FAMIX C++ language plug-in 1.0

Author	Holger Bär (baer@fzi.de)
Version	1.0
Last modified	1999-09-07

1 Abstract

This document defines a language plug-in for FAMIX, the FAMOOS information exchange model [Deme99]. It extends, instantiates and modifies the FAMIX core model to cover most of the entities and relationships that can be found in C++ source code.

2 Notation

The common exchange model is modified in three different ways to handle C++ sources:

- New classes are added to the basic FAMIX model to model entities and associations unique to C++. These classes are marked as new entities and new associations respectively.
- New attributes are added to existing classes of the basic FAMIX model. In this case the class is marked "modified" and only the new and modified (see below) attributes are listed in the definition of the modified class.
- The definition of attributes of existing classes are modified or their syntax and semantics are instantiated for C++. The instantiation case mostly occurs for attribute definitions in the core model having a phrase like "… is a language dependent issue".
  Like with new attributes the corresponding class is marked "modified" and the modified attributes are listed in the definition of the modified class. To discriminate modified from new attributes, modified attributes are listed without any type information since that information isn't modified anyway.

3 Modified classes

3.1 Class (interpreted and extended)

Class	
instantiatesTemplate (): Name	# new
friendsAt (pos Integer): Name	# new
isUnion (): Boolean	
isAbstract	

Figure 1: Class

Each *definition* of a class in source code constitutes this entity. A class definition in C++ takes the form

```
class A { ... };
```

in contrast to a class declaration of the form:

```
class A;
```

Structures (`struct`) are modelled as classes. They differ only from classes in their members and base classes being public by default in spite of private in the case of classes. So only the parser must know the difference between classes and structures.

Unions (`union`) are also modelled as classes. They are a kind of restricted classes: they can't have any base classes, can't be used as base classes, have no virtual methods, have no static methods or attributes. For a detailed description of all restrictions to unions please refer to [ISO98].

The new or modified attributes are:

- `instantiatesTemplate: Name; optional`
  If the class is a template instantiation this attribute refers to the corresponding template. The uniqueName of the class template is used as a reference. How to deal with class templates is described in section 4.1.
- `friends: 0 .. N Name; optional`
  This is a multi-valued attribute holding all friend classes and friend behavioural entities of a class. The friend entities are referenced by their uniqueName.
- `isUnion: Boolean; optional`
  This attribute is true, iff a C++ union is modelled by this entity.
- `isAbstract`
  In C++ a class is abstract, iff at least one of its methods is abstract.

3.2 BehaviouralEntity (interpreted and extended)

BehaviouralEntity	
baseReturnType (): Name	# new
instantiatesTemplate (): Name	# new
accessControlQualifier	
signature	

Figure 2: BehaviouralEntity

The new and modified attributes are:

- `baseType: Name; optional`
  The core model defines the totally not interpreted attribute declaredReturnType and the attribute declaredReturnClass, which refers to the class in the return type declaration of the Behavioural Entity, if there is any. For Behavioural Entities with a fundamental return type, for example, there is no declared class. The new attribute baseReturnType is similar to the declaredReturnClass attribute: it refers to the *most specific type representable* in the language model, i.e., core model plus language plug-in. If we have `C* f()` in the source code, the baseReturnType of f is class C with the current language plug-in. If future versions of the C++ plug-in would also define pointers as type entities, the baseReturnType would refer an array entity referring to the class C.
  baseReturnType can't replace declaredReturnClass, because generic FAMIX readers wouldn't see the class C, where the type declaration ends, because they don't know the intermediate pointer entity.
- `instantiatesTemplate: Name; optional`
  If the BehaviouralEntity is a template instantiation this attribute refers to the corresponding function template. The uniqueName of the function template is used as a reference. How to deal with function templates is described in section 4.2.

- `accessControlQualifier`
  The two behavioural entities, Method and Function, have different sets of allowed access qualifiers in C++. Functions have the access qualifiers `static` (local to compilation unit) and `extern` (global). The access to Methods can be controlled by the qualifiers `public`, `protected` and `private`.
- `signature`
  The signature string takes the form `name(T1,T2, ... ,Tn)` (without spaces) where `name` is the method name and $T_{1..n}$ are the types of the formal parameters of the method. Constant parameters of type `T` have the form: `const T`.

3.3 Method (interpreted and extended)

Method	
isDestructor (): Boolean	# new
isOperator (): Boolean	# new
isVirtual (): Boolean	# new
isConst (): Boolean	# new
accessControlQualifier	
signature	
isAbstract	
isConstructor	
uniqueName	

Figure 3: Method

Each *declaration* of a method in source code constitutes this entity. One could have expected each method definition to constitute this entity, but in C++ a method declaration can be used to just manipulate the access to inherited methods. This can only be represented by a new Method entity.

The new or modified attributes are:

- `isDestructor: Boolean; optional`
  Indicates whether or not the method is a destructor. A destructor is a method with no declared return type and a name equal to the name of the class it belongs to prepended with the tilde character ~.
- `isOperator: Boolean; optional`
  Indicates whether or not the method is an operator function.
- `isVirtual: Boolean; optional`
  Indicates whether or not the method is declared virtual.
- `isConst: Boolean; optional`
  Indicates whether or not the method is declared constant. Constant methods assert, that they do not alter the object's attributes, so that they can safely be called upon constant objects.
- `accessControlQualifier`
  The allowed access qualifiers are: `public` (access for anyone), `protected` (access restricted to derived classes) and `private` (access only from within the class).
- `isAbstract`
  A method is abstract, iff it is declared as pure virtual, e.g.: `virtual void m() = 0;`

- `isConstructor`
  A constructor is a method with no declared return type and a name identical to the name of the class it belongs to.
- `uniqueName`
  `const` is appended to the unique name without whitespace, iff isConst is true.

3.4 Function (interpreted)

Function
accessControlQualifier

Figure 4: Function

Each definition of a global function in source code constitutes this entity.

The modified attributes are:

- `accessControlQualifier`
  The allowed access qualifiers are: `static` and `extern`.
  They indicate whether the function is declared external, i.e. globally accessible from all compilation units (each compilation unit results in an object file *.obj), or), which is default for global functions in C++, or static. Static functions are local to a translation unit and can be declared using the keyword `static` like in:
  `static int helperFunc(int n) { ... }`

3.5 StructuralEntity (extended)

Attribute	
baseType (): Name	# new
isConstant (): Boolean	# new

Figure 5: StructuralEntity

The new or modified attributes are:

- `baseType: Name; optional`
  The core model defines the totally not interpreted attribute declaredType and the attribute declaredClass, which refers to the class in the type declaration of the Structural Entity, if there is any. For Structural Entities of fundamental type, for example, there is no declared class. The new attribute baseType is similar to the declaredClass attribute: it refers to the *most specific type representable* in the language model, i.e. core model plus language plug-in. If we have `C* a[]` in the source code, the baseType is class C with the current language plug-in. If future versions of the C++ plug-in would also define pointers and arrays as type entities, the baseType would refer an array entity referring to a pointer entity referring to the class C.
  BaseType can’t replace declaredClass, because generic FAMIX readers wouldn’t see the class C, where the type declaration ends, because they dont’t know the intermediate entities array and pointer.
- `isConstant: Boolean; optional`
  The `const` modifier is used in type declarations to express that the declared object must not be altered after its initialisation. This information could be interesting, e.g., to search for variables not declared as constant but only accessed once for writing. In such cases it is likely that the variable is a constant indeed and could therefore declared constant. In doing so future modifications of the source code cannot accidentally alter this variable.

3.6 Attribute (interpreted)

Attribute
accessControlQualifier

Figure 6: Attribute

The new or modified attributes are:

- `accessControlQualifier`
  The allowed access qualifiers are: `public` (access for anyone), `protected` (access restricted to derived classes) and `private` (access only from within the class).

3.7 GlobalVariable (interpreted)

GlobalVariable	
accessControlQualifier (): Qualifier	# new

Figure 7: GlobalVariable

Each definition of a global variable in source code constitutes this entity.

The new or modified attributes are:

- `accessControlQualifier: Qualifier; optional`
  The allowed access qualifiers are: `static` and `extern`.
  They indicate whether the variable is declared external, i.e. globally accessible from all compilation units (each compilation unit results in an object file *.obj), or), which is default for global variables in C++, or static. Static variables are local to a translation unit and can be declared using the keyword `static` like in:
  `static int helper;`

3.8 InheritanceDefinition (interpreted and extended)

InheritanceDefinition	
isVirtual (): Boolean	# new
accessControlQualifier	
index	

Figure 8: InheritanceDefinition

The new or modified attributes are:

- `isVirtual: Boolean; optional`
  Indicates whether or not the inheritance is virtual, i.e., whether a class that is derived from the same base class multiple times via different paths, should include the data members of the base class only once or not.
- `accessControlQualifier`
  The allowed access specifiers are: `public, protected, private`. The specifier sets the maximum access that clients of the derived class will have to the features of the base class.
- `index`
  The index is always ´null´ as name collisions in C++ are not resolved by the order of the base classes.

3.9 Access (extended)

Access	
receivingClass (): Name	# new
receivingVariable (): Name	# new

Figure 9: Access

The new or modified attributes are:

- `receivingClass: Name; optional`
  The statically determinable class of the expression receiving the variable access. For example:
  ```
  C* r;
  r->v = 0;
  ```
  Then C is the receiving class of this access. The receiving class is ´null´ for accesses to global variables. For accesses to local variables and to formal parameters the receiving class is the class the method defining the local variables resp. parameters belongs to, i.e. it is ´null´ for local variables and parameters of global methods (functions). The receiving class is referenced by its uniqueName.
- `receivingVariable: Name; optional`
  The variable r in the above example. The receiving variable is ´null´ for accesses to global or local variables, to formal parameters and within "chain calls". For example the access to `attr` in `r.m1().attr` has no receiving variable. The receiving variable is referenced by its uniqueName.

3.10 Invocation (extended)

Invocation	
receivingVariable (): Name	# new
base	

Figure 10: Invocation

The new or modified attributes are:

- `receivingVariable: Name; optional`
  The variable r in the above example. The receiving variable is ´null´ for invocations of static or global methods and within "chain calls". For example the call to m2 in `r.m1().m2()` has no receiving variable. The receiving variable is referenced by its uniqueName.
- `base: Name; optional`
  In C++ this attribute contains the statically determinable class of the expression receiving the invocation. For example:
  ```
  C* r;
  r->m();
  ```
  Then C is the receiving class of this invocation. For method invocations the candidate attribute holds all methods overriding the method `base::invokes`.

4 New classes

The new classes from 4.3 to 4.6 all define entities representing a type in C++. Consequently they can be referred in the attributes baseType and baseReturnType.

4.1 ClassTemplate

ClassTemplate
templateParameterAt (pos Integer): Qualifier

Figure 11: ClassTemplate

This **new entity** models class templates of C++. It inherits from the entity Class of the core model. One could argue that Class Templates are no proper subclasses of Classes, because they cannot be used in every place a Class can be used, e.g., as the target of a reference via a declaredClass attribute. This is because class templates are no classes but only a template for a class that needs its template parameters to be instantiated to become a proper class. On the other hand the ClassTemplate entity needs all the attributes of a Class entity plus an attribute describing its template parameters. Even the newly defined attribute instantiatesTemplate makes sense, since class templates can be partially instantiated. So letting ClassTemplate not inherit from Class would mean to define a new entity with exactly the same attributes like an existing one plus one attribute. Therefore ClassTemplate inherits from Class. The same argumentation also applies to the new entity FunctionTemplate defined below also inheriting from its corresponding core model entity Function.

The methods an attributes of a class template are modelled as Method and Attribute respectively. If it is necessary to determine whether a method/attribute is part of a template definition, this can be decided by looking at the type of the entity referred by the belongsToClass attribute of the method/attribute.

The usage of one of the template parameters within the class template, e.g., for a type declaration of an attribute or within a function signature, is modelled by a reference to the template parameter. Template parameters are defined in 4.3.

Fully instantiated class templates are modelled as ordinary classes with their template parameters substituted accordingly. Partially instantiated templates are themselves templates with only the bound template parameters substituted accordingly. Each different instantiation produces a different Class or ClassTemplate entity. The instantiated copies have the bound template parameters appended to the uniqueName attribute they would get as an ordinary class in the form of a comma separated list in <> without spaces, e.g.: `P::C<D,int>`.

Besides the attributes inherited from `Class`, the new or modified attributes are:

- `templateParameters: 0 .. N Name; mandatory`
  This attribute holds all template parameters of the modelled class template definition. The entities defined within the class template can use these names, e.g., as their declared type.

4.2 FunctionTemplate

FunctionTemplate
templateParameterAt (pos Integer): Qualifier

Figure 12: FunctionTemplate

This **new entity** models function templates. It inherits from the entity Function of the core model instead of defining a new heir of Entity for the same reasons as with class templates (see 4.1 for a discussion).

The usage of one of the template parameters within the function template, e.g., within the function signature or the type of a local variable, is modelled by a reference to the template parameter. Template parameters are defined in 4.3.

Fully instantiated function templates are modelled as ordinary functions with their template parameters substituted accordingly. Partially instantiated templates are themselves templates with only the bound template parameters substituted accordingly.

Besides the attributes inherited from `Function`, the new or modified attributes are:

- `templateParameters: 0 .. N Name; mandatory`
  This attribute holds all template parameters of the modelled function template definition. The entities defined within the template can use these names, e.g., as their declared type.

4.3 TemplateParameter

TemplateParameter
belongsToTemplate (): Name

Figure 13: TemplateParameter

This **new entity** models template parameters of a ClassTemplate or FunctionTemplate entity. It inherits from Entity and defines no new attributes.

Besides the attributes inherited from `Entity`, the new or modified attributes are:

- `belongsToTemplate: Name; mandatory`
  Refers to the unique name of the Template the TemplateParameter is a parameter of.

The formula for uniqueName is:

```
uniqueName (templParam) = belongsToTemplate (templParam) +
    "." + name (templParam)
```

4.4 FunctionType

FunctionType
signature (): Qualifier
declaredReturnType (): Qualifier
declaredReturnClass (): Name
baseReturnType (): Name
belongsToContext (): Name
isOperator (): Boolean

Figure 14: FunctionType

This **new entity** models the declaration of a function type.

It shares some attributes with BehaviouralEntity but the attributes of BehaviouralEntity describing its role as an callable piece of code do not apply. Unlike Method and Function a FunctionType can be defined in any scope resulting in an attribute belongsToContext that can refer to Package, Class, Method or Function.

The attributes of FunctionType are then:

- `signature: Name; mandatory`
  The signature is defined as in BehaviouralEntity but lacks the name in front of the left bracket.
- `belongsToContext: Name; mandatory`
  Refers to the scope (Package, Class, Method or Function) the function type is declared in. The reference is established by the unique name of the scope.
- The remaining attributes defined above have exactly the same syntax and semantic as they have in the definition of BehaviouralEntity and Method.

The formula for uniqueName is:

```
uniqueName (funcType) = belongsToContext (funcType) + "." +
    signature (funcType)
```

4.5 EnumerationType

EnumerationType
belongsToContext (): Name

Figure 15: EnumerationType

This **new entity** models the declaration of an enumeration type (`enum`).

Enumeration types are modelled because they are often used, especially in not pure object-oriented systems, to describe different options or states (e.g., drawing modes, output destinations). This way they introduce dependencies within the system.

The only attribute of EnumerationType is:

- `belongsToContext: Name; mandatory`
  Refers to the scope (Package, Class, Method or Function) the enumeration type is declared in. The reference is established by the unique name of the scope.

The formula for uniqueName is:

```
uniqueName (enumeration) = belongsToContext (enumeration) +
    "." + name (enumeration)
```

4.6 TypeDef

TypeDef
declaredReturnType (): Qualifier
declaredReturnClass (): Name
baseReturnType (): Name
belongsToContext (): Name

Figure 16: TypeDef

This **new entity** models type aliasing via the `typedef` keyword.

The attributes of TypeDef are:

- `belongsToContext: Name; mandatory`
  Refers to the scope (Package, Class, Method or Function) the type alias is declared in. The reference is established by the unique name of the scope.
- The remaining attributes defined above have the same syntax and analogous semantic as they have in the definition of StructuralEntity.

The formula for uniqueName is:

```
uniqueName (typeDef) = belongsToContext (typeDef) + "." +
    name (typeDef)
```

4.7 TypeCast

TypeCast
belongsToBehaviour (): Name
fromType (): Name
toType (): Name

Figure 17: TypeCast

This **new association** models type cast like `(C*)pointer`.

Type casts are interesting for re-engineering as they often point to problems in the design of a system. There will be an instance of this class for every type cast occuring in the source code, even if the cast is between the same types, because we are interested in all the places where casts occur.

The attributes of TypeCast are:

- `belongsToBehaviour: Name; mandatory`
  Refers to the BehaviouralEntity the cast appears in.
- `fromType: Name; optional`
  Refers to the unique name of the declared type the casted expression has. This is the type of `pointer` in the above example.
- `toType: Name; optional`
  Refers to the unique name of the type the expression is casted to (`C*` in the above example).

4.8 SourceFile

SourceFile

Figure 18: SourceFile

This **new entity** models a file of the source code (header file or implementation file). It defines no additional attributes.

Source files are a grouping unit in C++. An implementation file plus all included header files even defines a scoping unit, the compilation unit (see the definition of Function 3.4 and GlobalVariable 3.7).

The structure of the relationships between source files created by include directives gives a rough overview about the dependencies in the system, because a dependency between two entities *always* is only possible with an include dependency between the files the two entities are defined in. This makes source files together with their include relations quite important for re-engineering purposes.

The values of the `name` and `uniqueName` attributes are specific to the operating system used to compile the sources. One could think of the full path or of a relative path starting from a common root directory that contains any source files of the system in one of its subdirectories.

4.9 Include

Include
includingFile (): Name
includedFile (): Name

Figure 19: Include

This **new association** models an include directive of the preprocessor.

The structure of the relationships between source files created by include directives gives a rough overview about the dependencies in the system, because a dependency between two entities *always* is only possible with an include dependency between the files the two entities are defined in. This makes source files together with their include relations quite important for re-engineering purposes.

The attributes of Include are:

- `includingFile: Name; mandatory`
  Refers to the file containing the include directive.
- `includedFile: Name; mandatory`
  Refers to the file included by the include directive.

5 Excluded features of C++

Some features of C++ are not covered by this language plug-in:

- Type constructors `*`, `[]` and `&`.
  The interesting thing for re-engineering purposes with these type constructors is that the array constructor expresses multiplicity and the pointer constructor *may* mean multiplicity or reference. The reference operator is only a specification for code generation.
- Fundamental types and their operations.
  They do not carry interesting information and pollute the model of the system as, e.g., nearly every class uses some fundamental types.
- Anonymous classes.
  Anonymous classes are not modelled explicitly because they can only be referenced locally, e.g., as an actual parameter in a function or method call. We only see two possible local references:
  - References to the class as a type as in `typedef {int i; ... } moronsType`.
    Then we can set the declaredType to the whole class definition and leave baseType and declaredClass blank.
  - References to the class as a value, e.g. as an actual parameter.
    Then we can treat them as a ComplexExpression.
- Nested classes
- The modifiers `inline`, `volatile`, `auto` and `register`.
  These modifiers concern only the code generation and are therefore of no interest for re-engineering.
- Pointers to class members.
  We can model them as normal pointers by ignoring the fact that it can only access values within a certain class.
- Exceptions.
  If exceptions prove to be of interest we could model them as follows:

- A Class for every exception. Exceptions are classes in fact anyway.
- Ignoring the `try` block.
- The throwing (`throw`) of an exception is modelled as a call to the constructor of the exception.
- The `catch` statement results in a definition of a local variable with the type of the catched expressions.

6 References

[Deme99] Serge Demeyer, Sander Tichelaar and Patrick Steyaert, FAMIX – The FAMOOS Information Exchange Model, version 2.0 alpha, July 1999. See http://www.iam.unibe.ch/~famoos/FAMIX/.

[ISO98] International Standard ISO/IEC 14882, Programming Languages — C++, First Edition 1998-09-01, American Natinal Standards Institute, New York.

Cover Pages

Achievement 2.4.1a

FAMIX C++ language plug-in 1.0

1) Identification

Project Id:	Esprit IV #21975 “FAMOOS”
Deliverable Id:	D 2.2 – FINALFHB Final FAMOOS Methodology Handbook
Date for delivery:	31.08.99
Planned date for delivery:	31.08.99
WP(s) contributing to:	1
Author(s):	Holger Bär

2) Abstract

This document defines a language plug-in for FAMIX, the FAMOOS information exchange model [Deme99]. It extends, instantiates and modifies the FAMIX core model to cover most of the entities and relationships that can be found in C++ source code.

3) Keywords

Object-oriented, reengineering, reverse engineering, code repository, round-trip engineering, FAMOOS, FAMIX, C++.

4) Version History

Ver	**Date**	**Editor(s)**	**Status & Notes**
1.0beta	24.08.99	Holger Bär	
1.0	25.08.99	Sander Tichelaar	

5) Issues for future releases

6) Table of Contents

7) List of Figures

8) List of Tables