somFree Compiler and Emitter Framework

User's Guide

Introduction

somFree Compiler and Emitter Framework is a free open source binary compatible reimplementation of IBM SOM Compiler and Emitter Framework. It is tries to be as compatible as possible on API and ABI level.

somFree Compiler

The somFree Compiler is a tool to produce various file formats from Interface Definition Language (IDL) files or Object Interface Definition Language (OIDL) files. somFree Compiler reads IDL or OIDL file and produces an abstract graph tree. Using abstract tree, somFree Compiler generates an object graph tree. After the object graph is ready, somFree Compiler produces an output using template.

The somFree Compiler uses DLL-name based loading of classes libraries (other programs can user another approach, like WPS does. WPS uses an Interface Repository to find corresponding class). Most of the somFree Compiler classes libraries it is implementation of corresponding emitter. Emitters can be created with help of Emitter Framework.

somFree Compiler actually is a client program which uses Emitter Framework classes. somFree Compiler is open source program with an open architecture. The only things that couldn't be easily extended are parser, abstract graph builder and object graph builder. Other things can be shadowed and replaced by our own.

Let's look at somFree Compiler command line syntax to understand how to produce corresponding skeleton code from somFree Compiler template (below is somFree Compiler help screen):

```
sc [-C:D:E:I:S:VU:cd:hi:m:prsvw] f1 f2 ...
Where:
      -C <n>            - size of comment buffer (default: 200000)
      -D <DEFINE>       - same as -D option for cpp.
      -E <var>=<value>  - set environment variable.
      -I <INCLUDE>      - same as -I option for cpp.
      -S <n>            - size of string buffer (default: 200000)
      -U <UNDEFINE>     - same as -U option for cpp.
      -V                - show version number of compiler.
      -c                - ignore all comments.
      -d <dir>          - output directory for each emitted file.
      -h                - this message.
      -i <file>         - use this file name as supplied.
      -m <name[=value]> - add global modifier.
```

```
      -p                - shorthand for -D__PRIVATE__.
      -r                - check releaseorder entries exist (default: FALSE).
      -s <string>       - replace SMEMIT variable with <string>
      -u                - update interface repository.
      -v                - verbose debugging mode (default: FALSE).
      -w                - don't display warnings (default: FALSE).

Modifiers:
      addprefixes : adds `functionprefix' to method names in template file
      [no]addstar : [no]add `*' to C bindings for interface references.
            corba : check the source for CORBA compliance.
              csc : force running of OIDL compiler.
       emitappend : append the emitted files at the end of the existing
file.
         noheader : don't add a header to the emitted file.
            noint : don't warn about "int" causing portability problems.
          nolock : don't lock the IR during update.
            nopp : don't run the source through the pre-processor.
            notc : don't use typecodes for emit information.
       nouseshort : don't generate short names for types.
        pp=<path> : specify a local pre-processor to use.
         tcconsts : generate CORBA TypeCode constants.

Note: All command-line modifiers can be set in the environment
by changing them to UPPERCASE and preappending "SM" to them.

Environment Variables:
      SMEMIT=[h;ih;c;xh;xih;xc;def;ir;pdl]
        : emitters to run (default : h;ih).
      SMINCLUDE=<dir1>[;<dir2>]+
        : where to search for .idl and .efw files.
      SMKNOWNEXTS=ext[;ext]+
        : add headers to user written emitters.
      SMTMP=<dir>
        : directory to hold intermediate files.
      SOMIR=<path>[;<path>]+
        : list of IRs to search.

Pragmas:
      #pragma somemittypes on          : turn on emission of global types.
      #pragma somemittypes off         : turn off emission of global types.
      #pragma modifier <modifier stm>; : instead of modifier statement.
```

Now let's explain some command line switches deeper.

First of the most interesting switch is -s. By default somFree Compiler uses SMEMIT environment variable to determine which emitter to use. Look at emit*.dll files for corresponding emitter. Using switch -s you can change default logic and select one-time emitter instead of global emitters. In easy situation you need only one emitter (say, C emitter). In complex situations you need use more emitters (say, C, H, DEF and IH emitters). You can create your own emitter to produces, for example, some sort of documentation and other stuff.

Another interesting switch is -m. Using -m you can set and/or unset so named modifiers. Modifiers allow you to change default behaviour of emitter and compiler. As example, by default compiler adds new methods or modifies existent. You can tell compiler just add new text to end of file. Modifiers can control emitters. addstart and noaddstar controls C emitter to add or not add pointer sign (*) to references of objects.

Switch -u adds or updates Interface Repository with new information about class interface. Interface repository filename controlled by SOMIR environment variable. This thing useful to add info for Object REXX access and other things which uses Interface Repository.

Other switches are like for standard C/C++ preprocessor and not described here.

Now let's play with somFree Compiler. Most often, you need to create interface files for C/C++ client programs. Usually you need to call the SOM Compiler as following:

```
sc -sdef somobj.idl
sc -sh somobj.idl
```

In case of C++ you need to call:

```
sc -sdef somobj.idl
sc -sxh somobj.idl
```

Of course, not very nice to call somFree Compiler so often. And somFree Compiler provides such functionality:

```
sc -sdef;h;xh somobj.idl
```

The above command will do exactly as all recent commands.

The above emitters were designed for IBM toolset. Nowadays, developers also use GCC or Open Watcom Compilers. The problem here is that Watcom Linker doesn't support .DEF files, but has its own .LNK linker files. In case of one or two classes no many problems to convert .DEF files to .LNK files manually. But such approach just ugly for MUCH classes. So, one of good solution is write REXX script for DEF→LNK conversion. But somFree Compilers contains Open Watcom Linker Emitter for such approaches.

Now let's talk about internals of somFree Compiler. somFree Compiler designed in the way as most of C compilers implemented. It is exists of following parts:

- IDL Preprocessor
- IDL Parser
- OIDL Preprocessor
- OIDL Parser
- Emitter Framework

somFree Compiler first calls IDL or OIDL Preprocessor. Output of IDL/OIDL Preprocessor goes to IDL/OIDL Parser. IDL/OIDL Parser creates Object tree from IDL/OIDL source. Object tree, using templates and emitters, stored to file.

As you see, most of the parts can be extended or replaced by its own implementation. For example, we can reuse CPP instead of SPP. Why not? Just support required command-line switches for

compatibility. Also, default emitters can be rewritten. For C and C++ emitter it is not so hard. For other, more structured languages, like Pascal, Modula, etc. emitter creation is more hard work, but it is also possible.

Actually, we can extend and rewrite somFree Compiler as we want.

Usage of somFree Compiler will not be problem for most of you. But understanding some details about compiler internals makes life easer.

SOM Interface Definition Language

Latest IBM SOM 3.0 supports CORBA IDL mostly at level of CORBA 1.1. somFree supports CORBA IDL 4.2 with all extensions found in SOM IDL.

Preprocessing

IDL shall be preprocessed according to the specification of the preprocessor in ISO/IEC 14882:2003. The preprocessor may be implemented as a separate process or built into the IDL compiler.

Lines beginning with # (also called “directives”) communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of IDL; they may appear anywhere and have effects that last (independent of the IDL scoping rules) until the end of the translation unit. The textual location of IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (\), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline. A backslash character may not be the last character in a source file.

The primary use of the preprocessing facilities is to include definitions from other IDL specifications. Text in files included with a #include directive is treated as if it appeared in the including file.

todo: #line todo: #pragma

Syntax

- Type and Constant Declarations (optional)
- Exception Declarations (optional)
- Interface Declarations (optional)
- Module declaration (optional)

Example

Let's try to define our class interface.

Interface Definition Language (IDL) is the core of System Object Model. All classes have definition of its interface via IDL. With help of somFree Compiler IDL file can be translated to various formats,

including various language bindings. For example, to produce C header you can run

```
sc -s"h" somcls.idl
```

to produce DEF file you can run

```
sc -s"def" somcls.idl
```

somFree Compiler uses emitter to produce corresponding language binding. You can create new bindings emitter using Emitter Framework.

First of all, think about your class. What it must do? Define them in terms of object. Propose attributes and methods of class.

Ok. Imagine, we need class to have access to Java objects. Let's write a class interface in terms of Interface Definition Language.

```
#include <somobj.idl>

interface JavaObject : [[SOMObject]]
{
  implementation
  {
    somDefaultInit: override;
    // Init Java Virtual Machine (if no current) and create Java object
    somDefaultDestruct: override;
    // Destruct Java object and close Java Virtual Machine (if needed)
  }
}
```

As you can see, no many problems. Syntax of IDL too closest to C-like languages. First thing you need is to include definitions of parent classes. In our case it is '`SOMObject`' definition. Generic Java object doesn't need to have any methods. Only thing '`JavaObject`' will do its check for existence of Java Virtual Machine and execution of it if required.

On object destruction checks is Java Virtual Machine still required will be done and it will be destroyed if not required. Also same constructor and destructor will call corresponding constructor and destructor of Java object.

Classic IDL file contains definitions like

```
interface <class> : <parent_class>
{
  attribute <type> <name>

  <type> <method>(<parameters>)
}
```

[http://www.omg.org OMG] IDL doesn't support methods override. SOM IDL has such feature (and incompatibility with OMG CORBA). This is done via keyword '`implementation`'. To solve problems with other IDL compilers such part must be wrapped to #ifdef structure:

```
#include <somobj.idl>

interface JavaObject : [[SOMObject]]
{

  #ifdef __SOMIDL__
  implementation
  {
    somDefaultInit: override;
    // Init Java Virtual Machine (if no current) and create Java object
    somDefaultDestruct: override;
    // Destruct Java object and close Java Virtual Machine (if needed)
  }
  #endif
}
```

Such approach well known in C-world, but also have some problems. For example, IDL of Document Object Model (DOM) (Yes, [http://www.w3.org W3C] DOM uses same IDL as SOM and CORBA) has attribute '`implementation`'. As result, somFree Compiler has some problems with IDL compilation.

As a first step we'll create SOM class interface for java.lang.Object. It can be done with help of javah tool.

```
javah -jni java.lang.Object
```

As result you'll have such file:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class java_lang_Object */

#ifndef _Included_java_lang_Object
#define _Included_java_lang_Object
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     java_lang_Object
 * Method:    hashCode
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_java_lang_Object_hashCode
  (JNIEnv *, jobject);

/*
 * Class:     java_lang_Object
 * Method:    notify
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_java_lang_Object_notify
  (JNIEnv *, jobject);
```

```
/*
 * Class:     java_lang_Object
 * Method:    notifyAll
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_java_lang_Object_notifyAll
  (JNIEnv *, jobject);

/*
 * Class:     java_lang_Object
 * Method:    registerNatives
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_java_lang_Object_registerNatives
  (JNIEnv *, jclass);

/*
 * Class:     java_lang_Object
 * Method:    wait
 * Signature: (J)V
 */
JNIEXPORT void JNICALL Java_java_lang_Object_wait
  (JNIEnv *, jobject, jlong);

/*
 * Class:     java_lang_Object
 * Method:    getClass
 * Signature: ()Ljava/lang/Class;
 */
JNIEXPORT jclass JNICALL Java_java_lang_Object_getClass
  (JNIEnv *, jobject);

/*
 * Class:     java_lang_Object
 * Method:    clone
 * Signature: ()Ljava/lang/Object;
 */
JNIEXPORT jobject JNICALL Java_java_lang_Object_clone
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

So, this header can be used to generate actual class interface using this script:

```
/* REXX - our best dog */

do while lines('java_lang_Object.h')
```

```
   s=linein('java_lang_Object.h');
   parse value s with x 'Header for class' name '*/'
   if name\='' then
   do
     classname=strip(name)
     say '#include <JavaObject.idl>'
     say ''
     say 'interface '||classname||' : JavaObject'
     say '{'
   end
   parse value s with x 'Method:' name
   if name\='' then
   do
     /* skip 3 lines */
     s=linein('java_lang_Object.h');
     s=linein('java_lang_Object.h');
     s=linein('java_lang_Object.h');
     s2=linein('java_lang_Object.h');
     interpret("parse value s with 'JNIEXPORT' type 'JNICALL
Java_"||classname||"_' name")
     name=strip(name)
     s2=strip(s2)
     parse value s2 with start 'JNIEnv *, jobject' end
     s2=start||end
     parse value s2 with x ', ' y
     if x='(' then s2='('||y
     say '  '||type||name||' '||s2
   end
 end
 say '}'
```

As result, you'll have following:

```
#include <JavaObject.idl>

interface java_lang_Object : JavaObject
{
   jint hashCode ();
   void notify ();
   void notifyAll ();
   void registerNatives (JNIEnv *, jclass);
   void wait (jlong);
   jclass getClass ();
   jobject clone ();
}
```

Using such approach you can easily make SOM wrappers for all Java classes. Using Java_JNI_API you can create Java classes and using SOM wrappers you integrate Java code to SOM-based applications. Because SOM API more generic then Java API you can use any available language bindings for development. Also, you can start to extend Java classes by native code with help of SOM engine.

SOM Object Interface Definition Language

SOM Object Interface Definition Language is a pre-IDL object definition language used before IBM SOM 2. Since IBM SOM 2 uses CORBA IDL as defined in OMG CORBA 1.1. SOM Object Interface Definition Language (OIDL) is a simple definition language and not recommended to use. SOM Compiler support is only for compatibility with old source code. OIDL support implementation mostly based on [2] and various OIDL source files found on the Web. OIDL consist of sections set:

- Include section (optional)
- Class section (required)
- Release order section (optional)
- Parent class section (required)
- Passthru section (optional)
- Metaclass section (optional)
- Data section (optional)
- Methods section (optional)

Include section (Preprocessing)

Include section handled by SPP preprocessor. Preprocessor is a simple tool which includes files pointed by include section into the original file.

Include section is optional and contains names of OIDL files with definition of parent class, metaclasses and private interfaces of ancestor classes.

```
[#include ( <ancestor> | "ancestor" ) ] *
#include ( <parent> | "parent")
[#include ( <metaclass> | "metaclass" ) ]
```

ancestor is the name of the OIDL file containing the private part of an ancestor class' interface needed in the definition of this class. If ancestor is enclosed in angle brackets (<>), the search for the file will begin in system-specific locations. If parent is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations.

parent is the name of the OIDL file containing the parent class of the class for which the Include statement is provided. If parent is enclosed in angle brackets (<>), the search for the file will begin in system-specific locations. If parent is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations.

metaclass is the OIDL file containing the metaclass of the class for which the include statement is provided. If metaclass is enclosed in angle brackets (<>), the search for the file will begin in system-specific locations. If metaclass is enclosed in double quotation marks (""), the search for the file will begin in the local context, then move to the system-specific locations.

todo: @ line “file” “filepath”

Class section

```
class: name
  [, file stem = stem]
  [, external stem = stem]
  [, function prefix = prefix |
  , external prefix = prefix |
  , classprefix = prefix]
  [, major version = number]
  [, minor version = number]
  [, global | local];
  [, classInit = function];
[description]
```

Release order section

```
release order: name [, name ]* ;
```

Parent class section

```
parent [class]: name;
description
```

Passthru section

```
[passthru: language.suffix, [ before | after ];
line 1
line 2
endpassthru; [description]]*
```

Metaclass section

```
metaclass: name;
[description]
```

Data section

```
data:
[description1 ]
[declaration [ , private | , public | , internal] [, class];
[ description2]]*
```

Methods section

```
methods:
```

```
[description1 ]
[[group: name;
[ description2]]
[method prototype
[, public | , private]
[, method | , procedure]
[, class]
[, offset | , name lookup]
[, local | , external]
[, use = name];
[descriptionS]]*
[override: method name
[, public | , private]
[, class]
[, local | , external]
[, use = name];
[description4]] *
```

Appendixes

1. Appendix 1. SOM ABI

Due to switching from MSVC (IBM SOM 2.1) to VAC (IBM SOM 3.0) some problems were occurred:

The first problem is a calling convention. All non SOMLINK calls in IBM SOM 2.1 are a _cdecl calls. But under IBM SOM 3.0 all non SOMLINK calls are an Optlink calls. Read some info here: https://github.com/prokushev/SOM-Delphi-Wiki/blob/master/Known%20differences%20between%20SOM%202.1%20and%20SOM%203.0.md Goal of somFree SOM Compiler and Emitter Framework is to provide a possibility to use original IBM SOM emitters as from IBM SOM 2.1 as from IBM SOM 3.0. Another goal is a development of somFree emitters, which can be used on both IBM SOM 2.1 and IBM SOM 3.0 compilers. To achieve above goals somFree provides some solutions: 1. Automatic somc.dll calling convention switching. somFree SOMC.DLL provides automatic switching of IBM SOM 2.1 ABI and IBM SOM 3.0 ABI. Switching occurs on boatload call during loading of emitter. For IBM SOM 3.0 all emitter contains entry point emitSL, so, if loading was success, then somFree handles Optlink calling convention for all non SOMLINK calls. If no such entry (found only emit) then IBM SOM 2.1 ABI used. 2. Support both entry points (emitSL and emit) in emitters. somFree emitters automatically switches to IBM SOM 2.1 ABI on emit call and to IBM SOM 3.0 ABI on emitSL call.

todo: add info about internal structures like in
http://www.edm2.com/index.php/OS/2_Application_Binary_Interface_for_PowerPC_(32-bit)
http://www.os2site.com/sw/dev/info/abippc32.pdf

2. Appendix 2. Terminology changes and syntax construct

evolution

During SOM compiler evolution, some terms were changed and/or obsolete. Here is a table with terms mapping.

SOM 1.0	SOM 2.0	somFree
attribute	modifier	annotation
group	{obsolete}	{obsolete}

IBM SOM 1.0 introduced Object Interface Definition Language (OIDL) which predates Interface Definition Language (IDL). It was a simple language which provided simple objects description. IBM SOM 2.0 switched to SOM IDL, which was an extended version of OMG IDL. SOM Compiler provided OIDL to IDL conversion tools. Because of moving to IDL, `attribute` were renamed to `modifier` and `group` obsolete. SOM IDL extensions mostly `implementation` section. IBM SOM 3.0 added some more OMG IDL compatibility according to CORBA 2.x standards, like `#pragma` statement. Modifiers now can be declared via `#pragma modifier`. Some modifiers were only defined via `#pragma`.

somFree supports SOM IDL as found in IBM SOM 3.0.

osFree somFree compiler tries to implement current IDL 4.2 Specification. It supports `@annotation`. So, many constructions can be (and must be) implemented via standard and somFree specific annotations.

OIDL	SOM IDL	IDL 4.2

Bibliography

1. Object Management Group, “C Language Mapping Specification 1.0, 1999” Available: https://www.omg.org/spec/C/
2. IBM, “OS/2 2.0 Technical Library. System Object Model Guide and Reference. First Edition, 1991” Available: https://www.os2museum.com/files/docs/os220tl/os2-2.0-som-1991.pdf
3. Object Management Group, “The Common Object Request Broker: Architecture and Specification. Revision 1.1, 1991” Available: https://www.omg.org/spec/CORBA/1.1
4. Object Management Group, “Interface Definition Language™. Version 4.2, 2018” Available: https://www.omg.org/spec/IDL

Tools Reference
MKMSGF MSGEXTRT MSGBIND BIND JWASM UNI IPFC
somFree Compiler and Emitter framework
User's Guide Programmer's Guide Programmer's Reference

2024/10/09 03:43 · prokushev · 0 Comments