1 SBF codegeneration

The Simple Binary Format is a format for serialization/deserialization of a sequence of hierachically structured messages in a stream of raw binary data. The word "simple" in the name of the format serves the same purpose as the word "democratic" in the name of a country.

In the SBF framework, one first defines a *XML schema* – an XML file, that describes the binary layout of the messages. Typicaly, a special code-generating tool then reads this XML file, and produces a code (usually `Java` or `C++`) that contains the message type declaraions and read/write routines for them. The generated code can then be linked with the business logic software.

In this chapter I'll describe such a code-generation tool fori the OCaml programming language.

1.1 SBF brief overview

1.1.1 SBF simple types

At the root of it, SBE relies on a small number of common types: ASCII-characters, signed or unsigned integers of various sizes and IEE754 floating-point numbers.

In the XML schema any such "primitive" type might be augmented by one of the following modifiers:

- It can have a `nullValue`, which denotes the null or none value for the field of that type.
- It can have an associated `length` value, meaning that the given field is a sequence of values of that type.
- It can have a `constant` "presence" – in that case the corresponding field is never read or written to the binary stream. Instead, the field is always equal to a constant value, provided in the XML file.

1.1.2 SBF complex types

Three kinds of "complex" types can then be construced based on the primitive types described above:

- The `composite` type is a sequence (a record) of fields of various types.
- The `enum` represents a number of mutually-exclusive cases. Each case encoded with a constant "case-id", provided in the XML schema.
- The `set` is a collection of boolean fields, packed into a single bit field.

For `enum` and `set` types, an `encodingType` name must be provided.

1.1.3 SBF messages

In the XML schema, the declaration of all the necessary simple and complex types is folowed by the declaration of various messages. Each message contains a block of fields that are always present in the message, followed by a number of variable-sized groups. Each group is stored as a sequence of repeated blocks, with each block containing the same fixed number of fields.

1.2 OCaml codegeneration strategy

1.2.1 Encoding types

The following conversion between the simple types and their modifiers is used.

SBE type	OCaml type
`int8`	`int`
`int16`	`int`
`int32`	`Int32.t`
`int64`	`Int64.t`
`char`	`char`

SBE type modifier	OCaml parametrized type
`length`	`'a list`
`nullValue`	`'a option`

Instead of describing all the details of the conversion procedure and all the employed naming conventions, I'll instead rely on a number of self-explanatory examples, shown on the next page. Each example shows an XML schema entry followed by a corresponding OCaml code.

The `composite` types are represented as OCaml record-types (1) with each record entry having the correstponding primitive type. The `enum` types are represented as OCaml variant-types (2a) with each variant case beaing a constant. If the `enum` type has a nullable `encodingType` as in example (2b), then one extra case is added to the variant. Finally, the `set` types are treated as records (3) but with all entries being of the `bool` type.

(1) Composite type

```
<composite name="FLOAT">
    <type name="mantissa" primitiveType="int64"/>
    <type name="exponent" primitiveType="int"/>
</composite>
```

```
type t_FLOAT = {
     f_FLOAT_mantissa : int64;
     f_FLOAT_exponent : int
}
```

(2a) Enum type

```
<enum name="LegSide" encodingType="uInt8">
    <validValue name="BuySide" >1</validValue>
    <validValue name="SellSide">2</validValue>
</enum>
```

```
type t_LegSide =
     | V_LegSide_BuySide
     | V_LegSide_SellSide
```

(2b) Enum type with null `encodingType`

```
<enum name="AggressorSide" encodingType="uInt8NULL">
    <validValue name="NoAggressor">0</validValue>
    <validValue name="Buy">1</validValue>
    <validValue name="Sell">2</validValue>
</enum>
```

```
type t_AggressorSide =
    | V_AggressorSide_NoAggressor
    | V_AggressorSide_Buy
    | V_AggressorSide_Sell
    | V_AggressorSide_Null
```

(3) Set type

```
<set name="SettlPriceType" encodingType="uInt8">
    <choice name="Final">0</choice>
    <choice name="Actual">1</choice>
</set>
```

```
type t_SettlPriceType = {
    r_SettlPriceType_Final : bool;
    r_SettlPriceType_Actual : bool;
}
```

1.3 The `cme_codegen` tool

The OCaml codegenerator takes as an input the XML schema file (set with `-i` flag) and writes three files into a specified directory (set with `-d` flag)

```
$ ./cme_codegen.native -i templates.xml -d outputdir
$ ls outputdir
message_types.ml  readers.ml  writers.ml
```

The `message_types.ml` file contains all the OCaml type declarations and only them. At the very bottom of the file, the *message* type is declared – it encompasses all the messages in a single variant type.

The `readers.ml` and `writers.ml` files contain the reading and writing routines for various types, for individual messages and for the `Message_types.message`.

2 Linking the CME code with an IMANDRA model

This chapter is devoted to the details of connectig the code generated for the CME protocol with the model, that is going to be used in the `IMANDRA`.