Model-Driven Software Engineering

Metamodels and Domain-Specific Languages I

Dr. Jochen Küster (jku@zurich.ibm.com)
Motivation: Common Aspects of MDSE Approaches

- In MDSE approaches, the use of models and model transformations is proposed.
- Models are expressed in UML, an extension of UML, or a domain-specific language.
- The syntax and semantics of models used in a MDSE approach has to be clearly defined.

Modeling language
Extension of UML
Domain-Specific Language
Syntax and semantics of modeling languages
Contents

- Introduction to Syntax and Semantics of Modeling Languages
- Introduction to the Object Constraint Language
- Defining Syntax and Semantics of BPMN
- Summary and Literature
Introduction to Syntax and Semantics of Modeling Languages
Models and Modeling

- A model is an abstraction of phenomena of the real world
  - Focus on relevant aspects, abstract from irrelevant aspects
- Many different kinds of models in engineering, mathematics and computer science
- Examples include
  - Mathematical models used by mathematicians
  - Architectural models used by architects
  - Models of a software system used in software engineering
- Modeling is the activity of creating models
A Library Model

- A Library Model abstracts from the real world library
- In this case: UniZuerich Library
- How do we know whether this is a correct model?
Syntax and Semantics of (Modeling) Languages

- Syntax: Form of the “words” in a language
- Semantics: Meaning of the “words” in a language, e.g. how to execute program statement

- Programming languages:
  - Syntax: Context-free grammar
  - Semantics: Denotational semantics, attribute grammars,..

- Visual Modeling Language:
  - Syntax: Metamodel + OCL constraints
  - Semantics: often informally described in the language specification
Metamodels and Metamodelling

- A metamodel is a model that describes a set of models
- Metamodels are used for defining the abstract syntax of modeling languages
- Metamodels are visual and better suited to define the syntax of modeling languages
- Metamodelling is the activity of creating meta-models
- Instantiation concept from object-orientation

The Library Metamodel describes the set of all Library models
Library Example Revisited

Metamodel

Model

<<instance of>>

UniZuerichLibrary : Library

<<instance of>>

MDSDBook : Book
  pages = 355
  title = "Model-Driven Software Development"

<<instance of>>

SoftwareFactoriesBook : Book
  pages = 6000
  title = "Software Factories"
What is the Meta Object Facility?

- Meta Object Facility (MOF) is an OMG standard for enabling development and interoperability of model and metadata driven systems
- Current version MOF 2.0
- Systems that use MOF include
  - modeling and development tools
  - metadata repositories and data warehouse systems
- MOF is the foundation or used by other standards
  - UML standard
  - BPMN standard (Business Process Model and Notation)
The Meta Object Facility (MOF) of the OMG - Concepts

M3 Level

Meta Metamodel

instance of

M2 Level

Metamodel

instance of

M1 Level

Model

is described by

M0 Level

Real World
Defining UML with Metamodelling – Statemachine

M2 Level

M1 Level
Where is Meta-Modeling used?

- For defining modeling languages
  - Meta-model has to be defined for such a language
  - See State-machine example

- For defining domain specific languages
  - Meta-model based approach is one form
  - See later in this lecture

- For defining model-to-model transformations
  - Transformations are typically defined by a meta-model to meta-model mapping
  - See Lecture on Transformations

- For defining model-to-code transformations
  - Transformations are typically defined by a meta-model to code mapping

- For realizing tool integration
  - based on a common meta-model
Comparison of Metamodels and Grammars

- A grammar is operational
  - It describes how to derive words of a language
  - Formalisms such as context-free grammars, automata etc.

- Meta-models are declarative
  - It does not describe how to derive words of a modeling language
  - If a word (model) is given, one can easily check whether it conforms to the meta-model
Introduction to the Object Constraint Language
But: Metamodelling is not sufficient

- Only books with more than 0 pages are allowed
- Metamodelling on its own cannot restrict the allowed instances based on such constraints
Object Constraint Language

- The Object Constraint Language (OCL) is a formal declarative constraint language standardized by the OMG
- Current version OCL 2.2
- Constraints are used for restricting the set of allowed model instances
- OCL is widely used in the UML standard
- OCL can also be used in modeling tools for defining constraints on models
Usage of OCL in MDSE

- **Constraints in UML models**
  - Invariants for classes, interfaces, stereotypes, …
  - Pre- and postconditions for operations
  - Guards for messages and state transition
  - Specification of messages and signals
  - Calculation of derived attributes and association ends

- **Constraints in metamodels**
  - Invariants for meta model classes
  - Rules for the definition of well-formedness of meta model

- **Query language for models**
  - In analogy to SQL for DBMS, XPath and XQuery for XML
  - Used in transformation languages
Examples of using OCL - Invariants

“No writer has written more than 5 books.”

Context  Writer

Inv: self.book -> size() \leq 5
Examples of using OCL - Invariants

“No writer has written more than 3 SchoolBooks."

**Context**  Writer

**Inv:** self.book -> select (v | v.oclIsKindOf(SchoolBook)) -> size() <= 3
Examples of using OCL - Invariants

“If pages are less than 300 it is a fiction book.”

Context Book

Inv: page<300 implies self-> forAll (v | v.oclIsKindOf(FictionBook))

“There exists at least one SchoolBook”

Context Book

Inv: Book.allInstances()-> exists(c | c.oclIsKindOf(SchoolBook))
Context of OCL Expression and Invariants

- **Context** specifies the UML context of the OCL expression
- **Self** refers to the contextual instance
- Invariants must be true for all instances of the type at any time
- Self may be dropped
OCL Values and Types

- OCL is a typed language
  - Expressions are evaluated to a value of a certain type
  - For background see types and typing in programming languages

- OCL contains basic types such as Boolean, Integer, Real, String together with usual operations
  - Boolean : True, False

- OCL contains set-valued types such as collections, sets, bags, and sequences
  - Set: \{1,2,3\}

- OCL contains all user defined types that result from the class diagram
  - Book
Navigating a Class Diagram and Predefined Properties

- **Object.associationEndName**
  - Returns the set of objects
  - Returns an object (cardinality 0 or 0..1 or 1)

- **Predefined properties**
  - **Size()**, **isEmpty()**, **notEmpty()**
  - **self.oclIsTypeOf(t)** evaluates to true if self and t have the same type
  - **self.oclIsKindOf(t)** evaluates to true if self has the same type or is a subtype

**Context Book**

Inv: self.pages > 0

**Context Writer**

Inv: self.book -> size() > 0
Predefined Operations on Collections (1)

**Context**  
**Writer**

**Inv:** `self.book -> select (v | v.oclIsKindOf(SchoolBook)) -> size() <= 3`

- Returns set
- Returns collection as described below
- Evaluates the size

- `collection -> select(boolean-expression)`  
  - Results in collection of those elements for which boolean-expression evaluates to true

- `collection -> select(v | boolean-expression-with-v)`  
  - Results in collection of those v for which boolean-expression-with-v evaluates to true, iteration over v

- `collection -> select(v: Type | boolean-expression-with-v)`  
  - Results in collection of those v for which boolean-expression-with-v evaluates to true, iteration over v which is of type Type

- Same for `collection->reject(...)`
Predefined Operations on Collections (2)

- `collection -> collect(boolean-expression with variations)`
  - results in a new collection formed by the results of boolean-expression
  - can be used to collects properties of an object

- `collection -> forAll(boolean-expression with variations)`
  - results in a boolean, true if boolean-expression is true for all elements

- `collection->exists(boolean-expression with variations)`
  - results in boolean, true if boolean-expression is true for at least one element in the collection

- `collection->iterate(..)`
  - generic iteration over the collection elements
Further Examples

Context Writer
Inv: self.book -> collect(pages) -> size()

Context Writer
Inv: self.book -> forAll(pages>=0)

Context Writer
Inv: self.book -> exists( v | v.oclIsKindOf(SchoolBook))

Context Writer
Inv: self.book -> notEmpty()
An initial vertex can have at most one outgoing transition.
   Context Pseudostate: (self.kind = #initial) **implies** (self.outgoing->size() <= 1)

Transitions outgoing pseudostates may not have a trigger (except for those coming out of the initial pseudostate).
   Context Transition Inv:
   (source.oclIsKindOf(Pseudostate) **and** (source.kind <> #initial)) **implies** event->isEmpty()
Defining Syntax and Semantics of BPMN
Abbreviated Metamodel for Process Models (BPMN 2.0)

- Metamodell defines the abstract syntax of process models
Instantiation of the Meta Model

- Process
  - processType : ProcessType
  - aClosed : boolean

- FlowElementsContainer
  - from Cameron

- FlowElement
  - name : String

- FlowNode
  - from Cameron

- SequenceFlow
  - sourceRef : String
  - outgoing
  - targetRef : String
  - incoming

- Activity
  - from Activities
  - isForCompensation : Boolean
  - startQuantity : Integer
  - completionQuantity : Integer

- Event
  - from Events

- Gateway
  - from Gateways
  - gatewayDirection : GatewayDirection

Flow:
- Register Claim
- Grant Claim
- Reject Claim
- Close Claim
- Exception Handling
Semantics of BPMN 2.0

Informally described in the standard:

- A **Process** is instantiated when one of its **Start Events** occurs. Each occurrence of a **Start Event** creates a new **Process Instance** unless the **Start Event** participates in a **Conversation** that includes other **Start Events**. In that case, a new **Process instance** is only created if none already exists for the specific **Conversation** (identified through its associated correlation information) of the **Event** occurrence. Subsequent **Start Events** that share the same correlation information as a **Start Event** that created a **Process instance** are routed to that **Process instance**. Note that a **global Process** must neither have any empty **Start Event** nor any **Gateway** or **Activity** without **incoming Sequence Flow**. An exception is the **Event Gateway**.

- A **Process** can also be started via an **Event-Based Gateway** that has no **incoming Sequence Flow** and its **Instantiate** flag is **true**. If the **Event-Based Gateway** is **exclusive**, the first matching **Event** will create a new **instance** of the **Process**. The **Process** then does not wait for the other **Events** originating from the same **Event-Based Gateway** (see also semantics of the **Event-Based Exclusive Gateway** on page 437). If the **Event-Based Gateway** is **parallel**, also the first matching **Event** creates a new **Process instance**. However, the **Process** then waits for the other **Events** to arrive. As stated above, those **Events** must have the same correlation information as the **Event** that arrived first. A **Process instance** completes only if all **Events** that succeed a **Parallel Event-Based Gateway** have occurred.
Semantics of BPMN 2.0 (continued)

- An **Activity** is *Ready* for execution if the required number of *Tokens* is available to activate the **Activity**. The required number of *Tokens* (one or more) is indicated by the attribute StartQuantity. If the **Activity** has more than one **Incoming Sequence Flow**, there is an implied **Exclusive Gateway** that defines the behavior.

- When some data InputSet becomes available, the **Activity** changes from *Ready* to the *Active* state. The availability of a data InputSet is evaluated as follows. The data InputSets are evaluated in order. For each InputSet, the data inputs are filled with data coming from the elements of the context such as **Data Objects** or Properties by triggering the input **Data Associations**. An InputSet is *available* if each of its *required* data inputs is available. A data input is *required* by a data InputSet if it is not optional in that InputSet. If an InputSet is available, it is used to start the **Activity**. Further InputSets are not evaluated. If an InputSet is not available, the next InputSet is evaluated. The **Activity** waits until one InputSet becomes available.

- Complicated and subject to interpretation of the reader
- Now consolidated and defined in BPMN spec
Semantics and Scope of Events

Throwing of an event creates a trigger that carries information.

- Publication: Trigger can be received by any catching events in the scope where the trigger is published.
- Direct resolution: Timer and conditional events are directly resolved where they are activated.
- Propagation: Trigger is forwarded to the innermost scope which has an event capable to catch the trigger.
- Cancellation: Directed towards a process or activity instance. Terminates all running instances and compensates completed activities.
- Compensation: Compensation of a subprocess calls all compensations of enclosed activities.
Execution Semantics of Compensation Events

- Compensation is triggered using a throw compensation event
- The activity which needs to be compensated is referenced (optional if clear from context)
- If no activity is referenced the compensation event is broadcast inside the subprocess or process
- Compensation is triggered synchronously, throw compensation event waits for completion (exceptions are possible)
- Default compensation ensures that compensation activities are performed in reverse order of the execution of original activities
- Compare BPMN 2.0 pages 280-282, 405-406, 241
Semantics of Inclusive Gateway

The **Inclusive Gateway** is activated if

- At least one incoming sequence flow has at least one *Token* and

- for each empty incoming sequence flow, there is no *Token* in the graph anywhere upstream of this sequence flow, i.e., there is no directed path (formed by Sequence Flow) from a *Token* to this sequence flow unless
  - the path visits the inclusive gateway or
  - the path visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway.

- Upon execution, a *Token* is consumed from each incoming sequence flow that has a *Token*. A *Token* will be produced on some of the outgoing sequence flows. In order to determine the outgoing sequence flows that receive a *Token*, all conditions are evaluated. The evaluation does not have to respect a certain order.

- For every condition, which evaluates to true, a *Token* must be passed on the respective sequence flow. If and only if none of the conditions evaluates to true, the *Token* is passed on the default sequence flow. In case all conditions evaluate to false and a default flow has not been specified, the inclusive gateway throws an exception.
BPMN semantics: Inclusive Gateway

- In this case, second inclusive gateway waits
- Inclusive gateways still problematic, can lead to deadlocks
Syntactic and Semantic Correctness

- **Syntactic correctness of the process model**
  - Is the process model syntactically correct?
  - Is the process model an instance of the meta-model of the language
  - Usually checked by the modeling tool

- **Semantic correctness of the process model**
  - Is the process model correct with regards to the semantics of the modeling language?
  - Is the process model a correct model with regards to reality?
  - Does it contain deadlocks, does it terminate?
  - Can be checked by following *best-practices of process modeling*
  - Can be validated by using *simulation*
Summary and References
Summary of Lecture and References

- Metamodels allow definition of abstract syntax of modeling languages
- Semantics of visual modeling languages is often specified using natural language explanations
- OCL is used for restricting sets of instances and allows to specify invariants, pre- and postconditions

References:

- B. Pierce. Types and Programming Languages. MIT Press, 2002