Model Driven Systems Engineering Notes

Design, Develop and Manage complex systems

Author : Davide Galassi <davxy@datawok.net>

Based on the work of:
Mohamad Gharib <mohamad.gharib@unifi.it>

Version 0.2

Table of Contents

Requirements Engineering

What is requirement engineering

“A problem can be defined as the difference between things as they are now and as they are desired.”

Requirements engineering refers to the process of defining, documenting and maintaining requirements.

Requirements constitute a first specification for the new system.

Problem space → Requirements → Design → Implementation → *Solution space*

The new system requirements must identify:

- **Why** the system is needed.
- **What** system features are needed.
- **How** the system is to be constructed.

Two types of requirements

- **Functional** : what the system is supposed to do (inputs and outputs)
- **Non-functional** : how the system will do it (safety, performance, reliability, security, etc.)

Why requirement engineering

Bad requirements is a main reason for project failures.
Errors in requirements propagate to other activities.
Correct errors in later project stages is more expensive.

How to do requirements engineering

Phases:

1. Elicitation
2. Analysis and negotiation
3. Documentation
4. Validation

Requirements Elicitation

System **stakeholders**: end-users, designers, analysts, management.

Techniques: interviews, questionnaires, task analysis, market investigations, prototyping, demo, etc.

The application domain should be explored together with its environment (political, organizational, social aspects). The environment constraints may influence the system.

Analysis and negotiation

The stakeholders requisites are analyzed by domain experts to identify conflicts and issues.

Negotiation: conflicts are solved by relaxing/modifying requirements.
Documentation of the negotiation procedure is produced.

Documentation

Requirements documents are written in a structured, clear and unambiguous way.
Serve as **contract** between stakeholders and the subjects in charge of design and implementation.
Requirements must be **refined** to a level of detail sufficient for system design.

Requirements **specification** is the operationalized form of requirements. Provides enough information for the implementer to build the system without further knowledge. A specification contains use cases. In contrast, requirements are not so specific from the technical point of view.

Validation

Check for:

- **Validity** : verify the requirements with all stakeholders of the system-to-be.
- **Completeness** : requirements capture all the functions, features and constrains expected.
- **Consistency** : the documentation is coherent.
- **Realism** : requirements can actually be implemented.
- **Verifiability** : requirements are clear enough to be verified by different stakeholders.

When for a given domain there is a **standardization body**, then is up to this association to do a one-time requirements engineering. The product is a standard normative (e.g. ISO, IEC, IEEE) to be shared among the several implementers.

Public enquiry. Often, for open standards, for the validation phase a requirement draft is published for domain experts (stakeholders) review and approval.

Modeling

Modeling in its broadest sense is the cost−effective use of something in place of something else for some purpose. “Jeff Rothenberg. The Nature of Modeling”

In real life we use modeling every day to **infer** solutions to problems and simplify complex situations using **experience**.

Modeling. Represents a particular reference **domain** for a particular **purpose** in a **cost-effective** way.

Model. Abstraction of relevant system components, properties, functionalities and environment interactions.

A model allows problems solving, analysis, simulations and tests avoiding the complexity, cost, danger and irreversibility of reality.

Granularity. Is the level of abstraction of the model. A model should be:

- **Precise**: all relevant properties are described
- **Concise**: irrelevant properties are discarded.

All and only relevant properties should be described.

Model taxonomy

Timing:

- **discrete** : system moves from one state to the other in discrete steps (e.g. FSM).
- **continuous** : system described by a set of variables varying in real time. Often described by differential equations.

Formality: informal, semi-formal, formal.
Captured knowledge: domain, task specific.
Completeness: incomplete, complete.
Abstraction levels:

- **conceptual**: high level non technical terms/concepts.
- **logical**: specific but abstract terms/concepts (entities, attributes, relationships).
- **physical**: specific and implementation oriented terms/concepts (table, column, key).

Model driven engineering (MDE)

Methodology often used for software development that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem.

Highlights and aims at abstract representations of knowledge and activities that govern a particular domain, rather than implementation and operational (e.g. algorithmic) aspects.

Model Driven Architecture (**MDA**) : guidelines for structuring of specifications, which are expressed as models. Example:

- Requirements models: OO diagrams, UML, SysML, ...
- Software or business process: statecharts, petri-nets, ...

Advantages

Modeling allows people (usually stakeholders) to **communicate** and exchange knowledge.
Allows to separate the macro problem in **sub-components** that can be addressed separately by different teams and understood independently (e.g. dependability, security, HMI).
Allows revealing/detecting **errors** in the system design at early analysis.
A model is **easy** to test and modified in a controlled environment.

Disadvantages

Can be a very **time consuming** activity.
Models are abstractions that are not necessary **correct** for the implementation.

Model V&V

Verification and Validation are independent procedures used together to check that a product, service or system meets the specifications and requirements.

- **Verification** : checks that the model has been correctly implemented with respect to the conceptual model, matches the specifications and assumptions. It is like debugging and often performed internally.
- **Validation** : checks the accuracy of the model's representation of the real system. And thus that the model respects the requirements of the stakeholders. Often is performed externally.

One technique for automatic model V&V is the usage of constraint languages (e.g. OCL).
A correct model doesn't imply a correct implementation, however if a test fails on the model, then is unlikely that it will succeed on the real system.
In some domains, the V&V process is performed by specialized independent bodies.

System Development Life Cycle (SDLC)

1. **Initiation**. Begins with an idea. A concept proposal is created.
2. **System Concept Development**. Defines the scope and boundary of the system as well as performing the feasibility study.
3. **Planning**. Developing a management plan, allocating resources, etc.
4. **Requirements Analysis**. Analyzing the “users” needs and creates a “detailed” functional and non-functional requirements document.
5. **Design**. Transforms “detailed” requirements info into a “complete” detailed system design document.
6. **Development**. Converts a design into a “complete” detailed system.
7. **Integration and Test**. Demonstrate that the developed system conforms to requirements (validation).
8. **Implementation**. Implementing the system in its operational context.
9. **Operations and Maintenance**. Tasks to operate and maintain information system.
10. **Disposition**. Describes end-of-system activities.

MDA Life-Cycle (MDALC)

1. **Problem Definition** (SDLC 1,2,3)
2. **Requirements Analysis** (SDLC 4)
3. **Design** (SDLC 5)
   1. Conceptual Modeling
   2. Creating the DSML
   3. Constraining the DSML (e.g. OCL)
4. **Implementation** (SDLC 6)
5. **Experimentation** (SDLC 7)

The last three SDLC phases doesn’t map to any MDALC phase.

Experimentation comprises **validation**, that checks whether the product satisfies the intended purpose of use (stated problem). Each step can refines the previous (*agile* methodology).

Conceptual Modeling

Concept: abstract and general idea. Understanding is retained in the mind, from experience and/or imagination.

Conceptual model represent aspects and concepts of a specific domain in high-level terms.
Aims to **express** the meaning of **terms** and **concepts** used by the **domain experts** to discuss a problem and finds a correct relationships between different entities. Avoids ambiguities and misinterpretations.
Is independent of implementation and design concerns.
A conceptual model consists of entities and relationships between entitites.

Conceptual Modeling Languages

Languages used to *create*, *manage* and *validate* a model.

Brief history of languages used (mainly) for conceptual modeling:

ER (1970) → UML (1990) → SysML (2001)

A modeling language is used to express information/knowledge about a system/domain in a **structured** and **consistent** way relying on a set of syntactic and semantic rules. A conceptual modeling language includes:

- **Building blocks** (constructs) : primitive terms and abstraction mechanism (e.g. *UML classes*)
- **Semantics**: constraints on the use of the building blocks (e.g. *OCL predicates*)

Metadata : data about data.
Metamodel. The model used to describe another model. Defines the **key concepts** of a modeling language as well as various **relationships** among these concepts.

Example. Given the Simpsons family model we extract information to construct a Family meta model to represent all other families. Simpson family is an instance of the meta model Family.
If we work in UML then Family is an instance of a UML meta-model.

Meta Object Facility

The Meta Object Facility (**MOF**) is an Object Management Group (OMG) standard for modeling languages.

- Model for meta-models
- All UML modeling concepts can be represented with the MOF
- UML modeling concepts - as classes, associations, actors, relations, etc. - are defined as metaclasses (instances of MOF classes).

Four layers:

- M0 : an actual system (e.g. Simpson Family)
- M1 : is a system model. Entities and relationships that makes up a system (e.g. Family Model)
- M2 : defines a model for models in M1 (e.g. UML meta-model)
- M3 : defines a model for models in M2 (the meta-meta-model) (MOF)

Every element in M{i} is an instance of M{i+1}
Reference: https://www.omg.org/mof/

UML and SysML

UML is a modeling language (not only) for Software Engineering based on object oriented paradigm.

Support for modeling **static** (**structure**) and **dynamic** (**behavior**) aspects of real world.

SysML is a modeling language (not only) for System Engineering based on UML (*profile* mechanism).

Involves modeling **blocks** instead of modeling classes, thus providing a vocabulary more suitable for Systems Engineering.

Can be easily understood by the software community, due to its relation with UML2, whilst it is also accessible to other communities.

Makes possible to generate specifications in a single language for heterogeneous teams, dealing with the realization of the system hardware and software blocks.

A block encompass software, hardware, data, processes and facilities.

SysML Diagrams **Taxonomy**

- **Structure** (static)
    - **Block Definition** : design phase, modified UML2 class diagram.
    - Package
    - Internal Definition
    - Parametric : SysML extension to analyze critical system parameters.
- **Behavior** (dynamic)
    - **Use Case** : requirements analysis, same as UML2.
    - **Sequence** : design phase, same as UML2.
    - Activity
    - State Machine
- **Requirements** : SysML extension.

Papyrus is a widespread plugin for Eclipse for UML and SysML modeling.

Block Definition Diagram

Provides an overview of the structural architecture of the system.

Can be used in terms of **entities** (**blocks**) composing the system as well as their **inter-relationships**.

Block characteristics

Attribute : is a named slot within a class that describes a range of values that instances of the class may hold. Are always single-values in UML, their value can be derived.

Operation : behavioral feature that may be owned by an interface, data type or class.

Block relationships

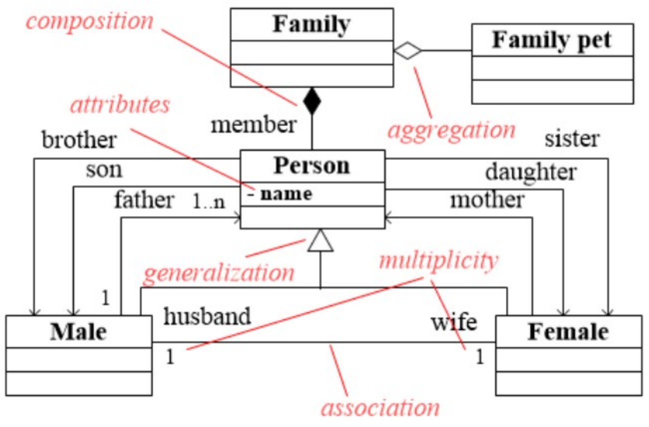
Generalization : is a mechanism for combining similar classes of objects into a single, more general class (person class is a generalization of both male and female classes).

Specialization : (*is a*) the reverse process of Generalization, that means creating new sub classes from an existing class (Male and Female classes are specializations of person).

Aggregation : (*has a*) an entity can exist independently of the other class. The component doesn't physically belongs to the composed entity (Family exists regardless of the "Family pet" existence).

Composition : (*has a*) an entity cannot exist without the component. The component physically belongs to the composed entity (Family cannot exist without Person).

Associations : a semantic relationships. Each association can have up to two **roles** for **participating** objects. Each role can also have an associated **multiplicity** (i.e. number of elements) that allows to specify the relation cardinality (e.g. *1* to *n*).



Flow ports are a SysML extension and represent what can go through a block (in/out) whereas it is data, matter or energy.

Blocks are shown as UML classes, stereotyped «block»

Use-Case Diagram

Offer a notation for building a preliminary "*abstract*" model of the system.

Used to represent a set of **actions** (**use cases**) that some system (**subject**) should or can perform in collaboration with one or more external users of the system (**actors**).

Use cases may represent user **goals** and user **interactions**.

Use cases allow to capture requirements of systems under design or consideration, describe functionality provided by those systems, and determine the requirements the systems pose on their environment.

Each use case should provide some **observable** and **valuable** result to the actors of the system.

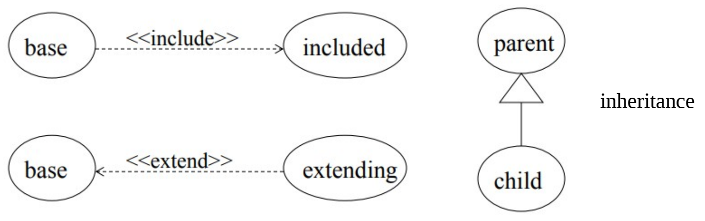
An actor specifies a **role played** by an external entity that interacts with the subject (e.g. by exchanging signals and data), a human user of the designed system, some other system or hardware using services of the subject. A single physical entity may play several different roles, and a specific role may be played by multiple entities. Actors must have names according to the assumed role.

An **association** between an actor and a use case indicates that the actor and the use case somehow interact with each other. Only binary associations are allowed between actors and use cases.

An actor could be associated to one or several use cases.

Relationships between use-case model entities:

- **Extend** dependency between use cases.
- **Include** dependency between use cases.
- **Inheritance** between use cases
- **Inheritance** between actors.



Include relation

The behavior of the included use case (the addition) is inserted into the behavior of the including (the base) use case. Could be used to:

- simplify large use cases by splitting into several use cases
- to extract common parts of the behavior of two or more use cases (sharing).

The purpose is modularization of behaviors making them more manageable.

Base use case is incomplete (abstract) and the included use case is required (not optional).

Extend relation

Extends the behavior of a base use case by adding some auxiliary actions. Used to model a part of a use case that the user may see as an **optional system behavior**; also models a separate sub-case which is **executed conditionally**.

The same extending use case can extend more than one use case and can be extended itself.

Base use case is complete (concrete) by itself, defined independently. Extending use case is optional, supplementary.

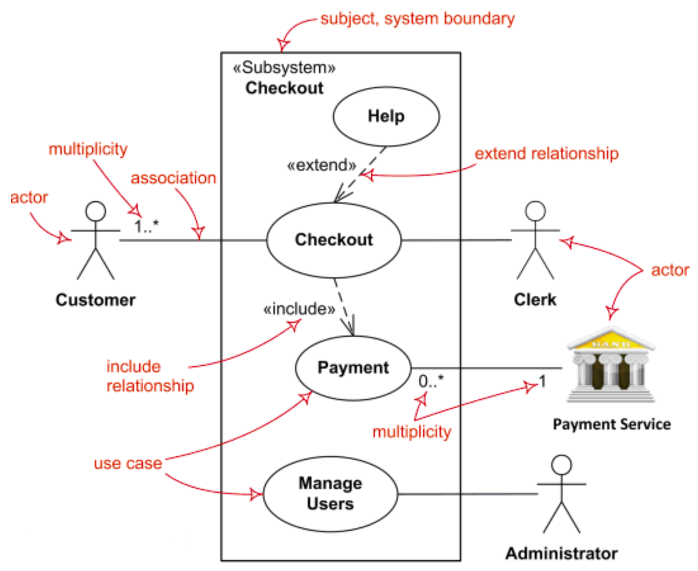
Generalization relation

The child use case inherits the behavior and meaning of the parent use case.

The child may add or override the behavior of its parent and can be used wherever the parent can be used (*is-a*).

Base use case could be abstract or concrete. Specialized use case is required if base use case is abstract.

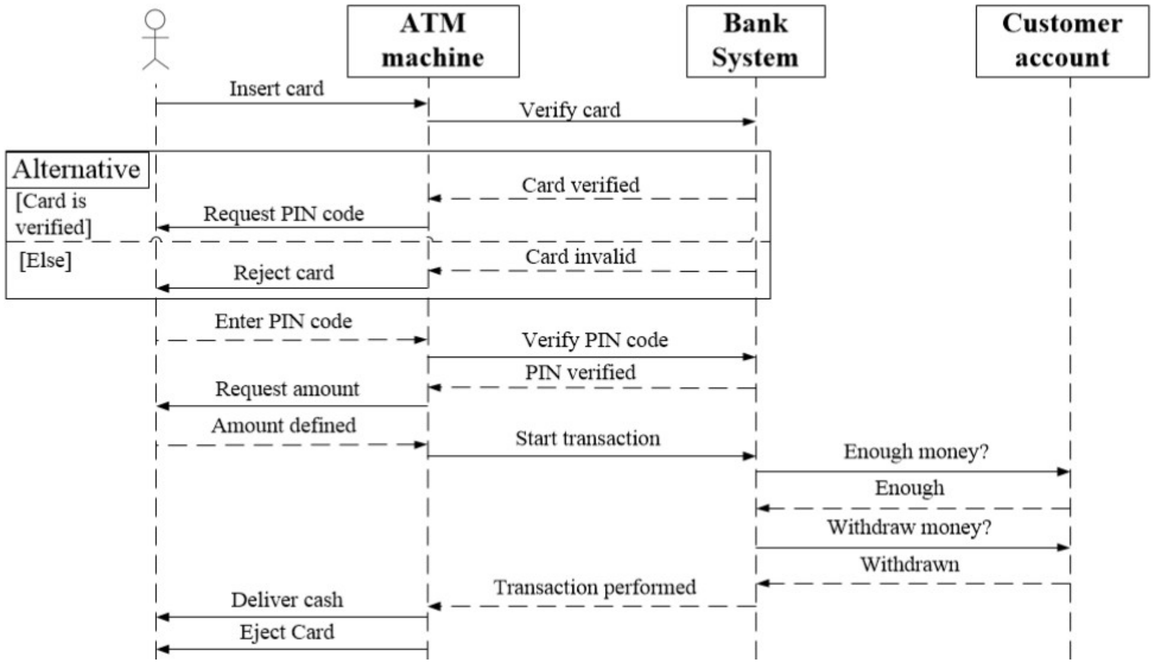
Major elements of the UML use case diagram are shown on the picture below.



Sequence Diagram

Captures **interactions** among entities during time. Sequence diagrams use **constructs** to represent the entities participating in the diagram as well as the **messages exchanged** between them during the execution of the **use case**.

Sequence diagrams usually models what happens for a single use case.



Vertical lines are called “**life lines**”.

Very useful to model communication **protocol** evolution and interactions.

Domain Specific Modeling Language (DSML)

DSML should provide the **required constructs** to model the main aspects of the **target domain**.

Required because a single modeling language is not enough to cover various specific areas of concern.

DSML provide constructs that are directly aligned with the concepts of the domain in question.

In a complex system (of systems) if each individual subdomain uses a different ad-hoc modeling language, to verify and test the system as a whole we have to resolve the problem of interfacing each individual model.

UML is a standard highly adopted by academia and industry.

UML was designed as a general purpose modeling language as well as a foundation for deriving different domain specific languages, mainly through its **profile** mechanism. The profile concept allow derivation of DSML from its set of general language concepts (UML is tailored to the domain needs).

Profiles allow to define domain-specific concepts as extensions or refinements of existing UML **metaclasses**. These extensions are called **stereotypes**.

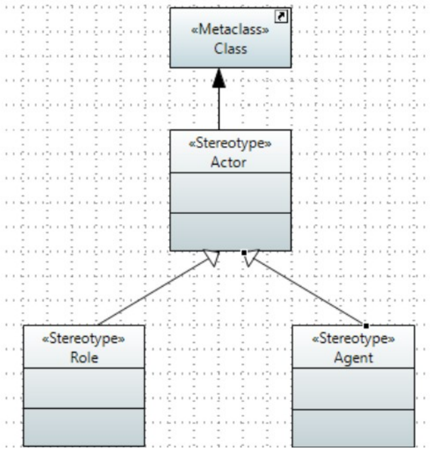
A profile-base model can be created and modified easily by any tool that supports standard UML.

Stereotypes are defined either as extensions of a UML base metaclass or as specialization of existing stereotypes.

A stereotype may have attributes and may be associated with other stereotypes or existing UML metaclasses.

A main advantage of the profile mechanism is that they can be applied and removed dynamically without modifying the underlying models.

Papyrus tool enables using a general purpose language (UML) to define a DSML.



Once the stereotype is created and the meta-class imported, you can model the **extension** relation from the stereotype to the meta-class. Stereotype **generalization** allows extending existing stereotypes to define other stereotypes possibly within a single profile.

Object Constraint Language (OCL)

Developed by IBM in 1995 and proposed to OMG (Object Management Group), has been integrated in UML in 1997.

Defined as a declarative language describes rules to be applied to UML models.

Developed to overcome some of the limitations of UML in terms of precisely specifying detailed aspects of a system design.

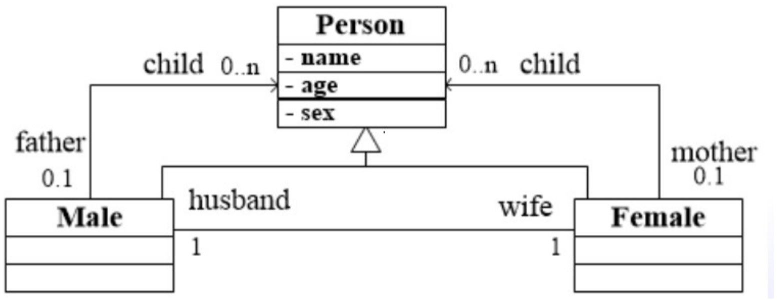
Provides **constraint** on models and meta-models that cannot be expressed using the graphical notation.

OCL is a key component of the new OMG standard recommendation for transforming models.

To keep graphical modeling language simple and manageable, the concepts they can represent has been reduced to the minimum set of modeling constructs. Thus the graphical notations such as plain UML can only represent a limited subset of the aspects of a domain.

OCL try to fill this gap by covering some aspects not captured by UML.

Some constraints can be expressed in UML (e.g. a person has one mother/father). Others cannot (A person cannot be his own father).



OCL is the only constraint language that is **standardized**.

More informative (descriptive) models: add to the UML model more rich relationships.

Reduction of **confusion** and **ambiguity** among developers and stakeholders with a well-defined formal semantics.

Constraints:

- Can also be used in a profile.
- Can be defined and applied to stereotypes.
- Can be used to further constrain primitive elements of the UML meta-model.

For example, OCL can control that all instances of a class in a model must have at least one operation.

However not all constraints can be written in OCL. In such a case extra constraints are written in natural language (drawback: they are not automatically interpretable).

Syntax

Context : is used to bound a constraint to a specific modeling entity (e.g. a class).

Invariant : contains a boolean expression that should be always satisfied.

Self : used to denote the context object.

```
context Female
inv: self.husband.age > 18
```

Additional notes

OCL is a specification language and not a programming language.

OCL is a typed language and it proposes a set of predefined types.

It is not allowed to compare different types.

The use of OCL constraints on operations is problematic.