Revisiting Visitors for Modular Extension of DSMLs - Artifacts

Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs van der Storm and Olivier Barais

The *Revisitor* pattern is a language implementation pattern that enables independent extensibility of the syntax and semantics of metamodel-based DSLs, with incremental compilation and without anticipation. It is inspired by the Object Algebra design pattern and adapted to the specificities of metamodeling.

On top of the *Revisitor* pattern, we introduce *ALE*, a high-level language for semantics specification of metamodels that compiles to *Revisitors* to support separate compilation of language modules.

ALE is tightly integrated with the Eclipse Modeling Framework (EMF) and relies on the Ecore meta-language for the definition of the abstract syntax of DSLs. Operational semantics is defined with *ALE* using an open-class-like mechanism. *ALE* is bundled a set of Eclipse plug-ins.

Artifacts

- MODELS'17 paper: http://gemoc.org/ale/revisitors/paper.pdf
- ALE Compiler: https://github.com/manuelleduc/ale-compiler/
- ALE Examples & Benchmarks: https://github.com/manuelleduc/ale-compiler-benchmarks/

ReMoDD: http://www.remodd.org/v1/content/ale-compiler-and-benchmarks

Instructions

Ale compiler

Installation

1. Download an Eclipse IDE for Java and DSL Developers (Neon.3) for your platform: http://www.eclipse.org/downloads/packages/eclipse-ide-java-and-dsl-developers/neon3
2. Install the ale plugins using the update site: http://gemoc.org/ale/revisitors/updatesite/. Follow the procedure and select every plugin available.
3. Restart your Eclipse environment
4. You're all set!

Usage

The ALE plug-ins provide two main operations to the user:

- On an Ecore (*.ecore) metamodel: `Right click -> ALE -> Generate Revisitor interface` generates the corresponding *Revisitor interface* in the *src* directory of the current project
- On an ALE (*.ale) file: `Right click -> ALE -> Generate Revisitor implementation` generates the corresponding *Revisitor* implementation in the *src* directory of the current project
- (note that *Revisitor* implementations depend on *Revisitor* interfaces and will not compile otherwise)

Building the updatesite

The Ale updatesite can be rebuild using ale.p2updatesite.

To do so import the projects from this repository in an Eclipse IDE for Java and DSL Developers (Neon.3) (available here) workspace. Then open ale.p2updatesite/site.xml and click "Build All". Wait for the build to finish, you can now use the produced artifacts as an eclipse updatesite of Ale.

Ale Examples & benchmarks

This repository is structured as follows:

- examples: Toy examples demonstrating semantics definition and language modularity in ALE
- fUML: An implementation of fUML using ALE inspired by the Model Execution Case of the Transformation Tool Contest 2015 (TTC'15).

Playing with the examples

1. Setup an ALE environment following the installation instructions
2. Clone this repository locally, eg. `git clone https://github.com/manuelleduc/ale-compiler-benchmarks`
3. Import all the projects contained in the `examples` directory in Eclipse ..* `File -> Import -> Existing Projects into Workspace` ..* Select the `examples` directory as root directory in the dialog ..* Check all the projects ..* Finish

- Each project contains a launch configuration that can be used to run it
- To re-generate the *Revisitor* interfaces: `Right click -> ALE -> Generate Revisitor interface` on an Ecore file generates the corresponding *Revisitor interface* in the *src* directory of the current project
- To re-generate the *Revisitor* implementations: `Right click -> ALE -> Generate Revisitor implementation` on an ALE file generates the corresponding *Revisitor* implementation in the *src* directory of the current project

Running the benchmarks

This repository contains benchmarks comparing different implementations of the execution semantics of fUML. The concrete semantics code is common to all implementations: the only variation is the pattern used to implement it.

- Interpreter: The reference implementation of TTC'15 following the Interpreter pattern
- Visitor: An implementation following the classical Visitor pattern
- EMF Switch: An implementation using the Switch mechanism of EMF
- MonolithicRevisitor: A first *Revisitor* implementation where the runtime concepts of the activity diagram (Tokens, Offers, etc.) are already merged in a single metamodel
- ModularRevisitor: An alternative *Revisitor* implementation based on a static metamodel defining the abstract syntax of activity diagrams and another metamodel defining the runtime concepts

The fUML/activitydiagram contains the reference implementation of activity diagrams from TTC'15, plus a variant where the static concepts and the runtime concepts are modularly split in two different metamodels.

- For convenience, we provide pre-compiled JARs for all the projects and a Bash script that runs all of the benchmarks one after the other:

1. Navigate to the ./fUML/benchmarks directory
2. Run the benchmarks: `./benchmark.sh` or `benchmark.bat`

- Otherwise, import all the Eclipse projects contained in the fUML directory and wait for all of them to compile without error
- Execute the BenchmarkGeneric class of the benchmark project of your choice (one per implementation folder). BenchmarkGeneric's main function expects 3 parameters:

1. The path to a folder with the *.xmi models of the benchmark
2. The name of the test to run (testperformance_variant1, testperformance_variant2, or testperformance_variant3)
3. A prefix for the *.csv file that will store the results

Each benchmark executes every performance test of the TTC'15 contest 500 times after 50 warmups everytime.