EVL Visual Builder (integrazione con utility di porting)

EVL Visual Builder è il plug-in Eclipse per la generazione di codice Epsilon (EVL, EPL, EWL).

Per l'utilizzo è necessario popolare una base di dati MySql con le funzioni, le threshold e i re factoring che si vogliono inserire durante la generazione di codice EVL.

Lo schema della base di dati è disponibile nel seguente repository:

https://github.com/SEALABQualityGroup/visual_builder_pa_formulae.git

nel file: visual_builder_pa_formulae/examples/**Visual Builder DB Schema.txt**

Nella stessa posizione è disponibile un dump del database popolato, utilizzabile per testare il plug-in (i valori presenti nelle tabelle sono causali, per cui non fanno riferimento a nessun dominio specifico).

Prima dell'integrazione il plug-in consentiva:

- Creazione di un nuovo progetto
- Salvataggio del progetto in formato XML
- Apertura di un progetto XML
- Export come file EVL

Sono state aggiunte le funzionalità:

- Porting di un file EVL verso EPL
- Porting di un file EVL verso EWL

Le funzionalità sono state implementate nei Listener:

visual_builder_pa_formulae/src/listeners/**port2eplListener.java**

visual_builder_pa_formulae/src/listeners/**port2ewlListener.java**

invocati da un menu grafico SWT in visual_builder_pa_formulae/src/view/**Gui.java**

Selezionando dal menu uno dei due porting, una finestra di dialogo richiederà di scegliere il file EVL da convertire, a seguire un'altra finestra di dialogo richiederà la posizione in cui salvare il file convertito.

L'utility di porting è disponibile al repository: https://github.com/SEALABQualityGroup/performance-driven-software-model-refactoring-framework.git nel package: "it.spe.disim.epsilon.porting", che è stato inserito come dipendenza nel MANIFEST di EVL Visual Builder.

Inoltre affinché i metodi "ast2file" e "evl2exl" fossero visibili a Runtime, è stato necessario dichiarare i corrispondenti package "it.spe.disim.epsilon.porting.util" e "it.spe.disim.epsilon.porting.evl2exl" come Exported Packages nel MANIFEST dell'utility di porting.

A livello implementativo le finestre di dialogo restituiscono due oggetti Java di tipo File, il primo rappresenta il file EVL, che verrà parsato e convertito, il secondo viene passato come parametro all'overload del metodo ast2file(AST ast, File file) della classe PortingUtil.java nel package "it.spe.disim.epsilon.porting.util", e rappresenta il file dove verrà memorizzato il codice EPL o EWL.

EVL Validation (salvataggio stati intermedi)

Le modifiche effettuate non richiedono nuove azioni da parte dell'utente durante una validazione EVL con Epsilon. L'esecuzione rimane la stessa, quindi si lancia una Run configuration, si applicano i refactoring e quando la sessione di validazione viene terminata, il modello rifattorizzato viene sovrascritto al modello UML originale.

In realtà con le nuova modifiche al codice, quando si avvia una nuova sessione di validazione EVL, viene automaticamente creata una nuova directory chiamata "Refactoring_del_<timestamp>" all'interno della cartella dove è contenuto il modello originale. Questa cartella sarà popolata all'atto della creazione con un backup del modello originale e in seguito con i modelli UML intermedi generati ad ogni fix applicato sul modello. Questi modelli saranno contenuti in sotto-cartelle rinominate nella forma "<Step_N°>---<Componente>---<Fix>".

Le classi di epsilon modificate sono org.eclipse.epsilon.emc.emf.AbstractEmfModel.java e org.eclipse.epsilon.evl.dt.views.ValidationView.java

Il comportamento di default del metodo store() contenuto in AbstractEmfModel.java è di salvare il modello UML nel percorso memorizzato in formato URI nella rappresentazione a runtime del modello, cioè un oggetto di tipo org.eclipse.emf.ecore.resource.Resource.java denominato modelImpl.

Il nuovo metodo denominato _store_current_model(String reason) permette di salvare i modelli intermedi evitando di sovrascrivere quello originale, operazione che invece avviene solo al termine della validazione. Per far questo il metodo modifica temporaneamente l'URI contenuto nel modello inserendo il nuovo percorso di salvataggio costruito nella forma spiegata prima.

Nella classe ValidationView.java il metodo viene invocato due volte: la prima in fase di avvio della validazione, all'interno del metodo:

```
98    public void fix(final IEvlModule module, ValidationViewFixer fixer) {
99
100       if (this.fixer != null) {
101           setDone(true);
102       }
103
104       this.fixer = fixer;
105       this.module = module;
106       PlatformUI.getWorkbench().getDisplay().asyncExec(new Runnable() {
107           public void run() {
108               viewer.setInput(module.getContext().getUnsatisfiedConstraints());
109               setDone(!existUnsatisfiedConstraintsToFix());
110
111               //Model backup
112               Calendar calendar = Calendar.getInstance();
113               SimpleDateFormat sdf = new SimpleDateFormat("dd-MMM-yyyy_HH-mm-ss");
114               AbstractEmfModel model = (AbstractEmfModel)module.getContext().getModelRepository().getModels().get(0);
115               model.setSession(sdf.format(calendar.getTime()));
116               String relative_path = model.getName() + "_backup";
117               model._store_current_model(relative_path);
118           }
119       });
120   }
```

dopo aver individuato tutti gli UnsatisfiedConstraints, salverà anche un backup del modello.

La seconda nel metodo run() della classe PerformFixAction che si occupa di applicare i refactoring.

```
192         public void run() {
193 
194             //PlatformUI.getWorkbench().getDisplay().asyncExec(new Runnable() {
195 
196             //  public void run() {
197                     try {
198                         fixInstance.perform();
199                         unsatisfiedConstraint.setFixed(true);
200                         setDone(!existUnsatisfiedConstraintsToFix());
201                         viewer.refresh();
202 
203                         List<UnsatisfiedConstraint> UC = module.getContext().getUnsatisfiedConstraints();
204                         int counter = 0;
205                         for (UnsatisfiedConstraint uc : UC)
206                         {
207                             if (uc.isFixed())
208                             {
209                                 counter++;
210                             }
211                         }
212 
213                         //Save the one-fix model
214                         AbstractEmfModel model = (AbstractEmfModel)module.getContext().getModelRepository().getModels().get(0);
215                         String relative_path = "Step_" + counter + "---" + unsatisfiedConstraint.getMessage() + "---" + fixInstance.getTitle();
216                         relative_path = StringEscapeUtils.escapeXml(relative_path.replace('<', ' ').replace('>', ' '));
217                         model.store_current_model(relative_path);
218 
219                     } catch (Exception e) {
220                         module.getContext().getErrorStream().println(e.toString());
221                     }
222             //}
223 
224             //});
225 
226         }
```

Per identificare l'ordine in cui sono stati applicati i refactoring, ho utilizzato un 'counter' che conta il numero di UnsatisfiedConstraints già risolti.