Generation of code for reading data from the declarative file format specifications written in language FlexT

Alexei Hmelnov, Andrey Mikhailov

Matrosov Institute for System Dynamics and
Control Theory of Siberian Branch of Russian Academy of Sciences
Irkutsk, Russia
http://idstu.irk.ru

November 23, 2018
Moscow
The Ivannikov ISP RAS Open Conference

The language FlexT

FlexT – **Flex**ible **T**ypes.
Flexible types – the types, that can adjust to the data (sizes and subitem offsets may vary).

The main goals of the language FlexT:

- provide the instrument, that can help us to explore and understand the contents of the binary files using format specifications (check and view data using specification);
- check whether the format specification is correct using the samples of the format data (check specification using data).

The FlexT data viewer makes the binary data transparent.

The advantages of specifications

in comparison with the possible sources of information about a file format:

- **documentation** The vast majority of the format specifications written in natural language contain errors and ambiguities, which can be detected and fixed by trying to apply the various versions of specification to the real data to find the correct variant of understanding of the format description
- **source code** The information about a file format may also be obtained from the source code of a program that works with it. But the code contains a lot of unessential details of some concrete way of data processing. So, the resulting specification will be much more concise and understandable
- **data samples** We have a successful experience of reverse engineering of some file formats using just the samples of data

Generation of data reading code

- The format specifications are required to write a correct program, that should work with the files of the format
- Because the FlexT language data types look similar to that of imperative languages, it is possible to immediately use some parts of specification to declare the data types, constants, and so on, which are required to write the data processing code. Anyway the process of writing the code manually is still time-consuming and error-prone
- So, we have implemented the code generator, which can automatically produce the data reading code in imperative languages from the FlexT specifications
- By its expressive power the FlexT language outperforms the other projects developing the binary format specifications, so the task of code generation for the FlexT specifications is rather nontrivial
- By now we have implemented the code generation for the most widely used FlexT data types, but some complex types are not supported yet

Features of the FlexT language

- The major part of the information about a file format is represented by the data type declarations
- In contrast to the data types of imperative programming languages, the FlexT data types can contain data elements, the size of which is determined by the specific data represented in the format. Thus, we can say that the types flexibly adjust to the data
- After defining the data types, it is required to specify the placement in memory of some data elements which have some of these types
- The language syntax was chosen to be well-understandable by human reader

Parameters and properties of data types

- Data types can have a number of properties (depends on the kind of the type)
- For example, the size and the number of elements are the properties of arrays, and the selected case number is the property of variants
- Each data type has the property Size
- The values of the properties can be specified in the statements of type declaration, and also by expressions that compute the value of this property using the values and properties of the nested data elements, and using the values of the parameters of the type
- The parameters in the type declaration represent the information that needs to be specified additionally when the type is used (called)
- Almost all the FlexT data types have the bit-oriented versions

FlexT data types (1)

Type	Example	Description/purpose
Integer[a]	`num-(6)`	differ by the size and the presence of a sign
Empty[a]	void	the type of size 0, marks a place in memory
Characters[a]	char, wchar, wcharr	In the selected character encoding or Unicode with the byte orders LSB or MSB
Enumeration[a]	`enum byte (A=1,B,C)`	specifies the names of constants of the basic data type

Term enumeration	`enum TBit8 fields(`
	`  R0: TReg @0.3,...)`
	`  of(`
	`  rts(R0) = 000020_,...)`
		simplifies description of encoding of machine instructions, specifies the bit fields, the presence of which is determined by the remaining bits of the number

Set of bits[a]	`set 8 of (` `  OLD ^ 0x02, ...)`	gives the name to bits, the bits can be designated by their numbers (the symbol '=' after the name) or masks (the symbol '^')

Record[a]	`struc`
	`  Byte Len`
	`  array[@.Len]of Char S`
	`ends`
		Sequential placement in memory of named data elements, which may have different types

Variant[a]	`case @.Kind of`
	`  vkByte: Byte`
	`else ulong`
	`endc`
		Selects the content type by the external information

FlexT data types (2)

Type	Example	Description/purpose

Type check[a]	`try`
	`  FN: TFntNum`
	`  Op: TDVIOp`
	`endt`
		Selects the content type by internal information (the first type, which satisfies its correctness condition)

Array[a]	`array[@.Len] of str`
	`array of str ?@[0]=`
	`  0!byte;`
		Consecutive placement of the constituent parts of the same type in memory (the sizes of which may vary). It may be limited by the number of elements, the total size, or the stop condition

Raw data[a]	`raw[@.S]`	Uninterpreted data, which is displayed as a hex dump
Align-ment[a]	`align 16 at &@;`	Skips unused data to align at the relative to the base address offset, which is a multiple of the specified value
Pointer	`^TTable near=DWORD,` `  ref=@:Base+@;`	Uses the value of the base type for specifying the address (for files – the file offset) of the data of the referenced type in memory
Forward declara-tion[a]	`forward`	allows to describe cyclic dependencies between data types
Machine instructions	`codes of TOpPDP ?(@.Op` `  >=TWOpCode.br)and ...;`	machine code disassembling

[a]Supported by the reader code generator

The main principles of our approach to code generation (1)

- We generate the source code, which would look like that carefully written by hand (which may be very eager and relentless). It should produce for us everything we would like to have, but it would be too time consuming to write all that ourselves
- The data reader should give a random access to the various data members and should not be limited by the sequential reading order
- If it is possible to represent some portion of the data by a static data type of the target imperative language, e.g. record (structure) or array, it should be done this way
- To get the random access to the file contents we use the file mapping.
- The simple static data structures are represented by the typed pointers to their location in the mapped file memory, and for the more complex dynamic data structures we generate special classes, which can store all the information necessary to access the data of this type (the *data accessors*)

The main principles of our approach to code generation (2)

- The data reader for a file format should have the methods, which allow to access all the variables declared in the specification. These methods should return the typed pointers for the variables of the static types and the corresponding data accessor class instances for the variables of the dynamic types. Whereas the data accessors provide access to the pointers or accessors of their data elements and so on
- We create the data accessors on demand, because many scenarios of the data reader usage will not work with the entire contents of the file
- The programmer may dispose of the data accessor after using it. It allows to perform, for example, some sequential data reading with a minimum additional memory consumption
- The code generation is performed using an intermediate representation, which describes the main features of a general imperative language

The bit pointers and bit-oriented data types

- The FlexT data types may be not only byte-oriented but also bit-oriented.
- The address of a bit-oriented data member is the address of its starting bit.
- To represent the bit pointers we use a special simple class, which stores the pointer to byte as well as the number of the bit in the byte.
- The interpretation of the bit number depends on the byte order (for LSB the numeration starts from the lowest bit of the byte, and for MSB – from the highest).
- The bit pointer itself is byte order agnostic: it provides the methods, which allow to get the required number of bits at the bit address for both the LSB and the MSB byte orders.
- The bit-oriented data accessors use the bit pointers according to their sizes and bit orders.
- For the static bit-oriented data types we generate the lightweight classes, which inherit from the class of bit pointer.
- The lightweight classes for the static bit-oriented records for each their field have the corresponding method, which returns the value of the field, if the field is numeric, etc, and the lightweight accessor, if the field is more complex.
- The lightweight accessors are created on demand by the functions, which return them.
- On the other hand, the dynamic bit-oriented FlexT data types still require the full accessor classes.

The static data types

FlexT	Pascal
`num+(4)`	`LongWord`
`array[4] of byte`	`array[0..3]of byte`

`struc`
`  long X`
`  long Y`
`ends`
	`record`
	`  X: LongInt;`
	`  Y: LongInt;`
	`end`

`num+(3)`	`TInt3 = packed object`
	` protected`
	`  FMem: array[0..2]of Byte;`
	` public`
	`  function Value: Integer;`
	`end ;`

The data accessor classes

- TDataAccessor is the basic class of all the data accessors
- The abstract class TComplexDataAccessor describes the data accessors for the complex data types. It inherits from the TDataAccessor and adds the properties ItemCount and Item[AIndex: Integer]: TDataAccessor. The properties allow to enumerate the data elements, which belong to the complex data reader
- The system module FmtSys has several concrete classes, inherited from TComplexDataAccessor, which can be parent classes of the accessors for record types, selector types and various kinds of array types
- The data accessors form a tree data structure with the data reader being the root of the tree
- The access to some sub-items may cause the growth of the tree, and the programmer can always cut any branch by freeing the corresponding data accessor

The main properties of the TDataAccessor class

Member	Type	Description
Owner	TComplexDataAccessor	the owner of the data element (corresponds to the postfix operator @ in FlexT)
Parent	TComplexDataAccessor	the parent of the data element (corresponds to the property :@ in FlexT)
Reader	TComplexDataAccessor	the data reader, to which the data accessor belong
DP	Pointer	the address of the data element
Size	TOffset	the size of the data element
Index	Integer	the 0-based index of the data element in the list of subitems of its Owner
Offset	TOffset	the offset of the data element in the file memory

The intermediate code

- encapsulates the features of a general imperative language
- for generation of the data structures we have developed the code library which makes the representation of the intermediate code evident and close to the target code
- we use method chaining and open array parameters to make the representation concise and visually compelling
- all the top-level names declared in the library start with the prefix il – the abbreviation for the words "imperative language"(or "intermediate language").
- the actual code is generated from the intermediate representation using the target imperative language specification,

Intermediate code sample

```
sFldName := CatPrefix('F',MName);
FE := ilFld(sFldName);
ABody := ilBlock([
  ilIf(ilAssigned(FE).UnOp(ilNot)).
    ilThen([ABody, ilLet(FE, FldExpr1)]),
  ilRet(FE)]);
```

The intermediate code and the code generated from it

Intermediate code

```
sFldName := CatPrefix('F',MName);
FE := ilFld(sFldName);
ABody := ilBlock([
  ilIf(ilAssigned(FE).UnOp(ilNot)).
    ilThen([ABody, ilLet(FE, FldExpr1)]),
  ilRet(FE)]);
```

Generated Pascal code

```
if not Assigned(FPRest) then begin
  prev := PTShapeRecDataGrp0(DP);
  FPRest := Pointer(TIncPtr(prev)+(4 + SD.Size));
end;
Result := FPRest;
```

Generated C++ code

```
if (!FPRest) {
  prev = (PTShapeRecDataGrp0)FDP;
  FPRest = (PByteArray)(TIncPtr(prev)+(4+SD()->Size()));
}
return FPRest;
```

Translation of expressions

- The internal evaluation of the FlexT expressions is performed by the functions, which, besides from computing the expression value itself, return the logical result – evaluation success or failure.
  Example: the array element fetch operation `a[i]` may fail due to the index `i` being out of the array range
- The failure of the FlexT expression evaluation may be nominal situation. These failures are easily handled by the `exc` operation and may be used to concisely write some expressions and avoid additional preliminary checking.
  Example: the expression `start[#+1] exc count` tries to fetch the value from the array `start` and, if the element index falls outside the array range, returns the value `count`
- Moreover, the `exc` operation is a part of the FlexT idiom, which is used as an alternative to the C++ ternary conditional operation:
  C++: `x>0 ? x : -x` FlexT: `x when x>0 exc -x`
- So, it would be ineffective to always use the structured exception handling of the modern imperative languages for translation
- We translate expressions into a more complex code, which may sometimes contain several operators including the conditional ones and use auxiliary variables for the intermediate values of sub-expressions if necessary

The code structure of the translated expression

We generate the target language expression, which represents the resulting value and, when it is required, the intermediate code of the following overall structure:

```
 <External operators>;
[if <External condition> then begin
   [<Operators>;
      @OpIntPos: ...
      if <Internal condition>... then begin]
        <Internal operators>;
        @OpNextOk: ...
     [end]
 end]
```

The generated code, which matches this template, in the order of increasing complexity may be:

- empty – the additional operators are not required
- simple sequence of operators – no <External condition> and its if
- single if operator after <External operators>
- the most general case with the <Internal condition>. The internal if operator may be located several levels deeper, inside the other if operators (all them without the else part, and each nested conditional operator is the last in the operator sequence)

When combining the code fragments for subexpressions of this structure we have shown that we can obtain the code of this structure again. I.e. the set of code fragments corresponding to the template is closed with respect to the actions performed by the translation function for the FlexT operations.

Example of translation of FlexT expression

FlexT specification of polygon/polyline data in Shape file format

```
TArcData struc
  TBBox BBox
  long NumParts
  long NumPoints
  array[@.NumParts] of long Parts
  array[@.NumParts] of struc
     TXPointTbl((@@@.Parts[@:#+1] exc @@@.NumPoints)-@@@.Parts[@:#])
          T
   ends Points
ends
```

Generated Pascal code, which provides accessor for the field T

```
function TTArcData_Sub1Accessor.T: TTXPointTblAccessor;
var
  i0: Integer; ndx0: Integer;
begin
  if not Assigned(FT) then begin
    ndx0 := Index+1;
    if (ndx0>=0)and(ndx0<TTArcDataAccessor(TTArcData_Sub2Accessor(
         Parent).Parent).Parts.Count) then
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).
           Parts.Fetch(ndx0)
    else
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).
           NumPoints;
    FT := TTXPointTblAccessor.Create(Self,0,0,i0-
      TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.
           Fetch(Index));
  end;
```

Generation of the test application

- The first thing any programmer will want to do after generation of a data reader is to test whether it works well
- To perform the test it is required to write some application, which will use the data reader somehow
- The most obvious and illustrative task here is to print using the data reader
- After creating manually several test programs of this kind we have found that the process is rather tedious and that it should be automated
- So, we have developed the algorithm, which automatically generates the test code
- The test program generated together with the data reader allows to immediately check the reader
- Of no less importance is the fact that the source code of the program demonstrates the main patterns of data access using the reader

Test application styles

We have implemented the following two styles of test code generation:

- For the simple file formats without the recursive data types we may use the immediate write style
- For the more complex file formats with the recursive data types we generate the write procedures, which may call each other recursively if it is required

Fragments of the test application code in C++, immediate write style

```
std::unique_ptr<TSHPReader> must_free_Reader(new TSHPReader(FN));
Reader = must_free_Reader.get();
if (!AssertTShpHeader(Reader->Hdr(),Reader))
  exit(2);
cout<<"Hdr:"<<endl;
cout<<sIndent<<"Magic: "<<Reader->Hdr()->Magic.Value()<<endl;
...
cout<<sIndent<<"FileLength: "<<Reader->Hdr()->FileLength.Value()<<
    endl;
cout<<sIndent<<"Ver: "<<Reader->Hdr()->Ver<<endl;
...
cout<<"Tbl:"<<endl;
for (i=0; i<Reader->Tbl()->Count(); i++) {
  V = Reader->Tbl()->Fetch(i);
  cout<<sIndent<<"["<<i<<"]:"<<endl;
  cout<<sIndent<<"RecNo: "<<V->RecNo()<<endl;
  cout<<sIndent<<"Len: "<<V->Len()<<endl;
  if (!V->Data()->GetAssert())
    exit(2);
  cout<<sIndent<<"Data:"<<endl;
  cout<<sIndent<<"ST: "<<TShapeTypeToStr(V->Data()->ST())<<endl;
  cout<<sIndent<<"SD:"<<endl;
  switch ( (TShapeRecData_Sub0_Case)V->Data()->SD()->hCase() ) {
    case hcPoint:
      cout<<sIndent<<"Point:"<<endl;
      cout<<sIndent<<"X: "<<V->Data()->SD()->cPoint()->X<<endl;
      cout<<sIndent<<"Y: "<<V->Data()->SD()->cPoint()->Y<<endl;
      break;
    ...
    case hcMultiPointZ:
      cout<<sIndent<<"MultiPointZ:"<<endl;
      ...
      cout<<sIndent<<"Points:"<<endl;
      for (i13=0; i13<V->Data()->SD()->cMultiPointZ()->A()->Points()
```

Fragments of the test application code in Pascal, procedural style

```
procedure printTClassFile_Sub0(const sIndent: String; AV:
    TTClassFile_Sub0Accessor);
var
  i: Integer;
  V: TCp_infoAccessor;
begin
  for i:=0 to AV.Count-1 do begin
    V := AV.Fetch(i);
    Writeln(sIndent,'[',i,']:');
    printcp_info(sIndent+'  ',V);
  end;
end;
...
procedure printTClassFile(const sIndent: String; AV:
    TTClassFileAccessor);
var
  sIndent1: String;
begin
  Writeln(sIndent,'minor_version: ',AV.minor_version);
  Writeln(sIndent,'major_version: ',AV.major_version);
  Writeln(sIndent,'C_pool_count: ',AV.C_pool_count);
  Writeln(sIndent,'C_pool:');
  sIndent1 := sIndent+'  ';
  printTClassFile_Sub0(sIndent1,AV.C_pool);
  ...
end;
...
    Reader := TClaReader.Create(FN);
    try
      Writeln('magic: ',Reader.magic);
      Writeln('Hdr:');
      printTClassFile('  ',Reader.Hdr);
    finally
      Reader.Free;
```

Conclusion

- The code generation is performed through the intermediate data structures. It allows to build the code for various programming languages from the same source. The currently supported languages are Pascal and C++.
- The current level of capabilities of the code generator is well characterized by that it have successfully produced a full-featured data reader code for the well-known for the GIS community Shape file format. The FlexT specification of the Shape format takes approximately 180 lines of code. The code generator have produced 1570 lines of the reader code, and 375 lines of the test program.
- The algorithm developed was also used for generation of the data readers for some custom scientific file formats.

We compared readers generated by Kaitai Struct and FlexT on TUNKA experiment files. For read only one package from file sized 4 000 000 bytes:

- Kaitai Struct use $\approx$ 6 340 848 bytes
- FlexT use $\approx$ 17 152 bytes

The real size of one package in memory is 8 327 bytes

Generation of code for reading data from the declarative file format specifications written in language FlexT

Alexei Hmelnov, Andrey Mikhailov

Matrosov Institute for System Dynamics and
Control Theory of Siberian Branch of Russian Academy of Sciences
Irkutsk, Russia
http://idstu.irk.ru

November 23, 2018
Moscow
The Ivannikov ISP RAS Open Conference