Compiler API: Parsing

Parsing is the process of creating an abstract syntax representation of some Alf code. If parsing is successful, the Alf compiler also performs constraint checking on the resulting abstract syntax tree. *Const raint checking* is the process of doing name resolution and validating the Alf code against the static-semantic constraints defined in the Alf specification.

A compilation takes place in the context of a specific Named Element in the UML model, called the *contex t element*. During parsing and constraint checking, name resolution is performed in the scope containing the context element (known, in Alf terms, as the *model scope*). After the successful mapping of Alf code to UML, the context element is updated with the results of the compilation. The context element should be an Activity, Opaque Behavior, Opaque Action or Opaque Expression

In UML, a Behavior may have a context Classifier (for example, the Class that owns a State Machine as its classifier behavior is the context Classifier for the State Machine). This use of the term "context" is unrelated to the term "context element" for an Alf compilation.

To compile some Alf code text, first set the context element using the *setContextElement* method and then call the *parse* method, passing the text. What Alf text can be validly parsed depends on what the context element is.

- If the context element is a Value Specification, then the text is assumed to be for an Alf Expression.
- If the context element is an Opaque Action, then the text may be an Alf Expression or Statement Sequence.
- Otherwise the text is assumed to be for an Alf Statement Sequence.

The Alf code associated with an Activity, Opaque Behavior, Opaque Action or Opaque Expression is known as the *Alf body* of that element. The Alf code is saved in the model differently for different kinds of Elements. However, you can use the *AlfElementUtil.getAlfBody* operation to get the Alf body of an Element (if it has one).

After the parsing process completes, you can check if it was successful by calling the *isSuccessful* method. If the parse is successful, then the resulting abstract syntax tree is rooted in an Alf Unit Definition obtained by calling the *getUnit* method, which in turn always contains an Activity Definition (which may be obtained by calling the *UnitDefinition.getDefinition* method). If there were errors, then you can retrieve them using the *getCompilerErrors* method. A single *AlfCompiler* instance my be used to compile multiple Alf code texts, for the same or different context elements.

Alf abstract syntax classes are found in sub-packages of the package *org.modeldriven.alf. syntax,* which is from the Alf Reference Implementation, with source available here. These abstract syntax classes allow the abstract syntax tree to be navigated based on the structure of the abstract syntax metamodel defined in the Alf specification. For instance, the value of the *definition* property of a *UnitDefinition* is obtained by calling *getDefinition,* the *name* of that definition is obtained using *getName,* etc. (Note that use of the Reference Implementation outside of MagicDraw tooling is subject to its separate open-source licensing terms.)

For the purposes of triggering automatic compilation, the Alf compiler also manages a record of the dependencies of Alf code in a model on other model elements. After successfully parsing some Alf code, if you want to update the dependency record based on the parse, you need to call the *AlfActionUtil. registerDependencies* method. If you do multiple parses with the same *AlfCompiler* instance, then you need to call *registerDependencies* after each parse operation.

```
AlfCompiler compiler = new AlfCompiler();
compiler.setContextElement(element);
compiler.parse(AlfElementUtil.getAlfBody(element));
if (compiler.isSuccessful()) {
        AlfActionUtil.registerDependencies(compiler);
        UnitDefinition unit = compiler.getUnit();
        ...
} else {
        for (CompilerError error: compiler.getCompilerErrors()) {
                ...
        }
}
```