# Model Driven Software Development con Eclipse, StatechartUMC *

Aldi Sulova
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo - CNR"
Via G. Moruzzi 1, 56124 Pisa, Italy
aldi.sulova@isti.cnr.it

21 dicembre 2009

*Parte del Progetto EU Sensoria IST-2005-016004 e del progetto PRIN 2007 D-ASAP

# Indice

# 1 Introduzione

L'obiettivo principale di StatechartUMC è di semplificare la creazione dell'istanza di input per il model checker UMC. Non formalmente, possiamo dire che un'istanza di input per UMC viene espressa come un modello a nodi ed archi basato su statecharts UML. A questo punto si può pensare ad uno strumento che modella graficamente le istanze e a partire da queste genera il codice testuale eseguibile da UMC. L'idea di base è quella di definire un modello attraverso diagrammi dai quali poter eseguire operazioni di verica e validazione e naturalmente ottenere completamente o parzialmente il codice. Sicuramente creare un diagramma è più semplice che scrivere il corrispondente codice testuale.

L'applicazione dell'approccio Model Driven Software Development allo strumento risulta essere la scelta più ragionevole. L'MDSD è un tentativo di portare il processo di sviluppo software verso un livello più alto eliminando quasi completamente la scrittura del codice e inserendo un concetto nuovo di "sorgente ": il modello. Per realizzare gli obiettivi sono utilizzati tre frameworks open source della piattaforma Eclipse: EMF [1], GMF [5] ed oAW4 [6].

La funzionalità principale di EMF è di ricevere in input il modello di dominio (Domain Model, il meta-modello) e di fornire come output una serie di classi Java completamente implementate, che realizzano i vincoli, le relazioni e le associazioni descritte nel modello di partenza. Il primo passo nel processo di sviluppo dell'applicazione e la definizione del meta-modello, quindi l'identificazione delle entità principali del dominio che naturalmente, parlando di statecharts, saranno gli stati e le transizioni. Per modellare il dominio, EMF mette a disposizione un'editor grafico con una notazione molto simile all'UML.

Il passo successivo è la connessione tra EMF e GMF. Lo scopo di GMF e facilitare lo sviluppo di istanze grafiche del meta-modello, creando editor grafici dotati di funzionalità quali drag & drop, copia/incolla, undo e redo: una classica applicazione GMF, ad esempio, è un editor che consente di disegnare diagrammi di vario tipo, con la possibilità di collegare tra loro le figure, ridimensionarle e spostarle. In sostanza con GMF le entità identificate nella prima fase hanno anche una rappresentazione grafica, dove le transizioni connettono tra loro gli stati. Un diagramma con stati e transizioni definisce un istanza di input per UMC.

Dalla rappresentazione grafica si può arrivare al codice testuale utilizzando il framwork messo a disposizione dal progetto Open Architecture Ware (oAW4) di Eclipse. oAW4 è un framework che definisce varie funzionalità per i modelli generati a partire da un meta-modello EMF. Sostanzialmente fornisce 3 linguaggi testuali che sono utili in diversi contesti: Check (.chk), per la verifica della correttezza del modello, Xpand (.xpt), per controllare l'output del processo di generazione ed Xtend (.ext), per definire librerie con operazioni generali, utilizzabili da Check ed Xpand. Xtend viene utilizzato anche in un contesto di trasformazione di modelli. La verfica della correttezza risulta essere un attività molto utile nel processo di definizione del modello. In una situazione normale, un'istanza di input, può avere un numero elevato di entità grafiche, quindi è molto importante mantenere una certa coerenza con il modello di dominio.

# 2 Strutttura dei modelli UMC

Un modello UMC è descritto mediante tre principali entità:

- le classi,
- gli oggetti,
- le regole di astrazione.

La classe rappresenta un modello che gli oggetti attivi o non attivi del sistema, realizzano. Nel caso di un oggetto attivo uno *statechart diagram*, associato alla classe, descrive il suo comportamento dinamico. Per le regole di astrazione si può dire che non hanno un ruolo particolare nel modello del sistema da descrivere. Il loro obbiettivo è semplicemente quello di mantenere una traccia sull'evoluzione di proprietà legate al sistema di interesse e verificabili tramite il *model checking*.

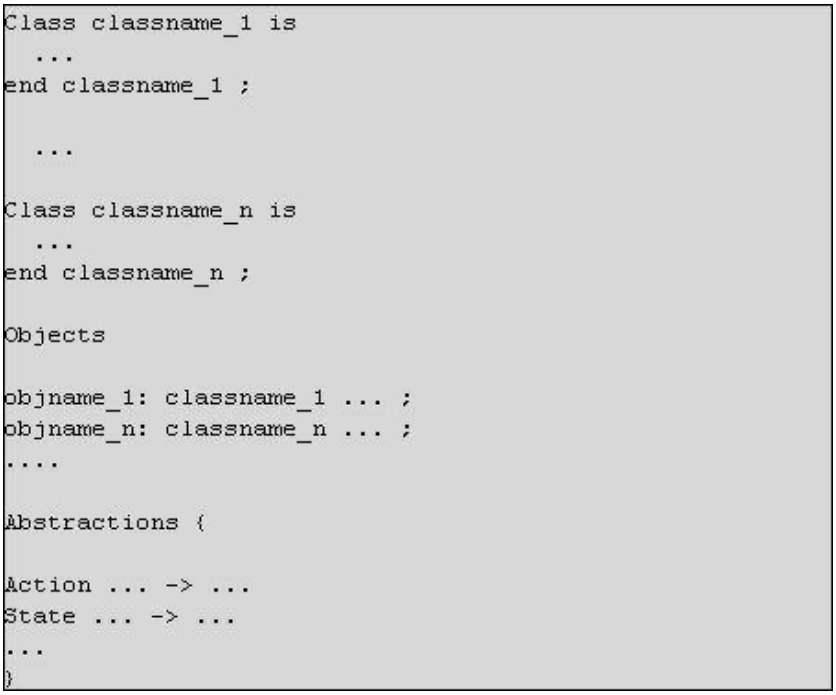


Figura 1: *Struttura dei modelli UMC*

## 2.1 Classi e oggetti

Il comportamento degli oggetti appartenenti ad una classe è definito da una *statechart diagram* associata alla classe stessa.
In particolare, la definizione di una class statechart diagram consiste nell'introdurre le seguenti proprietà:

- nome della classe,
- l'elenco degli eventi che attivano le transizioni degli oggetti della classe (segnali o call operations),
- l'elenco degli attributi, ovvero le variabili locali dell'oggetto,
- la struttura degli stati sul quale può transire, ovvero i nodi del statechart diagram,
- le transizioni, ovvero gli archi del statechart diagram.

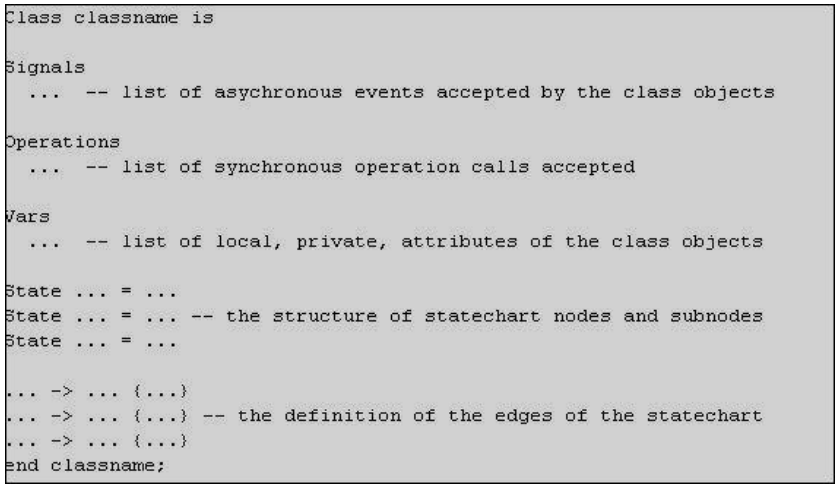


Figura 2: *Struttura delle classi UMC*

Nel caso di oggetti non attivi, la corrispondente classe ha definito solo la lista dei segnali e operazioni.

Gli eventi gestiti dalla classe si distinguono in segnali sincroni e operazioni call asincroni, gli ultimi possono anche avere un tipo di ritorno che può essere "void ", "int ", "bool ", "obj "oppure il nome di una classe. La figura 3 mostra la struttura dei segnali, la 4 quella delle operazioni.

```
-- signal with no parameters
signal_name;

-- signal with n parameters
signal_name (arg_1, ... , arg_n: type_n);
```

Figura 3: *Segnali*

```
-- operation with no parameters and no result value
op_name;

-- operation with n parameters and result type
op_name (arg_1, ... , arg_n: type_n): result_type;
```

Figura 4: *Operation call*

La sezione "Vars "definisce gli attributi, ovvero le variabili locali privati all'oggetto della classe. Possono essere definiti esplicitamente con i tipi, dove i tipi sono "int ", "bool ", "obj "oppure un nome di una classe. La loro struttura è definita nella figura seguente.

```
-- untyped, initialized by default, multiple local variable
varname1; varname2; varname3;

-- expicitly typed local variable
varname: int;

-- explicilty typed, expliciltly initialized, local array
varname1: int[] := [1,2,3];
```

Figura 5: *Variabili*

La struttura di una statechart consiste nella definizione di una sequenza di stati, che iniziano con la definizione dello stato *Top*, il quale deve essere un *Composite Sequential State*. Un Composite Sequential State contiene una lista di sottostati: stati sequenziali, stati paralleli oppure stati semplici.

```
State parentstate = substate_1, substate_2, ... , substate_n
```

Figura 6: *Stati sequenziali*

Un *Composite Parallel State* è definito come una composizione di una serie di stati sequenziali chiamati anche "regioni "dello stato parallelo.

Con "Defers "si può specificare una lista di eventi da attivare quando lo stato sarà attivato.

La definizione di uno stato composto deve precedere la definizione di un suo sottostato.

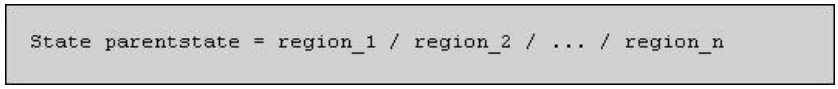


Figura 7: *Stati paralleli*

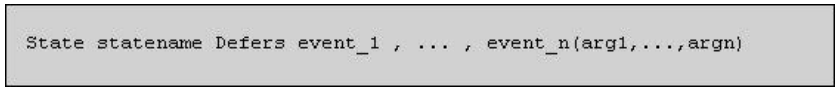


Figura 8: *State Defers*

Nella sezione "Transitions "sono definite le transizioni dello statechart che modella il sistema. La definizione di una transizione contiene uno stato d'origine, un trigger, una guardia opzionale, un elenco delle azioni e uno stato target.


Figura 9: *Transizioni*

Il trigger può essere un evento, definito nella sezione eventi dello statechart, oppure il simbolo "-"che significa l'assenza di un trigger esplicito. La guardia (se presente) è una semplice forma di espressione booleana che coinvolge gli attributi dell'oggetto.

Le azioni attualmente supportate da UMC sono: assegnamenti, invii di eventi, condizionali e loop finiti.

- Gli assegnamenti hanno la forma seguente: *right_side* può essere un espressione oppure un operation call e *varname* deve essere il nome di una variabile, il nome di uno dei parametri della trasnsition trigger oppure il nome di una variabile locale alla transizione.

```
varname := right_side; -- assignment to local variable
varname[index]:= right_side; -- assignment to a component of vector
```

Figura 10: *Assegnamenti*

- Invio asincrono di segnali. *target_object* deve essere un nome che deno-

```
target_object.signal_name(expr_1,...,expr_n);
```

Figura 11: *invio di segnale assincrono*

  ta un oggetto che fa parte nella definizione del sistema e *signal_name* deve essere un segnale dichiarato nella sezione *Signals* della classe a cui appartiene *target_object*. L'espressione e il segnale devono avere lo stesso numero di parametri.

- Chiamata sincrona. *operation_name* deve essere il nome di un opera-

```
target_object.operation_name(expr_1,...,expr_n); -- plain operation call
varname := target_object.operation_name(expr_1,...,expr_n); -- function call
```

Figura 12: *chiamata sincrona*

  zione dichiarato nella sezione *Operations* della classe a cui appartiene *target_Object*.

- Conditional and finite loops.

```
if condition then { actions-list } else { actions-list };
if condition then { actions-list };

for iterator in min_expre .. max_expr{ actions-list };
```

Figura 13: *Conditional and finite loops*

Una volta definito il comportamento delle classi tramite lo statechart, possiamo definire l'evoluzione del sistema come un insieme di istanze di oggetti. Gli oggetti hanno un nome, il nome della classe che lo definisce e i valori iniziali per i suoi attributi.

```
object_name: class_name    -- an object declaration with initiaizations
        ( obj_attribute_1 => initial_value_1,
                ...,
          obj_attribute_n => initial_value_n);

object_name: class_name; -- an object declaration without initiaizations
```

Figura 14: *Nuove istanze di oggetti*

Le regole di strazione sono definite all'interno del blocco:

```
Abstractions {
     ...
}
```

Figura 15: *Regole di astrazione*

e in particolare, come introdotto all'inizio di questo capitolo, non hanno un ruolo nella definizione del modello. Il loro obbiettivo è semplicemente quello di mantenere una traccia sull'evoluzione di proprietà legate al sistema di interesse e verificabili tramite il *model checking*. Per una descrizione dettagliata delle regole di astrazione vedi Franco Mazzanti, *Designing UML Models with UMC (ref. UMC V3.6 build p-April 2009)* [1].

## 2.2 Esempio

```
Class Classname is                                  // class definition
Signals:
   sig4(z:Classname)

Operations:
   op1(x:int), op2:int, op3(z:Classname,w:int):bool

Vars: v1:int :=0, v2:Classname, v3:bool := True;     -- local attributes

State Top= R1 / R2
State R1 = s1,final
State R2 = s2, final
State Top Defers sig4(z)

Transitions:
 R1.s1 -> R1.final
  { -[v1=0] /
   v1:=3;
   v2.sig4(self)
  }
 R2.s2 -> R2.final
  { sig4(z)[v1>0] /
    v1:=v1-1;
    v2.sig4(self);
    OUT.print(v1)
  }
end Classname;

Objects:
  Obj1: Classname (v2 -> Obj1)       // static object instantiation
  Obj2: Classname (v2 -> Obj2)       // static object instantiation
  JustAName, OrThisName: Token;      //  Token is a predefined empty class

Abstractions {
  Action:  $1($*)          -> $1($*)         -- observe events with all their params
  Action:  lostevent($1)   -> discarded($1)  -- observe discarding of triggers
  Action:  accept($1)      -> accepted($1)   -- observe dispatching of triggers
}
```

Figura 16: *Esempio*

# 3 Tecnologie utilizzate

In questo capitolo vengono introdotte le tecnologie utilizzate per lo sviluppo dell'editor grafico.

## 3.1 Eclipse Modelling Framework

Il progetto EMF [4] parte con l'obiettivo di mettere a disposizione degli sviluppatori una piattaforma sulla quale sia possibile implementare il *Model Driven Software Development*. Gli vantaggi di questa tecnica, dove il concetto di base è il modello, sono:

- utilizzare il modello, UML in generale, per documentare le applicazioni,
- trasformazioni model-to-model e model-to-code,
- l'utilizzo dei modelli e delle trasformazioni per migliorare il processo di sviluppo software e l'integrazione tra le varie applicazioni.

Ritornando su EMF diciamo che è un framework che prende in input il modello del dominio espresso mediante un linguaggio di modellazione, tipicamente UML, e fornisce come output una serie di classi Java completamente implementate, che realizzano i vincoli, le relazioni e le associazioni descritte nel meta-modello di partenza. A partire da queste classi java generate si possono creare editor grafici completi per la propria applicazione software.

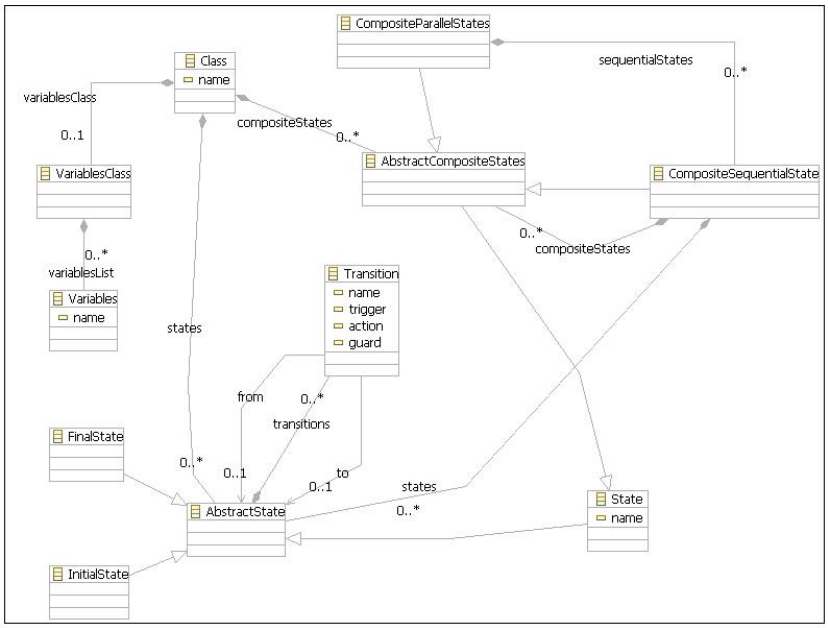


Figura 17: *umc_model.ecore*

Inizialmente si definisce il meta-modello *umc_model.ecore*, utilizzando una notazione molto simile al linguaggio UML. Il meta-modello rispecchia la grammatica UMC definita nel paragrafo 1.1.

## 3.2 Graphical Modelling Framework

L'obiettivo del progetto GMF [5] è di estendere il framework EMF con la possibilità di esprimere le istanze della *domain model* tramite editor grafici. Praticamente, GMF fornisce un metodo che a partire dal meta-modello *umc_model.ecore* del dominio dellapplicazione ti permette di definire una serie di altri modelli intermedi e di generare tutte le classi dell'editor grafico senza scrivere esplicitamente nessuna riga di codice. I modelli intermedi sono 3 e vengono descritti nei paragrafi seguenti.

### 3.2.1 umc_model.gmfgraph

Nel file *.gmfgraph* vengono definite le entità grafiche che rappresentano gli componenti sintattici di UMC. Nella figura 17 si può vedere come lo stato sia modellato tramite un rettangolo con il colore del bordo specificato nella

voce *Foreground*. Inoltre, la parte superiore del rettangolo contiene anche una label dove viene visualizzato il nome dello stato.

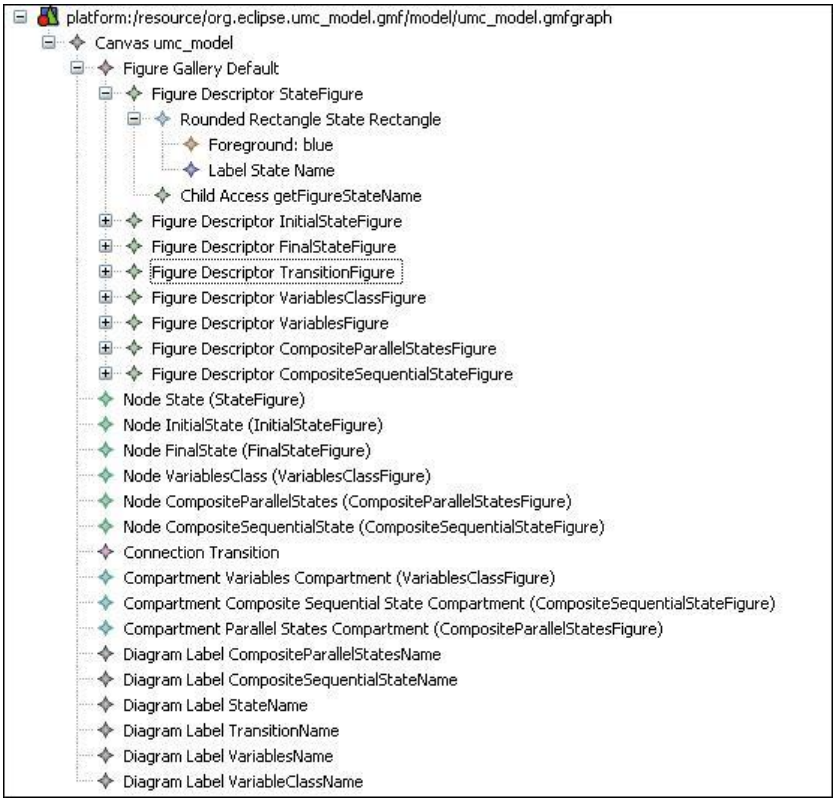


Figura 18: *umc_model.graph*

### 3.2.2 umc_model.gmftool

Nel file *.gmftool* vengono definiti i componenti della paletta. La paletta è l'entit grafica visualizzata nella parte destra dell'editor. Contiene i bottoni per creare nuovi oggetti grafici.

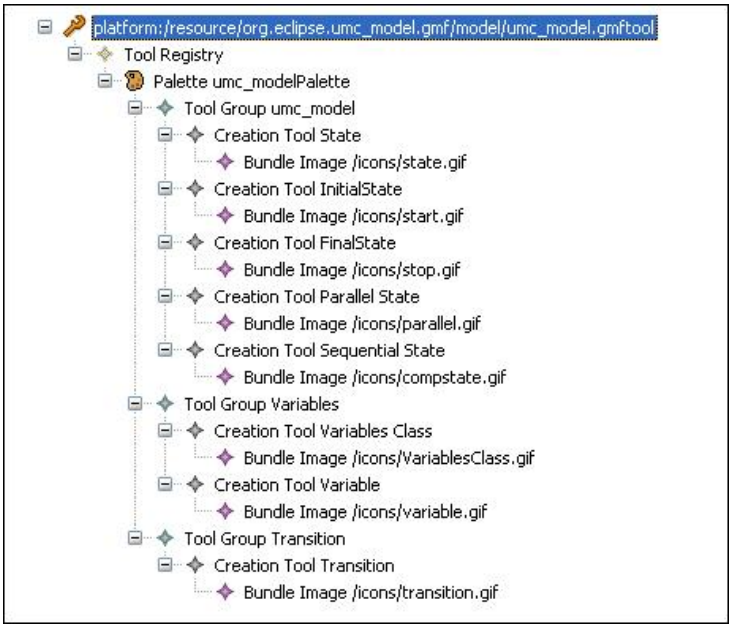


Figura 19: *umc_model.gmftool*

### 3.2.3 umc_model.gmfmap

Nel file *.gmfmap* viene fatto il mapping tra i componenti grafici definiti nel file *umc_model.gmfgraph* e i componenti della paletta definiti nel file *umc_model.gmftool*.

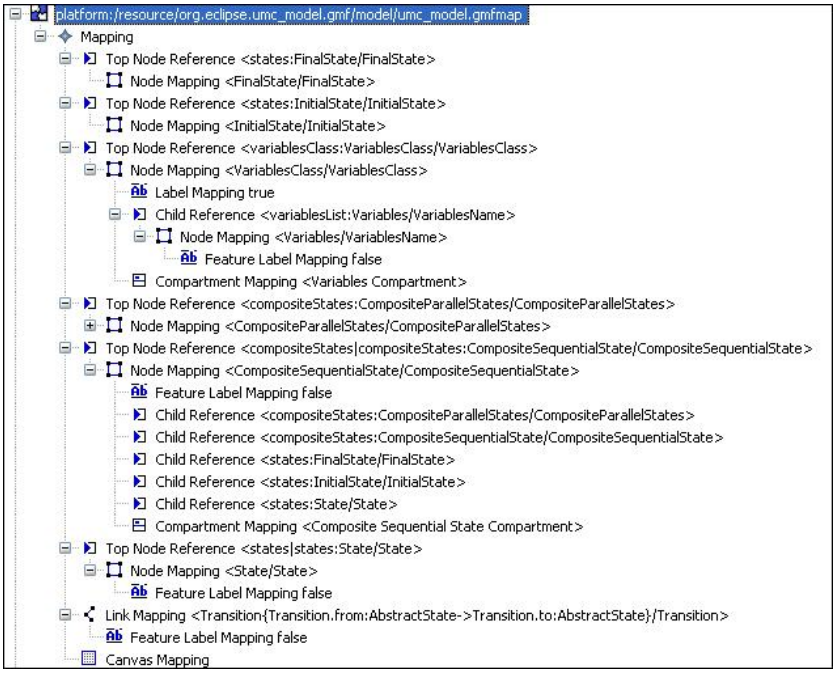


Figura 20: *umc_model.gmfmap*

## 3.3 Open Architecture Ware

OAW4 [6] è uno strumento utile nel contesto dell'implementazione della tecnica Model Driven Software Development per lo sviluppo delle applicazioni software. In particolare, OAW4 mette a disposizione una piattaforma per generare codice a partire dalle istanze di una *Domain Model EMF* e trasformazione tra meta-modelli EMF. Lo strumento fornisce 3 linguaggi testuali che sono utili in diversi contesti del processo del generazione del codice:

- Check ( .chk ), per la verifica della correttezza del modello,
- Xpand ( .xpt ), per controllare l'output del processo di generazione,
- Xtend ( .ext ), per definire librerie con operazioni generali, utilizzabili da Check ed Xpand.

Inoltre, OAW4 definisce anche un file di "workflow ", entità che controlla tutti i passi necessari (load model, check model, code generating) per una corretta esecuzione del generatore.

```
<?xml version="1.0"?>
<workflow>
    <property name = "modelFile" value = "" />
    <property name = "genPath" value = "" />

    <component id = "xmiParser"
        class = "org.openarchitectureware.emf.XmiReader">
        <modelFile value = "${modelFile}" />
        <metaModelPackage value = "org.eclipse.umc_model.gmf.umc_model.Umc_modelPackage" />
        <outputSlot value = "model" />
    </component>

    <component class="oaw.check.CheckComponent">
        <metaModel id="mm"
            class="org.openarchitectureware.type.emf.EmfMetaModel">
          <metaModelPackage value="org.eclipse.umc_model.gmf.umc_model.Umc_modelPackage"/>
        </metaModel>
        <checkFile value="model::checks::Umc_modelErrors"/>
        <expression value="model.eAllContents.union({model})"/>
    </component>

    <component id  = "generator"
        class = "org.openarchitectureware.xpand2.Generator" >
        <metaModel id = "mm"
            class = "oaw.type.emf.EmfMetaModel">
            <metaModelPackage value = "org.eclipse.umc_model.gmf.umc_model.Umc_modelPackage"/>
        </metaModel>
        <expand value = "model::template::Template::Template FOR model"/>
        <outlet path = "${genPath}"/>
    </component>
</workflow>
```

Figura 21: *workflow*

### 3.3.1 Text Code Generator dell'editor StatechartUMC

Al momento della traduzione al componenente workflow vengono passate come parametri due proprietà:

- il path file del modello creato dall'utente,
- path directory destinazione del file da generare.

Inizialmente, nel componente *xmiParser*, viene letta la specifica del modello. Successivamente, nel componente *oaw.check.CheckComponent*, viene fatta la verifica del modello per proseguire con la generazione del codice nel componente *generator*. Il file è salvato nella directory specificata dalla variabile *genPath*.

La parte più significativa della workflow è: dove la definizione del modello

```
<expand value = "model::template::Template::Template FOR model"/>
```

è dato come input al file di analisi Template.ext. Il file di template, scritto nel formato .xpt, lavora sui modelli definiti dall'editor e lì traduce in un formato testuale. Per un esempio completo di traduzione vedere il paragrafo 3.3.(non lo so se e giusto)

# 4 StatechartUMC

In questa sezione descriviamo l'editor grafico e illustriamo tutti i passi necessari per un suo corretto utilizzo. Il layout grafico dell'editor è suddiviso in 4 viste principali (Figura. 21):

- Editor view, dove viene definito il modello grafico,
- Outline view, offre una visione minimizzata del modello grafico,
- Navigator view, lista dei progetti e files creati. È da notare che l'applicazione utilizza un ambiente di lavoro "workspace "dove vengono definiti i progetti e i file. Quindi, tutto quello che si crea è disponibile sotto questa directory.
- Properties view, lista delle proprietà di ciascun componente dell'editor.

Eseguire l'applicazione Statechart UMC. Nella schermata principale scegliere la voce *File - New  New Project* Un nuovo progetto verrà creato con due directory:

- *diagram*, contiene la definizione grafica del file,
- *model*, contiene la rappresentazione XML del file, basata sulla definizione del meta-modello.

Scegliere *File-New- New UMC Diagram* per creare una nuova classe per il modello grafico. Ogni file rappresenta una singola classe.

Creare il modello utilizzando gli elementi nella paletta a destra dell'editor. In seguito viene data una breve descrizione sul significato degli elementi.

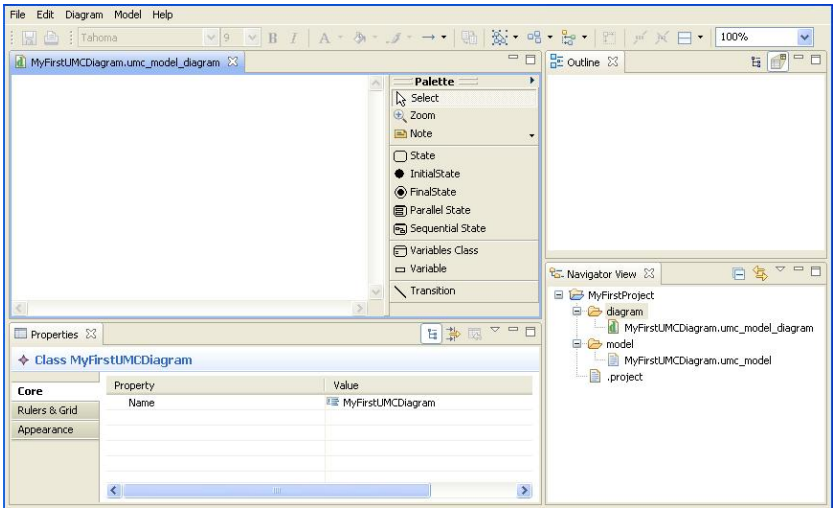


Figura 22: *Modello Statechart UMC*

## 4.1 Entità grafiche

Questa sezione descrive il modo in cui i componenti della grammatica UMC sono rappresentati nell'editor grafico.

**Class**: un singolo file nel progetto rappresenta una singola classe nel modello (Figura 22).

### InitialState, State, FinalState

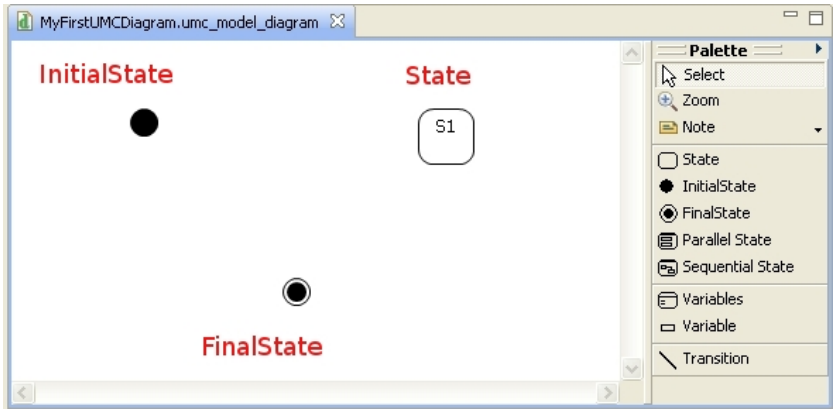


Figura 23: InitialState, State, FinalState

### Composite Sequential State

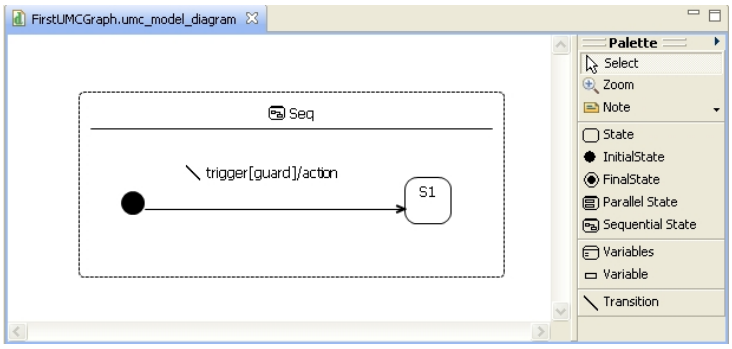


Figura 24: Composite Sequential State

### Parallel State

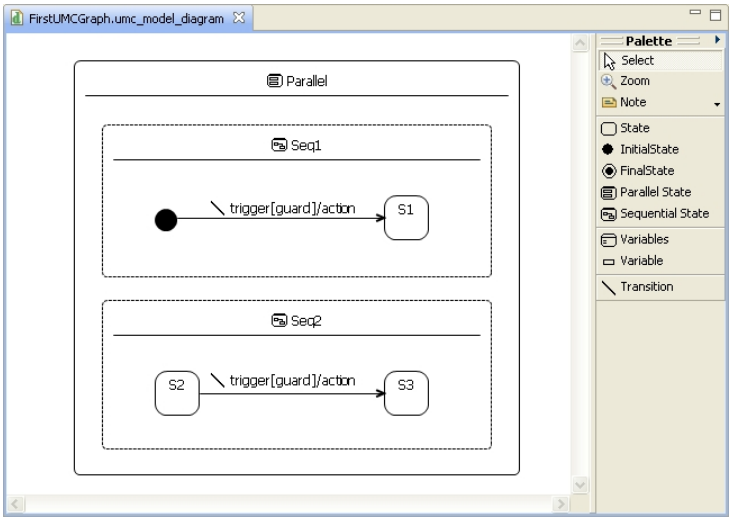


Figura 25: Parallel State

### Transition

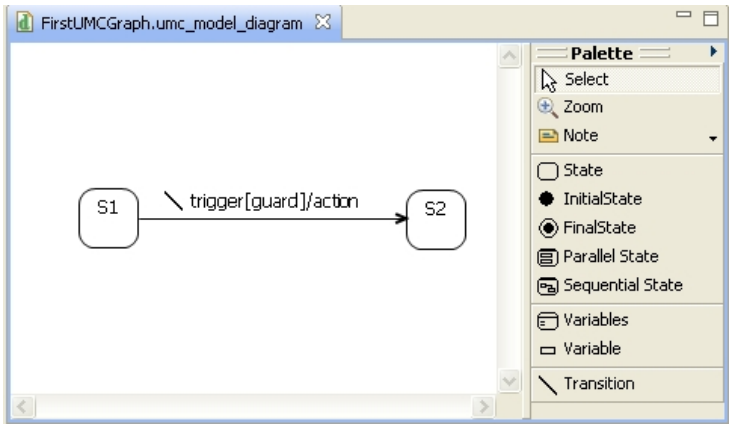


Figura 26: Transition

### Variables

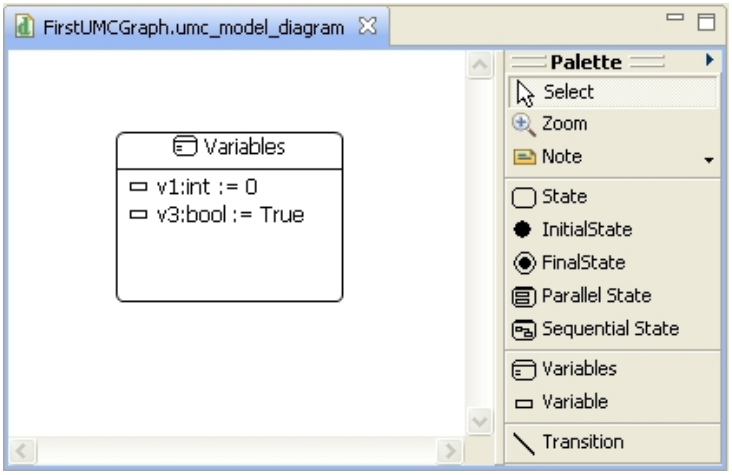


Figura 27: Variables

Nella figura seguente viene mostrato un esempio semplice di uno statechart diagram UMC.

**UMC Statechart Diagram**

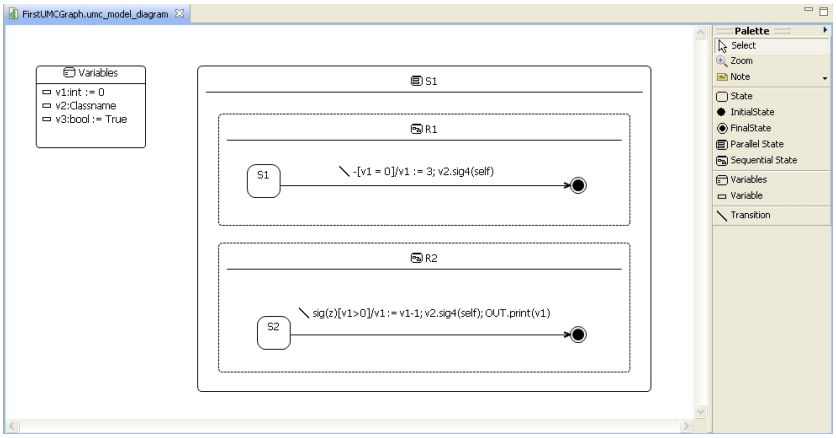


Figura 28: UMC Statechart Diagram

## 4.2 Lista degli oggetti

Oltre alle classi si deve creare anche una lista di oggetti. Nella voce File del menu selezionare *New-New Object List*. Nella finestra aperta scegliere il progetto di destinazione del nuovo file da creare e premere OK. Il file si trova nella directory "diagram "del progetto selezionato e ha l'estensione ".objectList ". Gli oggetti sono definiti con la sintassi specificata nel paragrafo 1.1.

## 4.3 Generazione del codice

Una volta definito il modello grafico si può procedere con la generazione del codice testuale che, a sua volta può essere utilizzato come input per il model checker UMC. Inizialmente si esegue la generazione in codice testuale delle singole classi. Aprire il diagramma della classe per la quale si vuole avere il codice testuale: nel menu principale scegliere *Model - Generate Text File*. Il nuovo file "nomefile.umc"è disponibile nella cartella *generated*.

Un file nel progetto rappresenta una singola classe nel modello. Per creare il file testuale finale contenenti tutte le classi create, includendo anche la

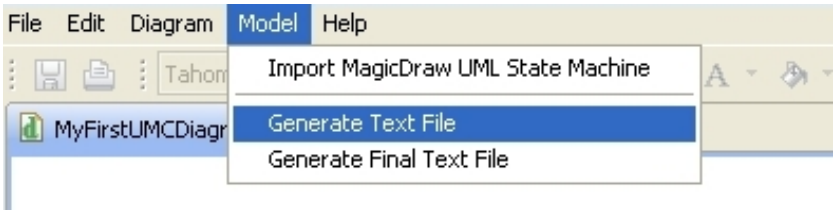


Figura 29: Generazione del codice

lista degli oggetti, si procede come segue: nella menu *Model* scegliere la voce *Generate Final Text File*, nella nuova finestra aperta selezionare il progetto per il quale si vuol avere il file testuale generato e premere *OK*. Se la generazione è andata a buon fine il file "generateFinal.umc"deve essere disponibile nella cartella "generated"del progetto selezionato.

### 4.3.1 Cosa manca nel codice generato

Ci sono casi in cui il codice testuale generato non rappresenta tutta l'informazione contenuta nello statechart diagram. In tale situazione l'utente può procedere manualmente. I casi sono:

- non si riesce a definire i tipi dei segnali,
- non si riesce a definire le operazioni.

## 4.4 Verifica della correttezza del modello

L'editor gestisce gli errori presenti nel modello in due modi:

- utilizzando la voce "Validate "nella sezione "Diagram "del menu,

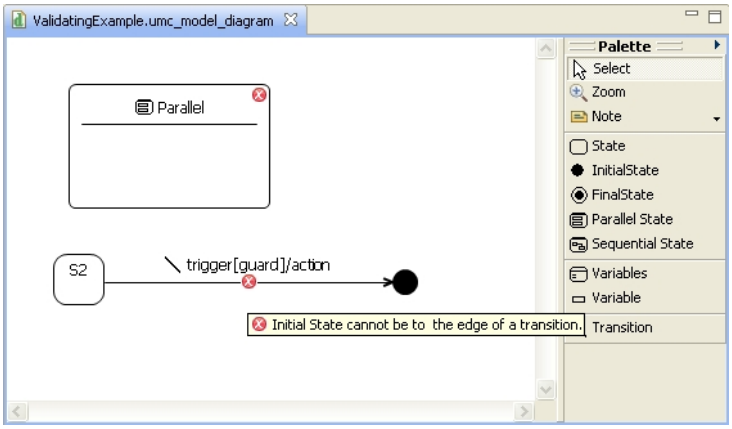


Figura 30: validazione del modello

- se la generazione del codice non è andata a buon fine una lista con gli errori viene visualizzata nel componente "Error "in basso del layout grafico dell'editor.

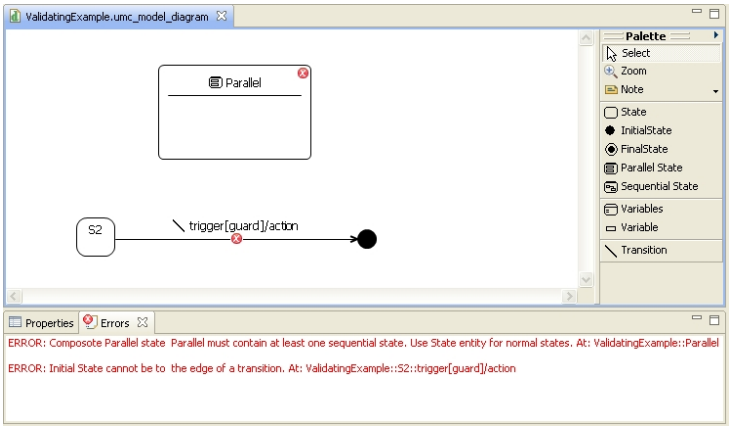


Figura 31: Errori

## 4.5 Importazione e trasformazione di modelli UML

StatechartUMC offre la possibilità di importare un statechart UML creato con Magic Draw. Nel menu *Model* scegliere la voce *Import MagicDraw UML State Machine*, nella finestra aperta scegliere il file ".uml" per il quale generare il corrispondente file ".umc_model".

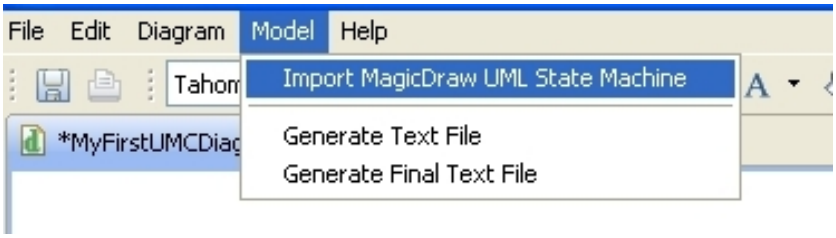


Figura 32: Import MagicDraw UML statechart

Dopo l'esecuzione del comando una nuova finestra elenca tutti i file ".umc_model" generati dalla trasformazione. Da notare che un file ".uml" può contenere diverse istanze di statecharts UML.
Da questa procedura si ottiene solo la definizione del statechart UMC. Per ottenere anche il diagramma seguire i passi seguenti: nel menu *File* scegliere la voce *Initialize UMC statechart diagram*, selezionare il file ".umc_model" e premere *Finish*. Nella schermata principale verrà visualizata la rappresentazione grafica del modello scelto.

# Riferimenti bibliografici

[1] Franco Mazzanti. *Designing UML Models with UMC (ref. UMC V3.6 build p-April 2009)*, http://fmt.isti.cnr.it/umc/V3.6/umc.html.

[2] Eclipse Project. http://www.eclipse.org/

[3] EclipseRich Client Platform. http://wiki.eclipse.org/index.php/Rich_Client_Platform

[4] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/

[5] Graphical Modelling Framework. http://www.eclipse.org/modeling/gmf/

[6] Open Architecture Ware project. http://www.openarchitectureware.org/