the data in these files are of the same kind.

Fundamentally, this approach can be considered as a special subclass of universal exchange formats. For example the already mentioned PostScript format may be placed into this subclass: it contains the header with definitions of functions and fonts, which are used for description of the text image itself, with the header occupying considerable and, sometimes, most part of the file and this header been duplicated with no change over all the files created by the same text editor.

1.3 Data description languages

To get rid of the above-mentioned disadvantages of the files containing metainformation it would be quite natural to isolate this metainformation into separate file. In this case to support yet unknown and may be existing long before format it would be enough to supply the file describing this format as a specimen of the data of the class for processing which the program was designed.

The problem here is how to develop such a data specification language, which can support description of some class of file formats. It is desirable to make this class wide enough, so some specific program would be able to use its specific subset. But it should be taken into account that the more universal will be this language the more complicated it will be. Good balance here is necessary for this language to be a specification language and not just another programming language.

2 Specification languages

According to their data processing capabilities we can distinguish two levels of data specification: *specification of data interpretation* and *specification of data modification*.

2.1 The specification of data interpretation

A language for specification of data interpretation should allow to describe *easily* the *interpretation* of arbitrary within some limits data class.

The term *interpretation* is not considered here to mean just a conversion of all data into some specific format, but it denotes the definition of observer functions for extraction from the described data the values of attributes of the data class to which the data belong according to this specification. For example, a specification of a raster image file format should enable to extract from such a file information about the number of images in the file, the dimensions of these images (width and height) and about the colour of each pixel. To increase efficiency of this specification it can be formulated in terms of some less abstract class, for which already exists a specification relating this subclass to the abstract class of interest.

2.2 The specification of data modification

A language for specification of data modification should allow to define not only observer functions but also constructor functions, which specify how to create new data elements with definite attributes. In this case we have to consider additional details, such as memory allocation, order of data element generation, data elements alignment and so on.

Ideally, the specification of data modification should provide all the necessary information to

transfer data into any other already described format, including that of the program internal representation. The specification language itself should not be very time efficient. To increase efficiency automatic code generation according to specifications of the source and the target formats could be used. Thus to read data of some format into internal representation a special reading procedure could be generated which optimizes data transfer by avoiding generation of intermediate representation.

This level of specification languages will not be considered subsequently - it was mentioned here as a possible direction of further research in the field of data format specification.

2.3 Data type identification as a basic data type interpretation

As a basic interpretation which can be used for description of any data format we suggest such an interpretation, which assign type to each data element - i.e. which performs the *data type identification*. Such an interpretation is a basic one and it constitutes the lowest level of the hierarchy of interpretations, because, when using some data element in some interpretation, it will be necessary to specify the data type of this element.

The class of data to which the raw data are mapped by the interpretation of data type identification is a set of the interrelated *data elements*. Each data element is characterized by its position (*address* and *size*) and its *type*. Data element can contain references (pointers) to other data elements, this property makes it unnatural to use for specification some kind of BNF extension. The size and the type of data element are interrelated: one of these properties defines the other, the direction of this relation depends on the data element. The data elements can be complex, in this case it is possible to distinguish inside of this data element data elements of smaller size. The data type identification can be *incomplete*. Such an interpretation contains data elements of indefinite type, or we can consider such data elements as belonging to special type - *raw data*.

3 The suggested approach

This paper considers the data specification language **FlexT** (**Flex**ible **T**ypes), which enables to describe the interpretation of static data, primarily in the form of data type identification. This language is used for data description in the disassembler/viewer of various file formats **BinView**, and also in the disassembler of the program **Portable EXE Explorer** for viewing the contents of the 32-bit executable files of **Windows 95/NT** (the **Portable Executable** format).

By *static data types* we will understand the data types, which are analogous to traditional ones of procedural programming languages. The distinctive feature of these data types is that the size of data element of such a type and the internal positions of its constituent parts are defined on the stage of compiling and don't depend on specific data. In procedural programming languages, at least in those, which are really in use at the present time, the composite data types can contain only static parts.

The size of data element of *dynamic type* and the positions of its constituents may depend on specific data. We'll use subsequently the term *"dynamic"* to denote dependence on the data element position and contents. Examples of dynamic data type in the traditional procedural languages are string constants both **Pascal** and **ASCIIZ**: to determine the size of such a constant it is required to check the first byte for **Pascal** strings and to find the last zero byte for **ASCIIZ** strings.

It is apparent that dynamic modifiable data types [1] can not be used as types of variables, because assignment of new value to such a variable or to some of its parts may cause change of the size of this variable. For the fields of records of variable size in traditional languages such as strings or variant records the memory is allocated to maximum possible extent. Meanwhile, as to static data, which are read only for the program, for their encoding can be used very sophisticated techniques, for example, using assembler macros.

An example of static data in code is the Run Time Type Information (RTTI) of **Delphi** programs:

```
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
  Kind: TTypeKind;
  Name: ShortString;
 {TypeData: TTypeData}
end;
PTypeData = ^TTypeData;
TTypeData = packed record
  case TTypeKind of
```

(Excerpt from ***TypeInfo.pas*** [2]).

The comment in the record *TTypeInfo* denotes that the field *TypeData* is placed immediately after the last symbol of the string field *Name*, the size of which depends on the length of the name of specific type. Because the offset of the field *TypeData* depends on specific data - the name of the type, this structure can not be immediately represented in **Pascal** - it is required to write a special code to access the field (excerpt from the same file):

```
function GetTypeData(TypeInfo:
 PTypeInfo):PTypeData;assembler;
asm
  {-> EAX Pointer to type info}
  {<- EAX Pointer to type data}
  {it's really just to skip the kind and the name}
  XOR EDX,EDX
  MOV DL,[EAX].TTypeInfo.Name.Byte[0]
  LEA EAX,[EAX].TTypeInfo.Name[EDX+1]
end;
```

The same difficulties are experienced by all the authors of data format descriptions when trying to describe some kind of dynamic data element using for instance the **C** language.

The reason for all this problems is that *for specification of data formats were used languages designed for specification of types of variables*. But if we'll not limit the constructors of data types by the constraints for the types of variables, it is possible to support in a natural manner description of more sophisticated data encoding schemes, for example, like when one field of record contains the number of elements of array which is another field of the record.

4 Extension of the set of data type constructors

It is possible to split any physical data into compact data blocks. Each such a block is characterized by its address, size and interpretation. A compound compact block can be split into smaller compact blocks. The main ways of combining data items can be expressed by three basic type constructors of compound data types: record, array and variant.

Let us consider now in greater detail the suggested extensions of the type definition technique.

1) Components of variable size

The language supports record fields and array elements of variable size.

2) Expressions, properties and parameters of types

Data types are characterized by a set of parameters, for instance, the size and the number of elements for array or the value of the case selector for variant. The values of parameters can be specified by expressions, which bind these values to the values of other fields of complex data type and/or its parameters.

3) The block of statements

Every data type definition, except the type call, can be followed by several blocks with additional information. The beginning of each additional block is marked by the '**:**' sign. Among them the block of statements contains in square brackets a series of comma separated statements about the values of the properties of the type and of the parameters of its constituents, which were not specified in the constructors' calls.

4) The type calls

The type calls with substitution of actual parameters in place of formal parameters is also considered as a special type constructor. In the type call the expressions of the actual parameters can be related to the formal ones by position or by name, it is similar to the calling convention for the methods of **COM** interfaces. The parameter passing by name is used when, for example, it depends on the value of the field, which was not defined by the moment of reading the type call.

5) Record

It consists of a fixed number of fields. It can contain fields of variable size, with the position of the next field been determined by the position and the size of the previous one.

6) Variant

The type of this element is determined by the value of its parameter - the selected case. By now we support integer and string selector values.

7) Array

It consists of variable number of elements of same type. The size of array can be limited either by specifying the number of its elements or by specifying the size itself or by stop condition, which should be satisfied for the last array element (as for ASCIIZ string).

8) Abstract data types.
A data element can serve as a representation of some more abstract data type [1]. A simple example of this case is the type *"index"* of **OBJ**-files [3]:

```
if (first_byte & 0x80)
  index_word=(first_byte & 7F)*
    0x100 + second_byte;
else
  index_word=first_byte;
```

I.e. to save space small (<0x80) numbers are encoded by one byte and larger numbers are represented by two bytes. This data type can be easily described by the record with variant field, but for the rest of the **OBJ** - file specification these low-level details of index representation are of no interest, and it would be better to be able to represent all the elements of this data type by the number which is encoded in it.

Abstract data types can also be used for specification of data interpretations of higher than just a type identification levels, but this problem is beyond the scope of this article.

9) Address spaces, address blocks and pointers.
Some parts of compact blocks can represent pointers - references to other data elements. In the simplest case the value of such a pointer is an offset from the beginning of the file or some linear function of this value. But, generally, it represents a *virtual address* in some *virtual address space*. An address space can contain one or more *address blocks*. Each address block is characterized by two address mappings: physical and virtual. The address space of lowest level is that of the data file itself. A good example of format with virtual address space is 32-bit executable of **Windows 95/NT** [4].

When declaring pointer data type the address space of this pointer and, in general, the expression for calculating the referred address by the value of the pointer representation type can be specified.

10) Machine instructions.
In the above-mentioned disassembler programs a special type of pointer - the pointer to machine code is implemented. For the present for specification of machine instructions encoding we use a separate language of binary unifications, but the approach of dynamic data types can also be immediately used for this purpose. We can confirm this assertion by successful specification of the **Java** virtual machine class file format [5] up to the machine instruction encoding level using only **FlexT** statements. In comparison with the widely cited approach of [6], we can consider machine instruction representation as yet another data type and describe it using the designed for arbitrary data techniques.

5 Limitations of the suggested approach.

Even with advanced technique of abstract data types specification the situations are possible anyway where pure declarative approach will not suffice. It can happen when the length of the formula, which describes the dependence between data elements or between data element and its interpretation, can grow unlimitedly. For example, it applies to some kind of compressed data, because to describe the encoding/decoding of this kind of data it is necessary to describe the compression/decompression algorithms, which use rather sophisticated data structures to represent intermediate information. Another example is a block of machine code: to completely separate code and data it could be necessary to simulate execution of machine instructions.

But even for this cases we can use the technique of dynamic data types to some limited extent. We can describe the rest of the data and leave those parts, which defy description in the state of raw data. As to the code description the simplest approach, which we apply in disassemblers, is to enable manual specification of additional entry and stop points.

6 Conclusion

This article describes the main concepts of the language **FlexT** for specification of data interpretation, which enables to describe wide range of binary data formats using simple extensions of the set of type constructors of traditional procedural languages. Using **FlexT** different binary formats were completely or partially specified, among which are EXE (DOS, NE, LE, LX, PE), ELF, TPU, OBJ, CLA, DBF, DB, BMP, ICO, CUR, ANI, WAV, DVI and so on. It is also possible to use this language for specification of machine instructions encoding. The current version of specification interpreter uses them for identification of data types. Further we are going to extend its capabilities using the Horn subset of the first order logical language of positively constructed formulas [7,8] for description of other possible interpretations. We also plan to implement specifications of modification and for the purposes of code analysis extend the language by elements of machine architecture and machine instruction semantics specification. Another projected

application of **FlexT** specifications is automatic generation of data reading/writing code for different programming languages.

Appendix: - **An Example of Specification**

1) Specification of the format of DBF file **(*dbf.rfh*)**

```
type
  TBinDate array[3] of num-(1)
  TDBF3FldKind enum Char (
    fkChar='C', fkNumeric = 'N',
    fkLog = 'L', fkDate = 'D',
    fkMemo = 'M')
  TDBF3FldDsc struc
    array[11] of Char Name
    TDBF3FldKind hType
    ulong DataP //like Delphi Tag
    Byte Len
    Byte DecNum
    Word MUsrRsrv1
    Byte WorkID
    Word MUsrRsrv2
    Byte SetFldData
    raw[8] Reserved
  ends
  PDataArray ^TDataArray near
  TDBF3Hdr struc
    Byte Ver
    TBinDate LastChangeDate
    ulong RecCnt
    PDataArray HdrLen
    Word RecLen
    raw[20] Reserved
  ends
  TDBF3HdrWithFields struc
    TDBF3Hdr H
    array[(@.H.HdrLen-@.H:Size-1)
      div 32] of TDBF3FldDsc Fields
  ends
data
0x0000 TDBF3HdrWithFields Hdr
type
  TFieldData array[Hdr.Fields[#].
    Len]of Char
  TFieldsData array of TFieldData:
    [@:Size=Hdr.H.RecLen-1]
  TRecData struc
    Char F
    TFieldsData D
  ends
  TDataArray array[Hdr.H.RecCnt] of
    TRecData
```

2) Excerpt from the data type identification result file for a simple table with 2 fields and 3 records.

```
0000:Hdr: TDBF3HdrWithFields = (
  H:(Ver:03;
    LastChangeDate: (0:99,1:3,
      2:3); RecCnt:00000003;
    HdrLen:0061; RecLen:0008;
    Reserved: …);
  Fields: (
    0:(Name:'ID_________';
      hType:fkNumeric{'N'};
      DataP:00000000; Len:02;
      DecNum:00; MUsrRsrv1:0000;
      WorkID:00; MUsrRsrv2:0000;
      SetFldData:00; Reserved: …),
    1:(Name:'NAME_______';
      hType:fkChar{'C'};
      DataP:00000000; Len:05;
      DecNum:00; MUsrRsrv1:0000;
      WorkID:00; MUsrRsrv2:0000;
      SetFldData:00; Reserved: …)))
    0060:0D ¦.¦
0061:Hdr.H.HdrLen^: TDataArray =  (
  0:(F:' '; D: (0:' 1',1:'Alpha')),
  1:(F:' '; D: (0:' 2',1:'Beta ')),
  2:(F:' '; D: (0:' 3',1:'Gamma')))
    0079:1A ¦.¦
```

References:

[1] B. Liskov, J. Guttag, *Abstraction and Specification in Program Development*, The MIT Press, 1986.

[2] *Borland Delphi 2.0*, .\SOURCE\VCL \TypeInfo.pas.

[3] Microsoft Product Support Services Application Note (Text File). SS0288: "Relocatable Object Module Format."

[4] M.J. O'Leary, "Portable Executable Format", *Microsoft Developer Support documents*, file PE.TXT.

[5] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley: The Java Series, 1996 (http://java.sun.com/docs /books/vmspec/index.html, ftp://ftp.javasoft. com/docs/specs/vmspec.html.zip).

[6] N. Ramsey, M. Fernández, *The New Jersey Machine-Code Toolki.*, http://www.cs.purdue. edu/homes/nr.

[7] S. Vassilyev, Machine Synthesis of Theorems, *Journal of Logic Programming*, Vol.9, No.283, 1990, pp. 235-266.

[8] S. Vassilyev, The Method of Synthesis of Derivability Conditions for Horn Formulas, *Proc. SMC'98 Conference*, 1998, pp. 1451-1456.