Requirements Modelling and Software Systems Implementation Using Formal Languages

Radek Kočí

Brno University of Technology, Faculty of Information Technology
Czech Republic

koci@fit.vutbr.cz

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

ICSEA 2018, 15.-18.10.2018, Nice, France

Outline

- **Software and Software Engineering**
- **Requirements Specification**
- **Formal Methods**
- **Model Driven Engineering**
- **Simulation Driven Development with Petri Nets**

What is software (system)

- requirement specification
- design specification
- source codes including comments
- executable programs
- reference/operation manuals
- validation/verification documents
- . . .

Software

- is a set of documents
- its properties are described with informal / semi-formal / formal languages
- how to validate documents against user's real needs?

- other engineering (mechanical, . . .) handles already established problem domains
- software engineering should handle any problem domain
- modeling, formalization and analyses of problem domains are major tasks

⇒ domain / requirements engineering

Software engineering

- is an discipline for studying methods to translate problem domain, i.e., semantics, into machine domain, i.e., forms
- how to validate that forms against user's real needs?

Requirements Specification

What can be used to requirements specification

- unrestricted natural languages
- structured natural languages
- semi-formal specification languages
- formal specification languages

User Requirements (needs)

- the initial user requirements are always informal
- it is not possible to prove that any specification satisfies user requirements (needs) – only the user can say
- requirements specification has to be clear and readable for users

Informal Requirements Specification

Informal specification

- described in natural languages with diagrams or pictures
- has no limits in the expressions used and usually does not require special preparation
- on the other hand, it is prone to vague expressions, ambiguities, and unmeasurable statements (difficult to evaluate accuracy)

Predefined expression patterns

- simplify the creation of requirements using standardized form of statements
- simpler document passage, fulfillment criteria, etc.

Decision tables

- the original claim is divided into a set of conditions
- it is possible to examine behavior in all variants
- a set of simple claims that eliminates ambiguous understanding

Inp. cond.	Train accepts an acceleration command	Y	Y	Y
	The train passes the signal too fast	Y	Y	N
	The previous train is closer than X meters	Y	N	Y
Outp. cond.	Braking activated.	X		X
	A station alarm is generated.	X	X	X

Semi-formal languages

- replacement for natural language or its supplementation
- usually captured in a form of schemes or diagrams (diagrammatic notation)
- semi-formal: elements of systems and their relationships are declared formally, but the statements describing their properties may be specified informally (structured natural language)
- syntax and semantics of elements are formally defined, it is enabled automatic processing the specification (consistency checking, partially transformations etc.)
- system properties are described informally, the full analysis/simulation/transformation not allowed

Semi-formal models

- Entity-Relationship Diagram
- Use Case Diagram
- Interaction Diagram
- Statecharts (State Diagram)

Formal specification

- predefined rules for determining the meaning of specifications
- written in formal languages
- supported by tools

⇒ enable rigorous software development

Formal description

- specifying requirements and desired properties
- modeling internal behavior
- the description is typically at certain level of abstraction
- precise, consistent and unambiguous

Formal languages

- algebraic specification techniques (CASL)
- rewriting systems (OBJ3)
- Model-oriented languages (Z, VDM)
- UML + OCL; MOF + Alf language
- Petri nets
- logics
- ...

Formal specification

- formal specification let designers use abstractions and reducing the conceptual complexity of the system under development
- formal specification formalizes the statements describing element properties
- precise formulation of statements permits machine manipulation
- a more sophisticated form of validation and verification that can be automated using tools
- the specification may be mechanically transformed into another one, more detailed than its predecessor, and, eventually, into executable program

Formalization properties

- formal methods can be beneficial even if no formal verification is used at all – since since the rigorous specification is required the designer has to do the job more thoroughly, reaches a better understanding of the problem and it leads to better solution
- can be difficult to understand not only for users but also for developers
- a formally verified program is only as good as its specification
- it is very easy to create a wrong specification that does not meet the user needs (requirements)

Formal methods properties

- The greatest benefit in applying formal methods often comes from the process of formalization rather than from its outcomes.
- Formal methods do not guarantee correctness.
- Can be difficult to understand not only for users but also for developers.
- How to demonstrate that the specification meets the user needs?

Summary

- How to validate documents/formalized documents against user's real needs?
  - only the user can say
  - a combination of the formal notation and prototyping
- Formal methods can be difficult to understand
  - requirements specification has to be clear and comphrehensible to users as well as developers
  - a possibility of formal notation as well as graphical modeling

$\Rightarrow$ formal models that can be simulated, graphically represented, and formally processed

Principle

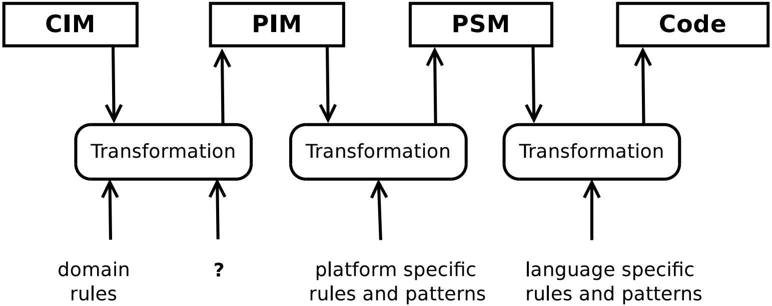
- the essential outputs of the development process are models instead of the program
- the program (code) should be (automatically) generated from models
- it is possible to highlight some aspects of the developed system without having to deal with the implementation details
- different levels of abstraction

Model Driven Architecture as an implementation of MDE

- different levels of abstraction
  - Computation Independent Model (CIM) – the business requirements for the system
  - Platform Independent Model (PIM) – describes software functions and is independent of realization details
  - Platform Specific Model (PSM) – is combined with technical details of platform for realizing system
- used models
  - use case diagrams
  - class diagrams
  - statecharts
  - activity diagrams

Transformation

- the lower the abstraction (the closer to the design), the transformation mechanisms are more sophisticated
- Can CIM be formal models? Is it possible to automate the CIM-PIM transformation?
- Is it possible to change model and propagate changes to higer abstraction models?



Motivation

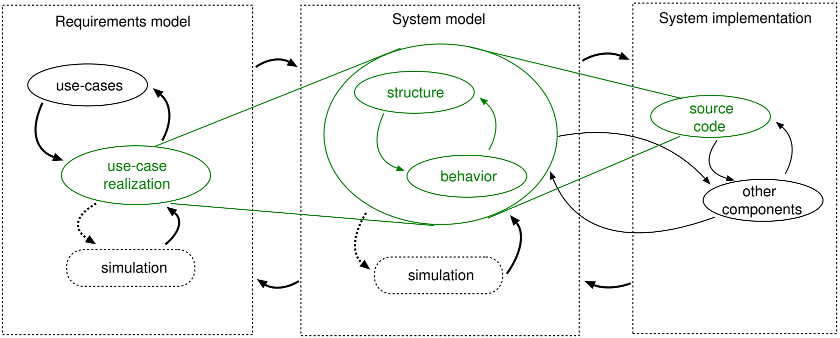
- reduce the gap between real needs and specified needs to sofware system under development
- combination of semi-formal and formal models
- formal and executable models showing a sketch of the system to help visualize what the system will do

Model continuity

- elimination of the overhead caused by creating models at different level of abstraction
- continuous incremental development of models
- models can work in live system
- no need of implementation or code generation

Essential parts of the systems are presented through simulation (formal) models

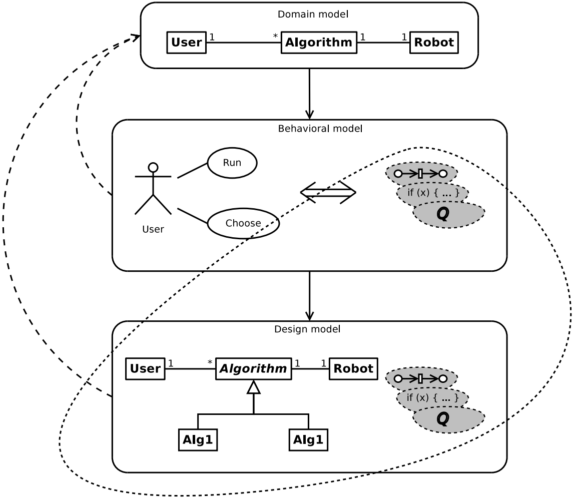
- requirements model = CIM
- system model = PIM + PSM



Principle

- Domain model – captures the concepts of the domain system as identified and understood
- Behavioral model – captures an external view of system functionality, its behavior, and interaction with the surroundings
    - User requirements modeling – use cases
    - Scenario modeling – behavior and interactions of individual cases
- Design model – sophisticated domain and behavioral models, more details

Simulation Driven Development



Principle (cont.)

- Scenarios at the behavior level coincide with scenarios at the design level and are no longer distinguished.
- Continual development of behavior models becomes design models, which serve simultaneously for specification purposes.
- Design models can contain other objects from the domain environment to simulate the system or run under real-world conditions without having to show this implementation details at the requirements or behavior model.
- The same model can therefore be used for both documentation requirements and the executable version (prototype, implementation) of the developed system.

Use Case

- it models a sequence of interaction between actors and the system (=scenario)
- there is a main sequence, which can be supplemented by alternative sequences for less commonly used interactions

Formalism of OOPN (Object-Oriented Petri Nets)

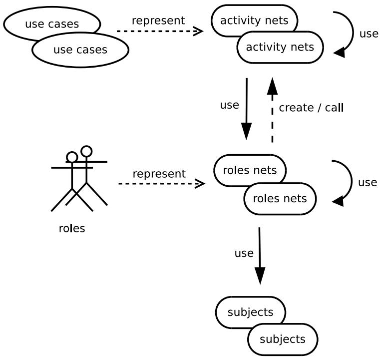
⇒ clear formal syntax
⇒ clear semantics
⇒ usable by developers having no power mathematical backgroud
⇒ Petri nets are also a simulation model
⇒ Petri nets can be executed in real environment

⇒ models scenarios of use case diagram elements
⇒ the behavior description can contain parts of code (prg. language)

Identification of elements

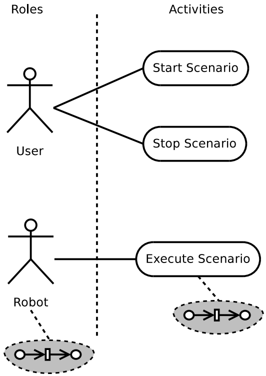
Identification of model elements from the use case model

- use case ⇒ activity net
- actor ⇒ role
- more actors can have the same basis ⇒ subject

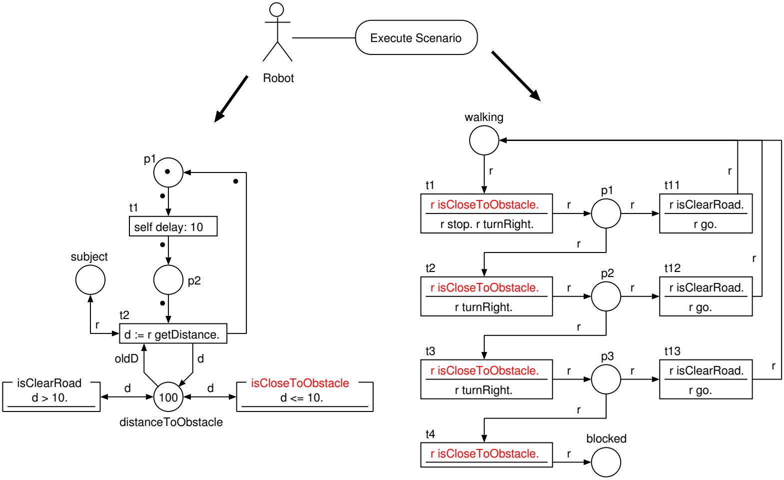


A simplified example of a robot control system

- the use case model and
- identification of roles and activities.



- actor Robot ⇒ role Robot
- use case Execute Scenario ⇒ activity net Scenario

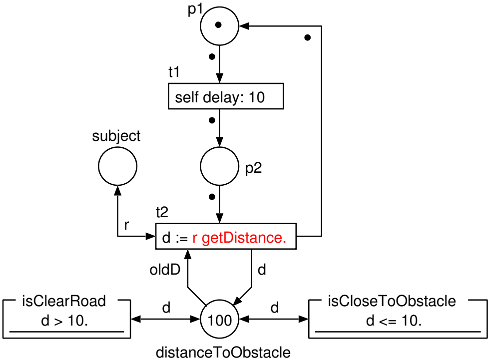


Role *Robot*

Activity net *Scenario*

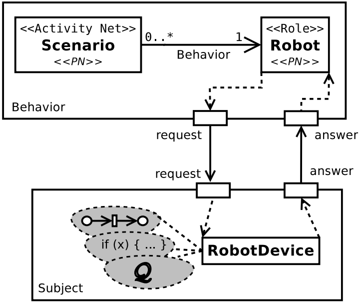
Role Robot uses a subject, which can be defined in different ways

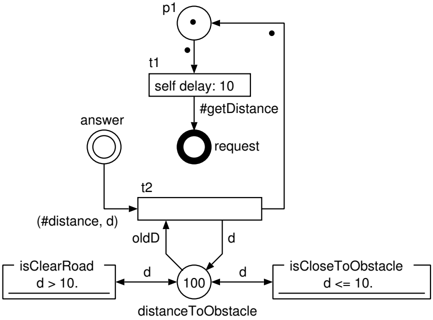
- modelled by OOPN, statecharts, . . .
- implemented in a programming language



System Development – DEVS Architecture

- second way of components connection
- message passing ⇒ data carrying through ports
- looser links between components ⇒ easier changing of components
- no dependence on a component realization (methods, predicates, . . .)





Conclusion

We have set several conditions the formalism has to satisfy to be suitable for requirements modeling and realization

- formal notation – an unambiguity of the specification
- a possibility to validate specification against real needs using prototyping or simulation
- specification has to be comphrehensible to users as well as developers; graphical modeling allowed
- a possibility to keep models during entire development process

Conclusion

We have set several conditions the formalism has to satisfy to be suitable for requirements modeling and realization

- formal notation – an unambiguity of the specification
- OOPN, DEVS

- a possibility to validate specification against real needs using prototyping or simulation
- OOPN, DEVS combining with product objects

- specification has to be comphrehensible to users as well as developers; graphical modeling allowed
- OOPN, DEVS combining with use cases, classes, . . .

- a possibility to keep models during entire development process
- OOPN, DEVS combining with use cases

Thank you for your attention!