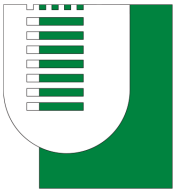

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

Corso di Laurea Magistrale in Informatica

A.A. 2022/2023

Progetto MBSE

Trasformazione da BPMN a UML con notazione SoaML

PROFESSORE

Prof. Andrea D'Ambrogio

STUDENTI

Gabriele Benedetti
Giulia Pascale

Sommario

Il seguente documento contiene il lavoro svolto per il progetto del corso Model Based Systems Engineering nell'Anno Accademico 2022-23 dagli studenti Gabriele Benedetti e Giulia Pascale. Il progetto ha riguardato la realizzazione di uno strumento software per la trasformazione di modelli BPMN in modelli UML annotati con il profilo SoaML. Il linguaggio di trasformazione utilizzato è ATL.

Indice

Capitolo 1

Model Driven Engineering MDE

Model-driven engineering è una metodologia di sviluppo software che si focalizza sulla creazione di modelli. L'obiettivo è semplificare il processo trasformando i requisiti iniziali in modelli, che a loro volta indirizzano (e quindi semplificano) l'implementazione delle soluzioni. Questo approccio diventa sempre più importante con l'avanzare delle tecnologie, poiché la complessità nello sviluppo software è in costante aumento. Modellare è essenziale per l'attività umana poiché ogni azione è preceduta dalla costruzione (implicita o esplicita) di un modello.

1.0.1 Modello

Un modello è una rappresentazione astratta, ovvero semplificata, di un sistema. Dove per sistema si intende un insieme di elementi che interagiscono.
Un modello è in grado di dare una visione parziale del sistema, cioè può mettere in evidenza alcuni aspetti e caratteristiche di esso, tralasciandone altri.

Principio di sostituibilità limitata

Il principio di sostuibilità limitata è ciò che permette la rappresentazione semplificata che ci forniscono i modelli dei sistemi. Infatti esso indica che:
Un modello M si dice rappresentazione di un sistema S, per un dato insieme di do-

mande Q, se, per ogni domanda di questo insieme Q, il modello M fornirà esattamente la stessa risposta che il sistema S avrebbe fornito rispondendo alla stessa domanda.

1.0.2 Metamodello

Il metamodello è a sua volta un altro modello che fornisce gli strumenti per capire il modello. Infatti quest'ultimo dovrà essere conforme al MetaModello. Il modello sarà di facile comprensione se userà gli strumenti del metamodello, altrimenti non sarà leggibile.

1.1 Model to model (M2M) Transformations

La Model to Model (spesso abbreviata con M2M) è la creazione automatica di modelli target partendo dai modelli source. È possibile passare da un modello ad un altro (1 to 1 transformations), da un modello ad N possibili modelli (1-to-N), da N-to-1 oppure da N-to-M. [1]

Trasformazioni In-Place e Out-Place

La trasformazione **In-place** apporta modifiche a parti specifiche del modello senza eliminarne completamente quello esistente.
La trasformazione **Out-place** crea un nuovo modello.
Di seguito due esempi illustrativi dei due tipi di trasformazioni.

Strategia e architettura In-Place Transformation

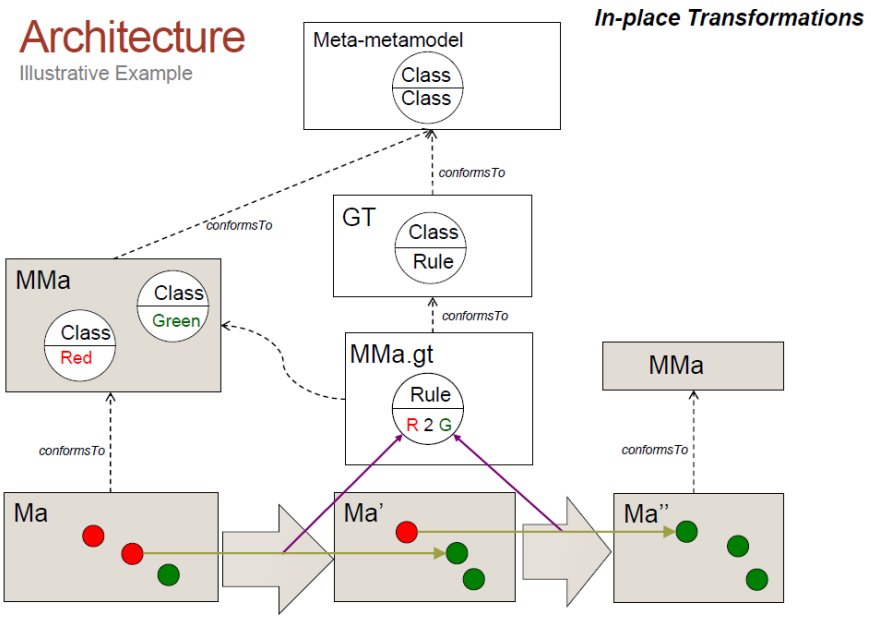

Figura 1.1: Trasformazione In-place

Strategia e architettura Out-Place Transformation

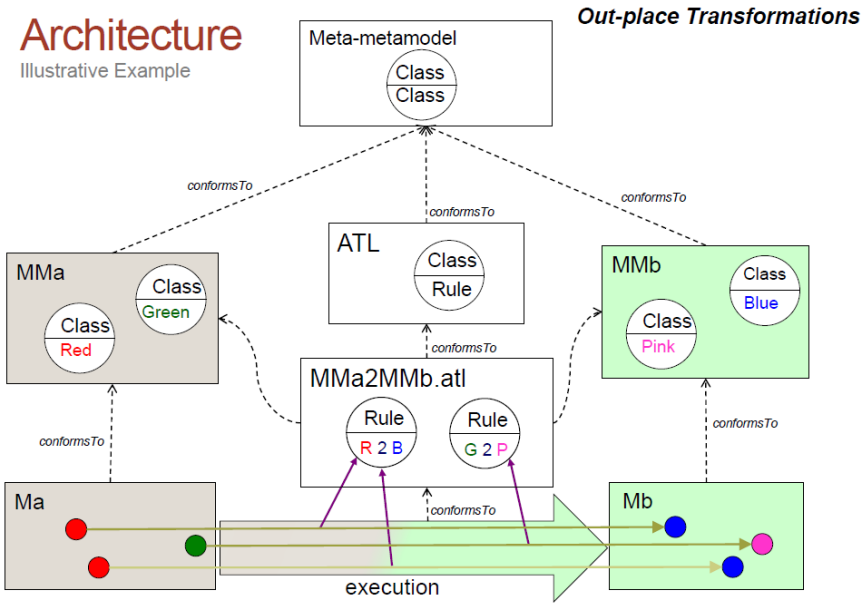

Figura 1.2: Trasformazione Out-place

1.2 Out-Place Transformation: linguaggio ATL

Il linguaggio ATL ("Atlas Transformation Language")[2] è un linguaggio di programmazione utilizzato principalmente per trasformare modelli da un formato ad un altro. È quindi usato nel contesto dello sviluppo software in cui si devono convertire i dati da una rappresentazione ad un'altra. I modelli source (sorgente) sono *read-only* e i modelli target sono *write-only*.
ATL è un linguaggio ibrido **dichiarativo-imperativo**. Alla base di ATL si ha l'uso delle *rule*, cioè le regole di trasformazione che permettono di passare dal modello *source* al modello *target*, trasformando gli elementi del primo negli elementi del secondo. In breve si può dire che:

- Per quanto riguarda la parte **dichiarativa** abbiamo:
  - le *Matched Rule* che sono *rule* che vengono applicate una volta per ogni corrispondenza dall'esecuzione e consentono agli elementi del modello *source* di "fare match" per generare da essi gli elementi corrispondenti del modello *target*.
  - *OCL* (Object Constraint Language) che è un linguaggio dichiarativo per descrivere dei vincoli e regole. In ATL sono disponibili le *OclAny Operations* per tutti i tipi di dati esistenti, fornendo un insieme standardizzato di funzionalità che può essere utilizzato per eseguire query.
- Nella parte **imperativa** abbiamo:
  - le *Called* e le *Lazy rule* che sono *rule* che vengono applicate tante volte quante vengono richiamate da altre *rule*. La differenza con le *Matched*

sta nel fatto che una *Called rule* deve essere invocata da un blocco imperativo per poter essere eseguita. Il codice imperativo può essere definito nell'*Action block* delle *Matched rule* oppure nelle *Called rule*.

- Un *Action block* è un blocco **opzionale** che può servire per effettuare operazioni all'interno della *rule*, che non siano la transformazione di un elemento.
- Gli *Helper* permettono di definire dei metodi che possono essere richiamati in punti diversi del codice.

Capitolo 2

Da BPMN a UML annotato con SoaML

2.1 BPMN

Business Process Modeling Notation (BPMN) [4] è uno standard per rappresentare graficamente un processo aziendale (e non). È come un diagramma di flusso e utilizza una grafica standardizzata per rappresentare i partecipanti, le scelte e il flusso del processo. I diagrammi sono dettagliati e facili da leggere in modo da consentire a tutti gli addetti ai lavori di comprenderli. In particolare, fornisce una notazione grafica per specificare i processi di business in un *Business Process Diagram (BPD)*, basato su una tecnica di flusso molto simile ai diagrammi di attività di Unified Modeling Language (*UML*) . Un diagramma BPMN non si traduce direttamente in alcuna implementazione specifica.

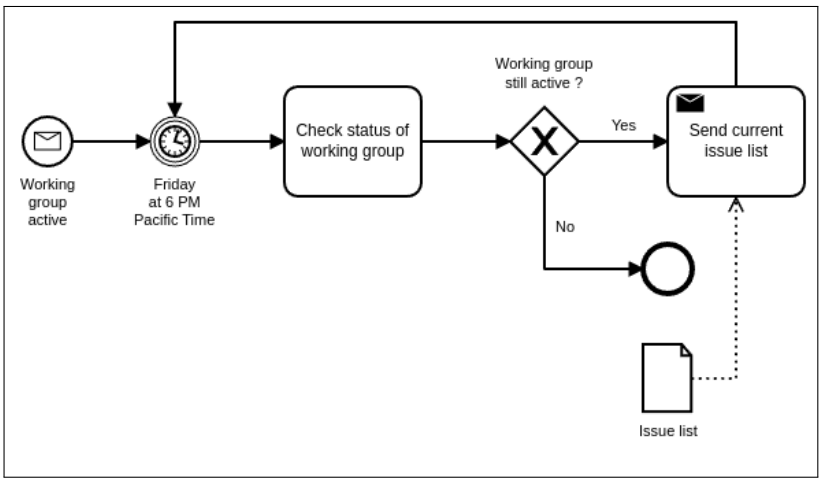

Figura 2.1: Esempio grafico BPMN

2.2 UML

Unified Modeling Language (UML) è un linguaggio di modellazione visuale usato principalmente nel campo dell'ingegneria del software per descrivere, progettare e documentare i sistemi software. È costituito da un insieme di simboli grafici e regole di notazione che consentono agli sviluppatori di rappresentare in modo chiaro e comprensibile gli aspetti strutturali e comportamentali dei sistemi software, inclusi concetti come classi, oggetti, relazioni, attività, casi d'uso, sequenze, e altro. UML fornisce un linguaggio standardizzato per la comunicazione e la comprensione dei progetti software tra i membri del team e le parti interessate.

UML Activity Diagram

Un diagramma di attività UML (Activity Diagram) è una rappresentazione grafica dei flussi di lavoro o delle attività all'interno di un sistema. È utilizzato per descrivere il comportamento sequenziale e concorrente di un processo o di una funzionalità, mostrando le azioni/attività, le decisioni prese, le condizioni e il flusso di controllo.
Gli UML AD sono utili per rappresentare in modo chiaro e comprensibile il compor-

tamento di un sistema, fornendo una visione ad alto livello dei processi senza entrare nei dettagli implementativi.

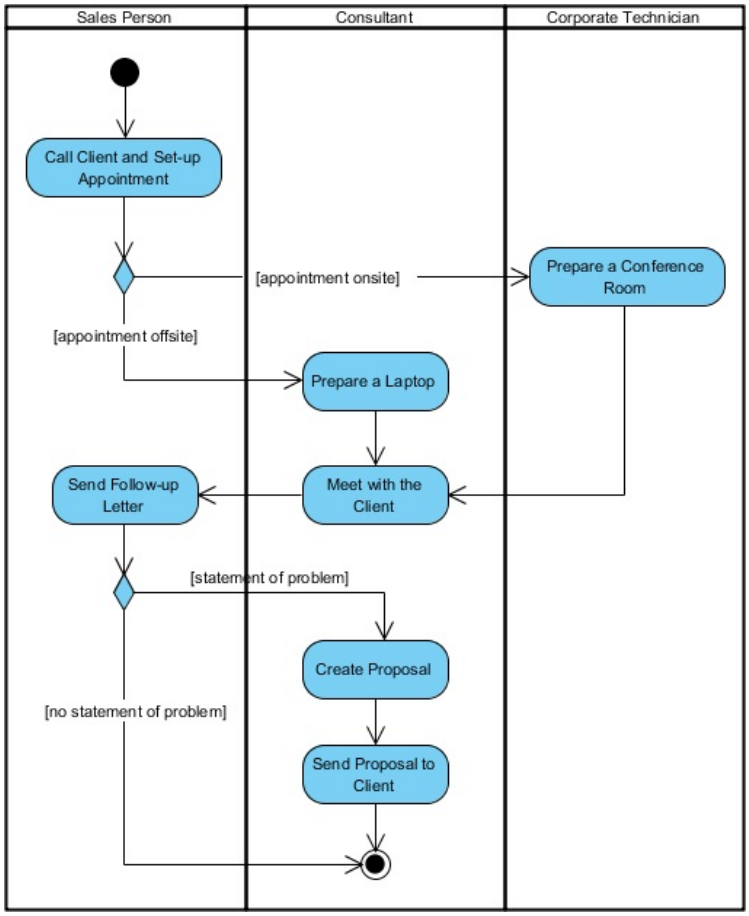

Figura 2.2: Esempio grafico UML Activity Diagram

Rapporto tra UML e BPMN

I diagrammi UML e quelli BPMN sono due notazioni per modellare processi che possono essere impiegati in modo simile. Sebbene il diagramma di attività di UML contenga un insieme più limitato di simboli rispetto a BPMN e sia meno esplicito, alcuni dei suoi elementi condividono una semantica simile con quelli presenti in BPMN.

2.3 Mapping da BPMN a UML

Per effettuare una trasformazione da un modello ad un altro, occorre definire un *mapping* degli elementi, ovvero occorre decidere in quali elementi del modello *target* saranno trasformati gli elementi del modello *source*. Nella seguente tabella saranno inseriti il mapping effettuato nella trasformazione dei modelli da BPMN a UML.

BPMN	**UML**
Start Event	Initial Node
Participant	Activity Partition
End Event	Activity Final Node
Task	Accept Event Action
Message	Opaque Action
Exclusive Gateway	Decision Node
Inclusive Gateway	Merge Node
Intermediate Catch Event	Time Event
Message Flow	Control Flow
Sequence Flow	Control Flow*

Ci sono dei casi in cui due elementi BPMN vengono trasformati in un unico elemento UML. Nel particolare è possibile notare come sia i *Message Flow* che i *Sequence Flow* verranno trasformati in *Control Flow* di UML. Ci saranno delle differenze, infatti nel secondo caso, cioè quello in cui si parte da un elemento *Sequence Flow*, si è in possesso di più informazioni riguardanti la sorgente e il target dello specifico, di conseguenza si vogliono conservare queste informazioni anche nel modello UML.

2.4 Profilo SoaML

La specifica SoaML (Service oriented architecture Modeling Language) [3] fornisce un metamodello e un profilo UML per la specifica e la progettazione di servizi all'interno

di un'architettura orientata ai servizi (SOA). SoaML estende UML2 per supportare i concetti di servizio. SoaML fornisce uno standard per progettare e modellare soluzioni SOA utilizzando il linguaggio UML.

2.5 Annotazioni SoaML

Qui di seguito una tabella riassuntiva del *mapping* da un modello BPMN ad un modello UML, con le rispettive specifiche SoaML, ma con un insieme ristretto di elementi specifico al caso in esame.

BPMN	**UML**	**Annotazione SoaML**
Start Event	Initial Node	
Participant	Activity Partition	Consumer/Provider
End Event	Activity Final Node	
Task	Opaque Action	
Message	Accept Event Action	Service Contract
Exclusive Gateway	Decision Node	
Inclusive Gateway	Merge Node	
Intermediate Catch Event	Time Event	
Message Flow	Control Flow	
Sequence Flow	Control Flow	

Capitolo 3

Trasformazione modello "Esercitazione in classe"

In questa parte sarà affrontata l'applicazione di una trasformazione di un modello BPMN in un modello UML annotato con profilo SoaML tramite l'utilizzo del linguaggio ATL. Nello specifico sarà utilizzato il modello "Esercitazione in classe" come modello sorgente.

3.1 Parte 1: trasformazione dal modello BPMN al modello UML

Di seguito il modello BPMN sorgente utilizzato nel caso in esame che simula una semplice esercitazione in classe:

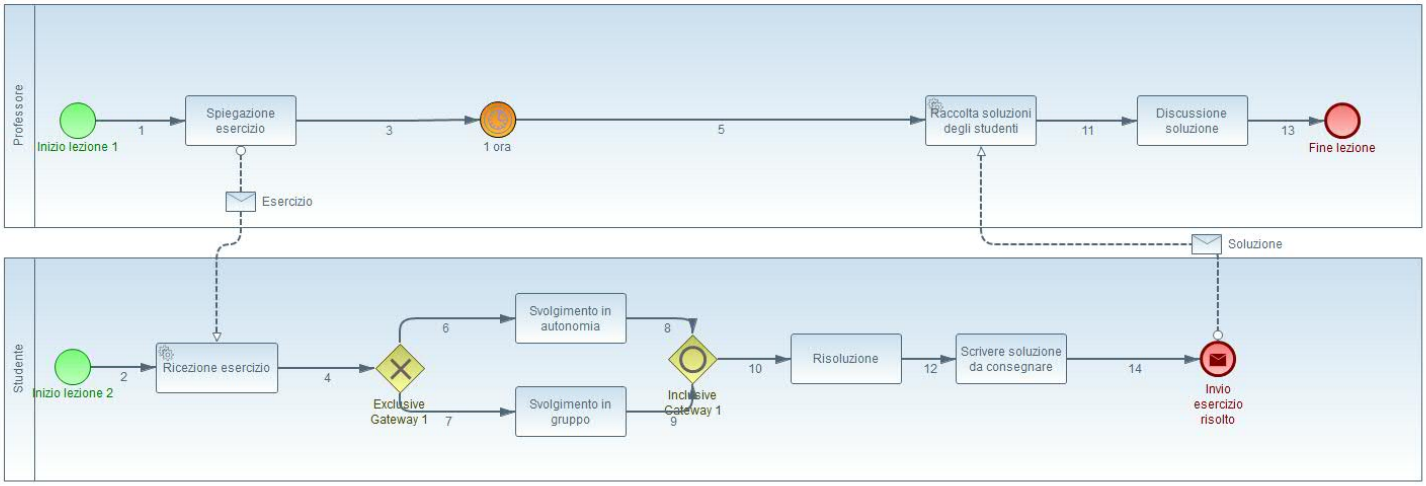

Figura 3.1: Modello BPMN "Esercitazione in classe"

Partendo da questo modello, utilizzando l'IDE Eclipse per lo sviluppo, dobbiamo configurare correttamente il file *.atl*, specificando il *path* del modello sorgente, quello di destinazione e quelli dei relativi metamodelli:

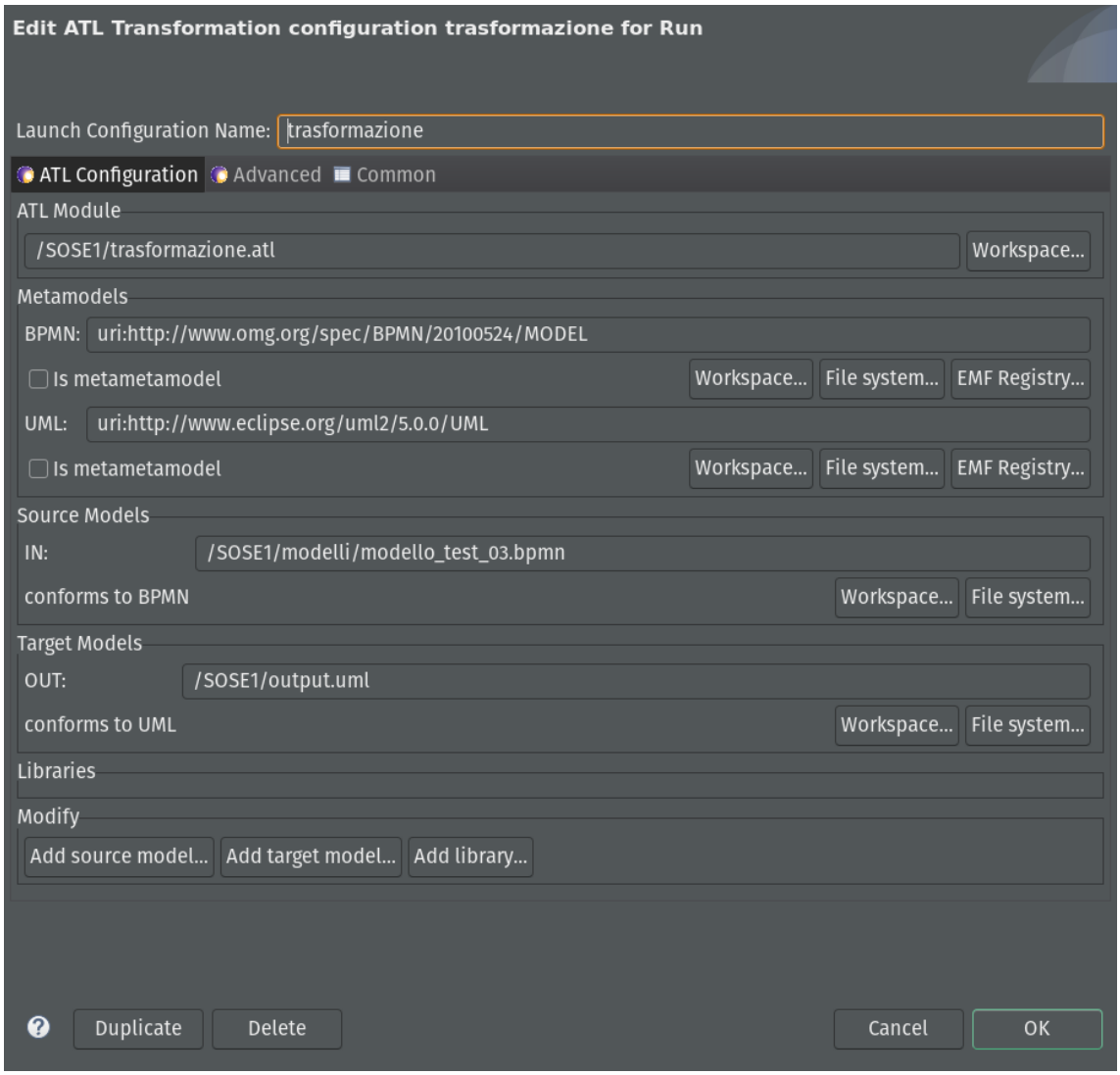

Figura 3.2: Configurazione di una ATL Transformation da BPMN a UML

I metamodelli utilizzati sono quelli nel registro EMF di Eclipse con i seguenti URI:

`http://www.omg.org/spec/BPMN/20100524/MODEL` e

`http://www.eclipse.org/uml2/5.0.0/UML`

Per effettuare una trasformazione è necessario utilizzare la corretta Virtual Machine di ATL. In questo caso, si utilizza ***EMF-specific VM***, selezionandola dalla finestra

"Advanced" del menù sopra riportato:

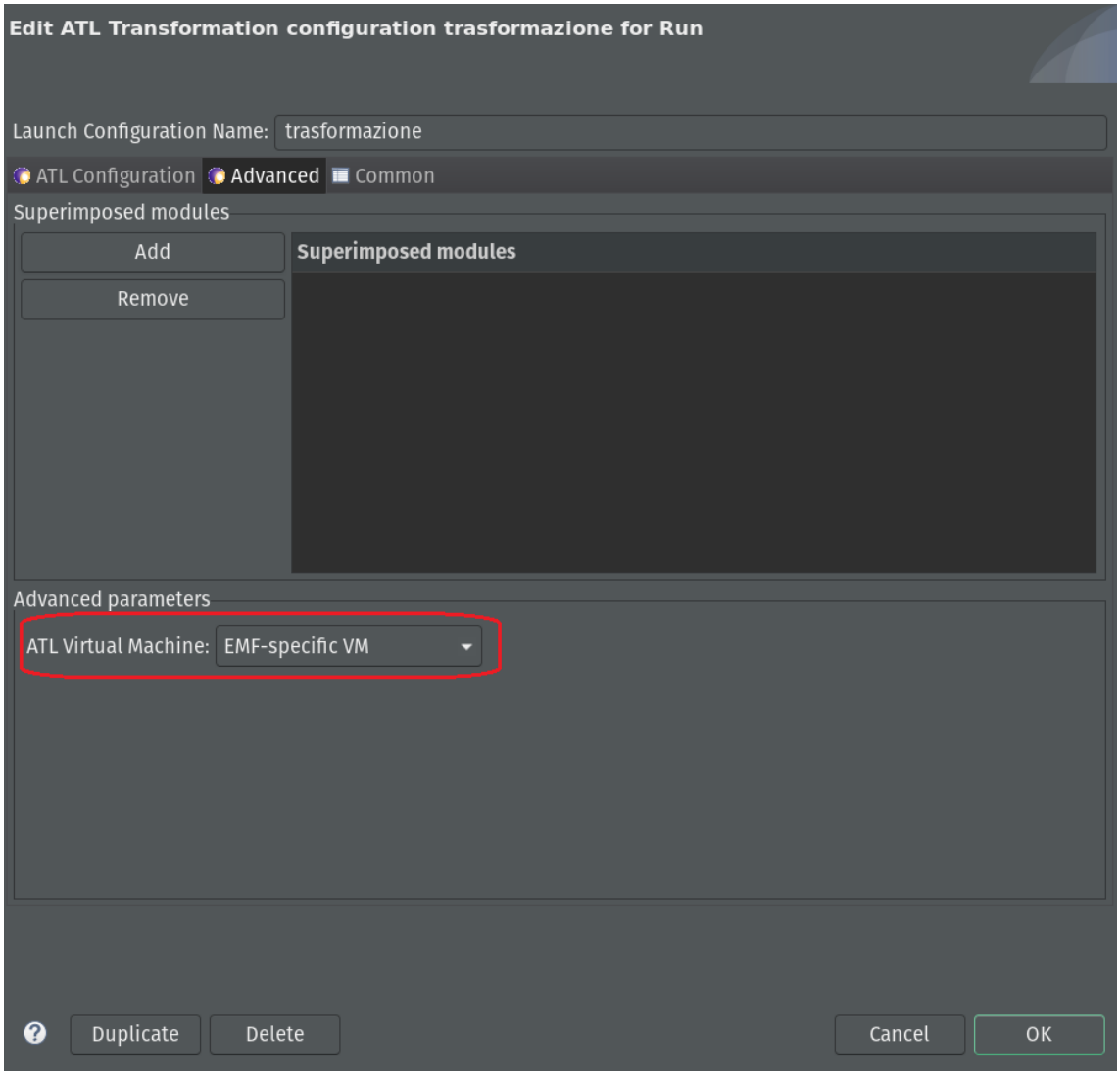

Figura 3.3: Sezione Advanced della ATL Transformation

Il codice ATL è strutturato con alcune righe che determinano quali metamodelli saranno usati in **INPUT** e **OUTPUT**, le loro sorgenti dati e il nome del modulo ATL.

```
1 -- @nsURI BPMN=http://www.omg.org/spec/BPMN/20100524/MODEL
2 -- @nsURI UML=http://www.eclipse.org/uml2/5.0.0/UML
3
4 module transformationBPMN2UML;
5 create OUT : UML from IN : BPMN;
```

Questa intestazione permette di creare in automatico un file *.asm*, necessario per l'esecuzione di una ATL Transformation. A questo punto, possiamo iniziare a scrivere le *rules* per strutturare il file di output e trasformare gli elementi da BPMN a UML. Di seguito un esempio di codice:

```
unique lazy rule BPMNPkg {
  from
    s : BPMN!Definitions
  to
    t : UML!Model
}

unique lazy rule ActivityPkg {
  from
    s : BPMN!Definitions
  to
    t : UML!Activity (
      package <- thisModule -> BPMNPkg()
    )
}

rule StartEvent2InitialNode{
  from
    s : BPMN!StartEvent
  to
    t : UML!InitialNode(
      name <- s.name,
      activity <- thisModule.ActivityPkg())
}

rule SequenceFlow2ControlFlow{
  from
    s : BPMN!SequenceFlow(not (s.sourceRef.oclIsKindOf(BPMN!
    IntermediateCatchEvent) or s.targetRef.oclIsKindOf(BPMN!
    IntermediateCatchEvent)))
  to
    t : UML!ControlFlow (
      name <- s.name,
      source <- s.sourceRef,
      target <- s.targetRef,
      activity <- thisModule.ActivityPkg())
}
```

Eseguendo il codice otterremmo il seguente risultato:

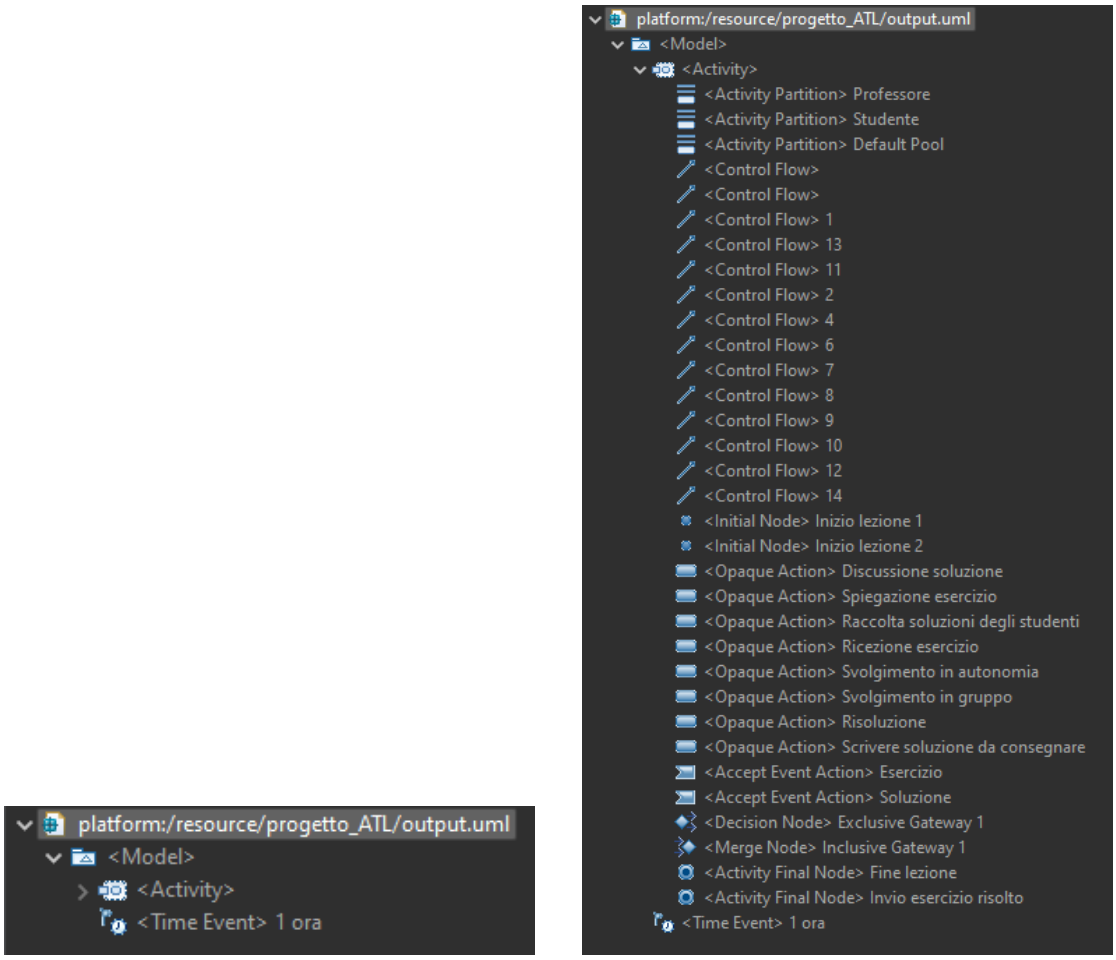

Figura 3.4: File output della trasformazione

3.2 Parte 2: applicazione del profilo SoaML

Per applicare i profili in ATL è necessario usare una Virtual Machine specifica, in particolare la ***EMFTVM*** (Eclipse Modeling Framework - Transformation Virtual Machine). Per utilizzarla è necessario creare un nuovo file *.atl* e inserire come prima riga

```
1 -- @atlcompiler emftvm
```

Una volta salvate le modifiche viene generato automaticamente un file *.emftvm* eseguibile dalla VM. Il file viene generato in automatico solo se si è in possesso del

package reperibile dall'URI https://download.eclipse.org/releases/latest e inserendolo nel campo del menù "Help → Install New Software" di Eclipse. A questo punto, possiamo configurare il nostro file *.atl* cliccando con il tasto destro del mouse sul file e selezionando "Run As → Run Configurations", compilando nel seguente modo i campi:

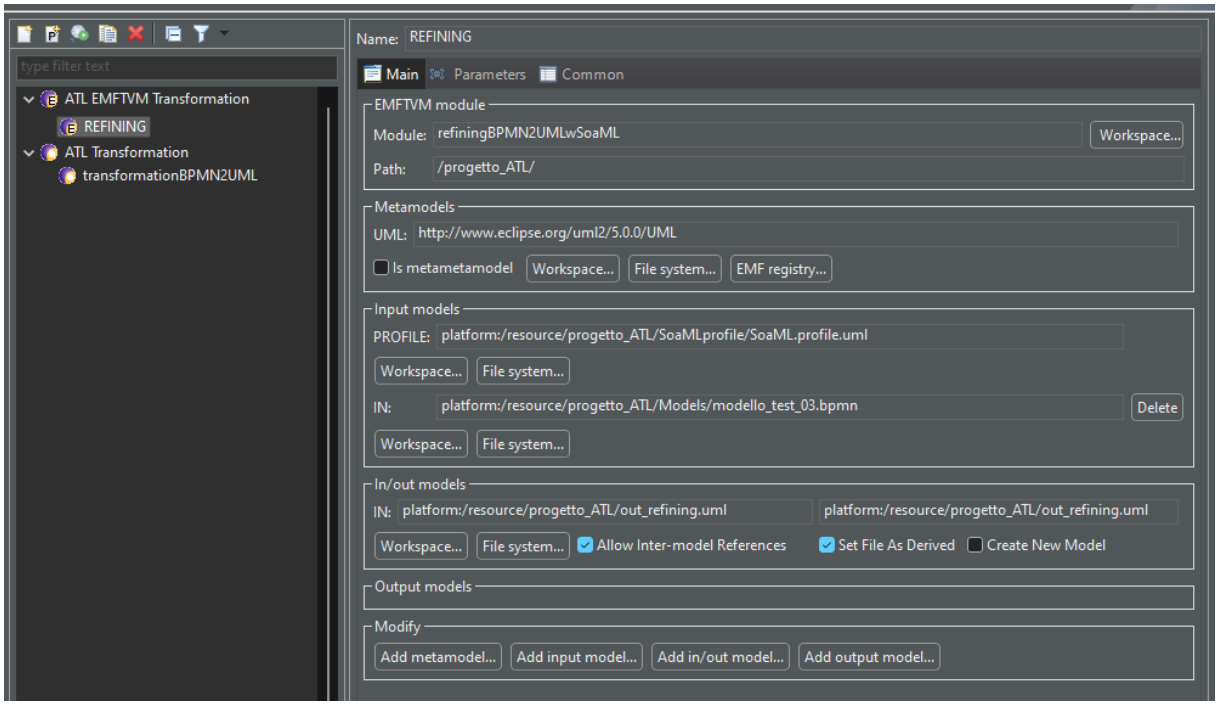

Figura 3.5: Configurazione di una ATL EMFTVM Transformation

Ora dobbiamo utilizzare la refining mode di ATL. Ossia dobbiamo specificare nel codice che abbiamo bisogno di una trasformazione **in-place** e non out-place. Infatti il file output della configurazione è lo stesso che viene inserito in input.

```
1 -- @atlcompiler emftvm
2 -- @nsURI UML=http://www.eclipse.org/uml2/5.0.0/UML
3 -- @path PROFILE=/progetto_MBSE/SoaMLprofile/SoaML.profile.uml
4
5 module refiningBPMN2UMLwSoaML;
6 create OUT : UML refining IN : UML, PROFILE : UML;
```

A questo punto, applichiamo il profilo al package più esterno del nostro file *.uml* in input:

```
helper def : SoaMLProfile : UML!Profile =
  UML!Profile.allInstancesFrom('PROFILE')->any(p | p.name = 'SoaML').
   debug();

rule UMLPkg{
    from
        s: UML!Model
    to
        t: UML!Model
  do {
    t.applyProfile(thisModule.SoaMLProfile);
  }
}
```

Ottenendo:

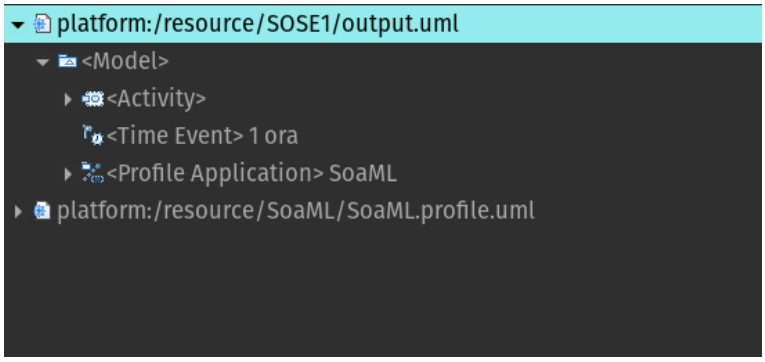

Figura 3.6: File output della trasformazione con profilo applicato

> **Nota**
>
> Rispetto a EMF-specific VM, EMFTVM offre funzionalità aggiuntive per eseguire trasformazioni più sofisticate e dettagliate tra i modelli EMF.

3.3 Parte 3: applicazione degli stereotipi

Per applicare gli stereotipi ai singoli elementi del file *.uml* con il profilo SoaML, ci sono due metodi (ufficiali): da codice o da interfaccia grafica. Da codice:

```
rule AcceptEventActionStereotype{
  from
```

```
    s : UML!AcceptEventAction
  to
    t : UML!AcceptEventAction (name <- s.name)
  do {
    t.applyStereotype(thisModule.getStereotype('SoaML::Contracts::
     Service Contract'));
  }
}
```

Da interfaccia grafica (Eclipse 4.4) [5], invece:

- Selezionare un elemento (e.g., <AcceptEventAction>) nell'editor UML.
- Selezionare la voce dal menù UML Editor > Element > Apply Stereotype...
- Selezionare uno stereotipo (e.g., 'SoaML::Contracts::Service Contract'), premere su Add e poi OK.

Bibliografia

[1] M. Brambilla, J. Cabot, M. Wimmer, *"Model-Driven Software Engineering in Practice"*, Morgan & Claypool, Capitolo 8.

[2] ATL documentazione, `https://eclipse.dev/atl/documentation/`.

[3] SoaML documentazione, `https://www.omg.org/spec/SoaML`.

[4] BPMN documentazione, `http://www.bpmn.org`.

[5] Eclipse applicazione stereotipi, `https://wiki.eclipse.org/MDT/UML2/Introduction_to_UML2_Profiles`