Verification of the Requirements Specification

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Overview

- Inputs and outputs of the phase
- Preparing the requirements specification
  - Formal languages
  - Semi-formal and structured methods
  - Example: SysML
- Verification tasks
  - General aspects and verification techniques
  - Verifying completeness and consistency
- Managing requirements
  - Traceability
  - Basic tasks and tool support

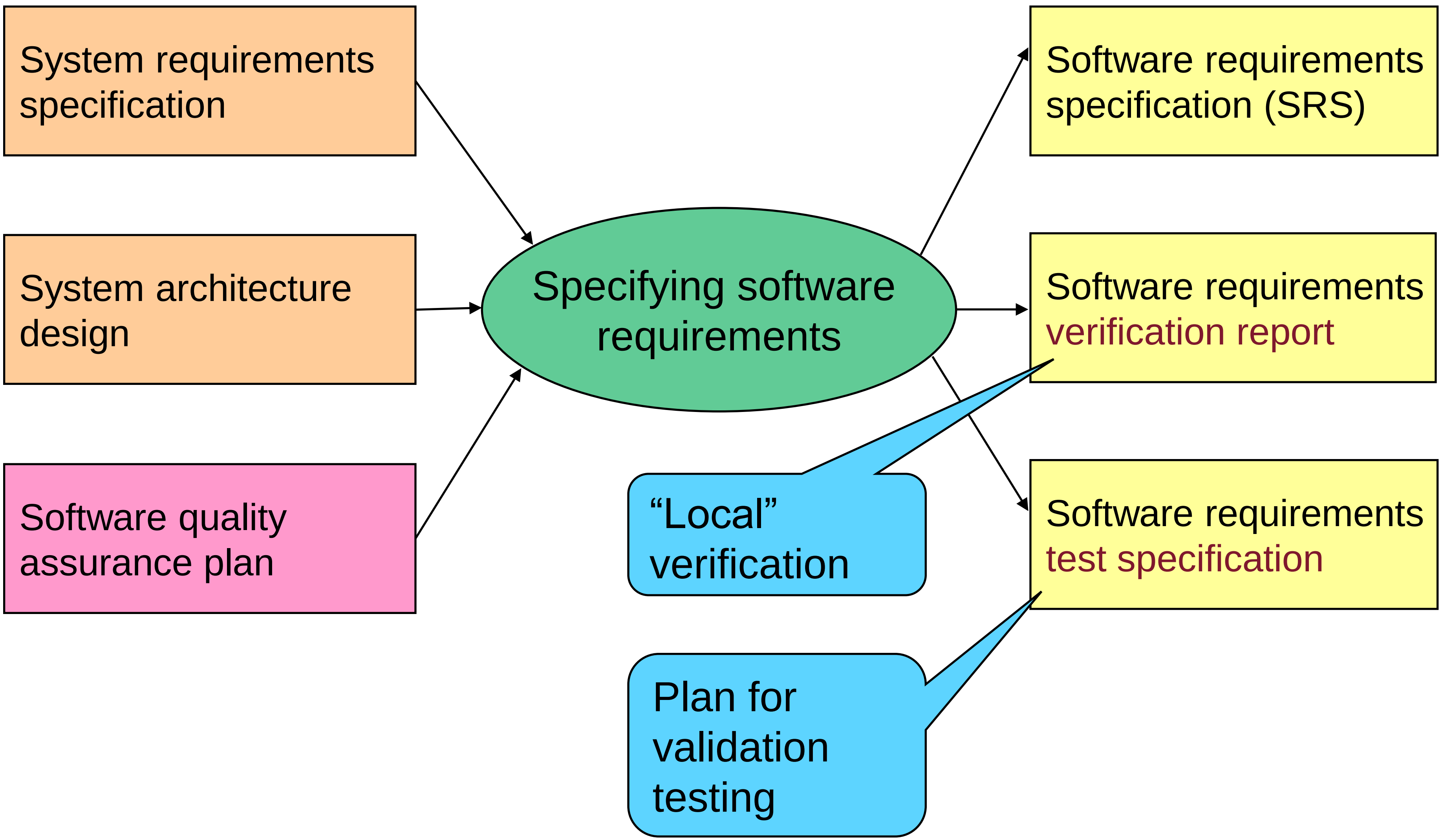
Inputs and outputs of the phase

Inputs and outputs

Related: Software Quality Assurance Plan

and Software Verification Plan

Inputs and outputs of the phase



Software Quality Assurance Plan

- Goals:
  - Preventing systematic faults and controlling residual faults
  - Determining the required technical and control activities
- Main aspects to be included:
  - Activities, their input and output criteria in the lifecycle
  - Quantitative quality expectations (e.g., ISO/IEC 9126)
  - Specification of its own review and maintenance
- Methods for checking external suppliers
  - Compliance of the QA Plan of the supplier
  - Verification of external software components
- Issue tracking
  - Documentation and feedback mechanisms
  - Analysis of issues (root causes)
  - Diagnosis and maintenance/repair activities and techniques
  - Verification and validation of corrections
  - Fault avoidance

Software Verification Plan

- Often a separate plan (especially in safety-critical systems)
- Planning the verification activities
  - Planning the techniques and measures (from the development standard)
  - Determining acceptance criteria
- Overall aspects of verification:
  - “Local” checking of the given development step: Completeness, consistency
  - Conformance checking: W.r.t. the output of previous phases
- Details:
  - Participants roles and responsibilities
  - Tools (e.g., test equipment)
  - Evaluation of verification results (acceptance criteria)
    - Checking the required test coverage
    - Evaluation of quality requirements

Software requirements specification - Terminology

- Requirement
  - Incoming need, vision, expectation
    - From the future users
    - From stakeholders (management, operator, authority, ...)
  - Basis for validation
- Requirement specification
  - Requirements in converted form, for the designers
    - Result of requirement analysis
    - Abstraction, structuring, filtering applied
  - Several types of requirements
    - Property specification, behavior specification, …
    - Later: architecture specification (/design), module specification, …
  - Basis for verification

Preparing the requirements specification

Formal languages

Semi-formal and structured methods

Example: SysML

Approaches for specifying requirements

- Contents of the requirements specification
  - Functional requirements + Extra-functional requirements
- Typical: Natural language based specifications
  - Problems with unambiguity, verifiability
- Possible solutions:
  - Using strict specification language (e.g., formal, or semi-formal)
  - Using verified “specification patterns” (e.g., for safe behavior)
  - Systematic verification after the requirement specification phase
- Example: Solutions proposed by EN 50128
  - Formal methods (VDM, Z, B, TL, PN, ...)
  - Semi-formal methods (diagram based techniques, SysML)
  - Structured methods (JSD, SADT, SSADM, …)
  - Natural language based description (explanation) is mandatory

Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, …)
- Algebraic languages (ADT, OBJ, …)
- Process description languages (CSP, CCS, …)
- Logic languages (HOL, CTL*, …)
- Constructive languages (NUPRL, …)
- Hybrid or wide spectrum languages (CPN, E-LOTOS, …)

Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, ...)
- Algebraic langua es
- Process desc on lan
- Logic lan
- L, ...)
- LOTOS, ...)

Mathematical model:

- Elements in the system (set-theoretic structures like sets, subsets, relations)
- Functions, operations, events (with pre- and post-conditions, invariants)

Example: Specification of an access control system (in Event-B):

Persons:	`prs ≠ 0, p ∈ prs`	(set)
Buildings:	`bld ≠ 0, b ∈ bld`	(set)
Authorization:	`aut ∈ prs ↔ bld`	(binary relation)
Situation:	`sit ∈ prs → bld`	(complete function)
Invariant:	`sit ⊆ aut`	

An event (change of situation):

```
pass = ANY p,b WHERE (p,b)∈aut ∧ sit(p)≠b
    THEN  sit(p):=b END
```

Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, ...)
- Algebraic languages (ADT, OBJ, ...)

Abstract data types: sorts (set of values), operations, properties as equations

```
Type Boolean is
    sorts Bool
    opns
        false, true :        -> Bool
        not : Bool           -> Bool
        and : Bool, Bool     -> Bool
    eqns
        forall x, y: Bool
        ofsort Bool
            not(true) = false;
            not(false) = true;
            x and true = x;
```

Abstract algebra and category theory

- Abstract data types: values, operations, properties
- First order logic is typical

...anguages

(CPN, E-LOTOS, ...)

Overview of the types of formal languages

- Model-oriented langua
- Algebraic languages
- Process description languages (CSP, CCS, …)
- Logic l
- Cons
- Hybr

- Processes: Sequential execution of statements
- Operations among the processes (synchronization, communication)

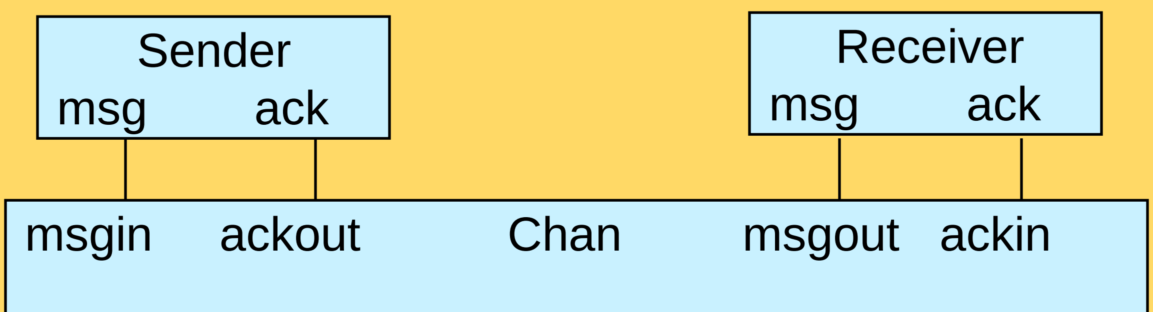
Example: Process algebra language (CCS):

Sender = $\overline{\texttt{msg}}$.ack.**Sender**

Receiver = msg.$\overline{\texttt{ack}}$.**Receiver**

Chan = msgin.$\overline{\texttt{msgout}}$.**Chan** + ackin.$\overline{\texttt{ackout}}$.**Chan**

Proc = **Sender**[msgin/msg,ackout/ack] | **Chan** |
Receiver[msgout/msg, ackin/ack]



Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, …)
- Algebraic languages (ADT, OBJ, …)
- Process description languages (CSP, CCS, …)
- Logic languages (HOL, CTL*, …)
- Constructive lang
- Hybrid or wide spec

.)

- Formal mathematical logic (first order or higher order logic)
- Temporal logics (with temporal operators like “future”, “next time”, “until”, “before”)

Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, ...)
- Algebraic languages
- Process description l
- Logic languages
- Constructive languages (NUPRL, ...)
- Hybri

Constructive logic systems (computable functions): Proof of a property of a function at the same time provides a construction (implementation)

Example for a non-constructive proof (in mathematics)

- The existence of an artifact with a given property can be proven without giving exactly what is that artifact
  - Example: There exist $a,b \notin Q$ such that $a^b \in Q$
- Properties with non-constructive proof are not feasible for software specification, this way restrictions are needed that guarantee the synthesis of functions

Overview of the types of formal languages

- Model-oriented languages (VDM, Z, B, …)
- Algebraic languages (ADT, OBJ, …)
- Process description languages (CSP, CCS, …)
- Logic languages (HOL, CTL*, …)
- Constructive languages (NUPRL, …)
- Hybrid or wide spectrum languages (CPN, E-LOTOS, …)

> - Properties and advantages of different formalisms are combined, e.g.,
>   - LOTOS: process algebra + ADT
>   - CPN: Petri-nets + data manipulation (ML)

Semi-formal languages: Examples

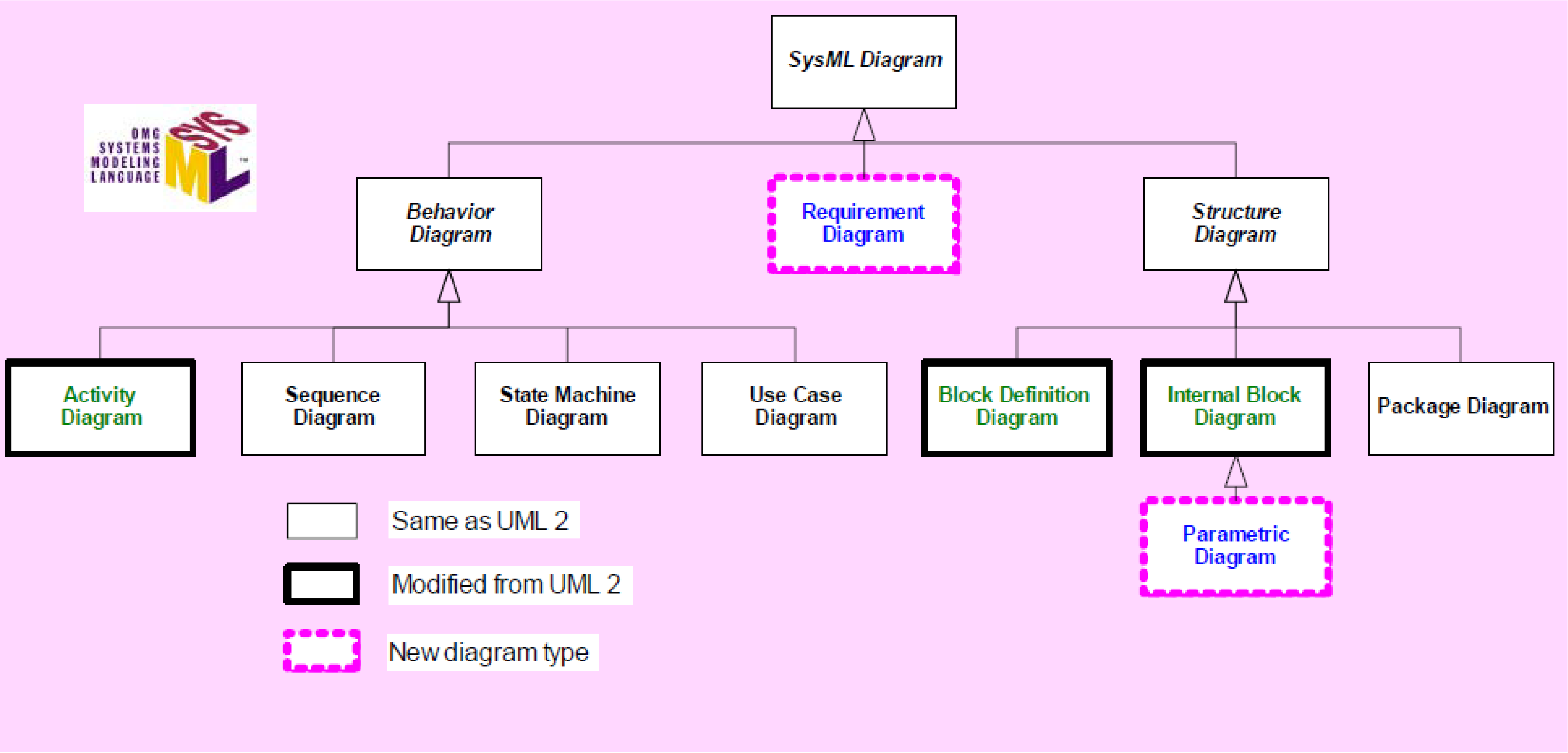
- Description of the structure:
  - (Functional) block diagrams
- Description of data flow:
  - Data flow diagrams, data flow networks
  - (Message) sequence diagrams
- Description of the control flow:
  - Control flow diagram, state machine, statechart
- Description of logic conditions:
  - Truth tables
  - Constraint languages (e.g., OCL with structure)

Structured methodologies: Historical examples

- Jackson System Development (JSD)
  - Entity structure: Entities + actions (ordering) + processes
  - Network: Communicating sequential processes
- Real-time Yourdon (Ward-Mellor)
  - Basic: Environment (input events) + behavior (response)
  - Construction: Processes (+ processors)
- SSADM
  - Data model (entity relationship diagram)
  - Data flow diagram (processes, data storage)
  - Entity diagram (life history)
  - Entity effects
- Structured Analysis and Design Technique (SADT)
  - Activity-factor diagram: tasks + relations;
    input, control, resource, output
- ROOM: Real-Time Object-Oriented Modeling

Semi-formal requirements specification: SysML

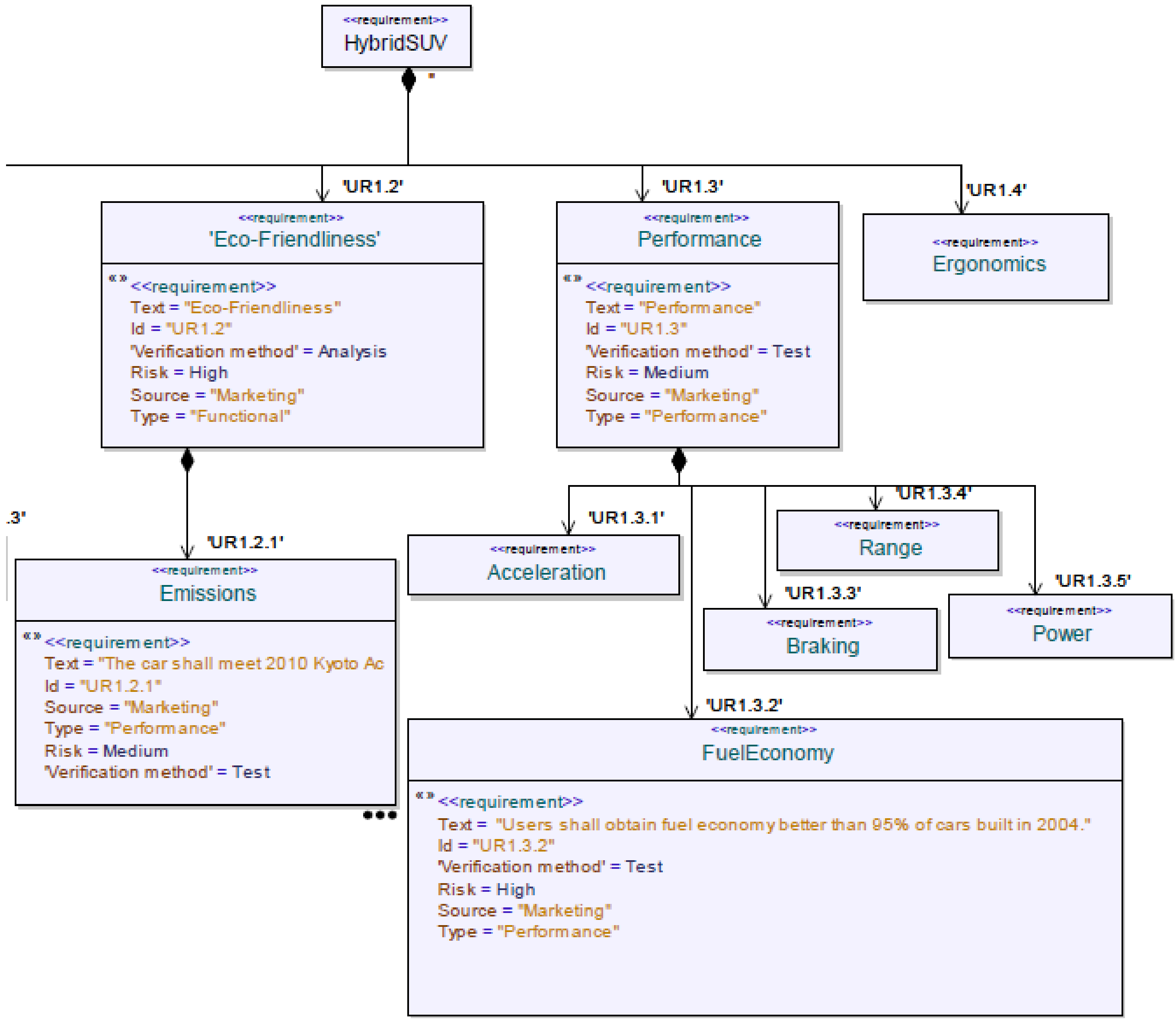
- Systems Modeling Language
  - UML subset and extensions for system modeling
  - Novelties: Requirement and Parametric diagram



Requirement diagram

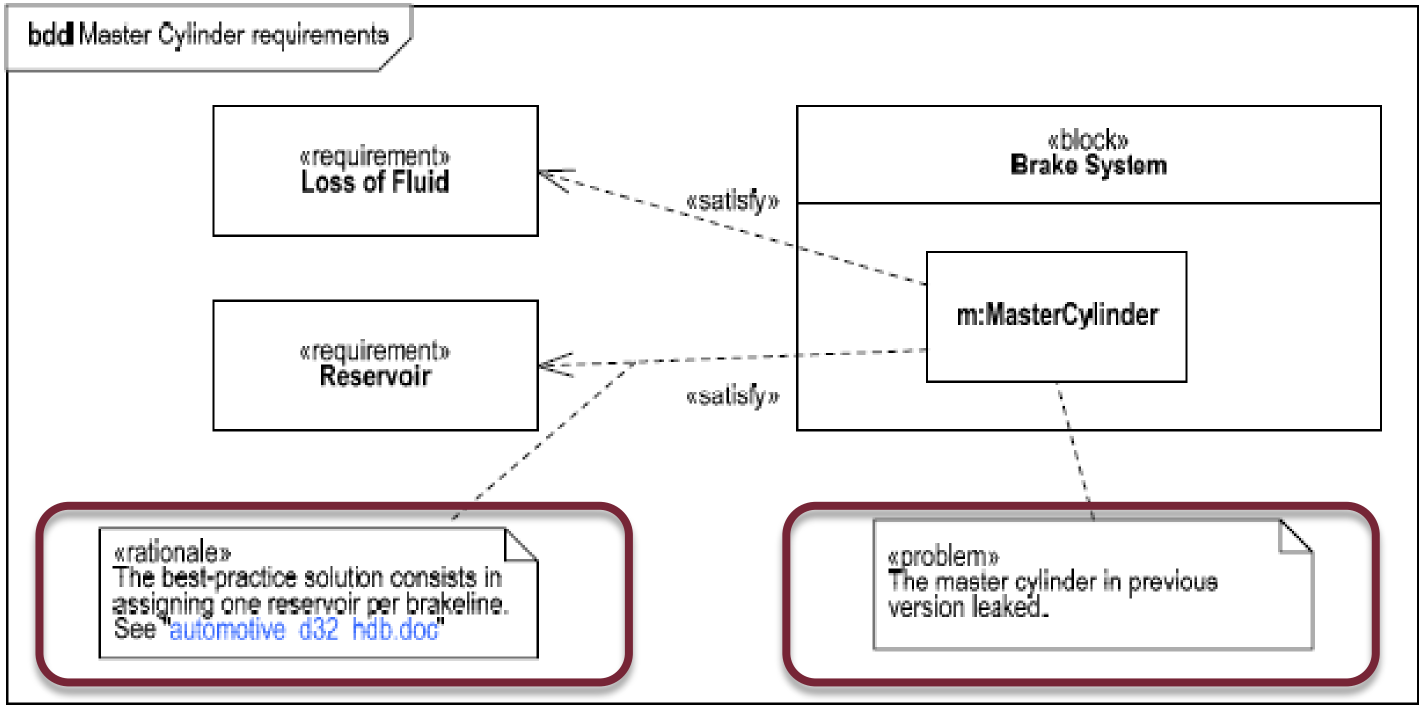
- Requirements (textual) with identifier are model elements
  - <<requirement>> stereotype
  - Id (identifier) and text (description) fields
  - User-specified attributes: e.g., type, source, risk, ...
  - Tabular form is also supported
- Requirements can be grouped into hierarchic packages
  - Functional, performance, etc. categories
- Refinement among requirements (~ subclass), composition
- Relations can be used (e.g., inserted as structured comments):
  - Copy: between requirements (master – slave)
  - Trace: between requirements (client – supplier)
  - DeriveReqt: between requirements (source – derived)
  - Refine: between requirements and design elements
  - Satisfy: between requirements and design or implementation elements
  - Verify: between requirements and test elements

Example requirements diagram: Structure

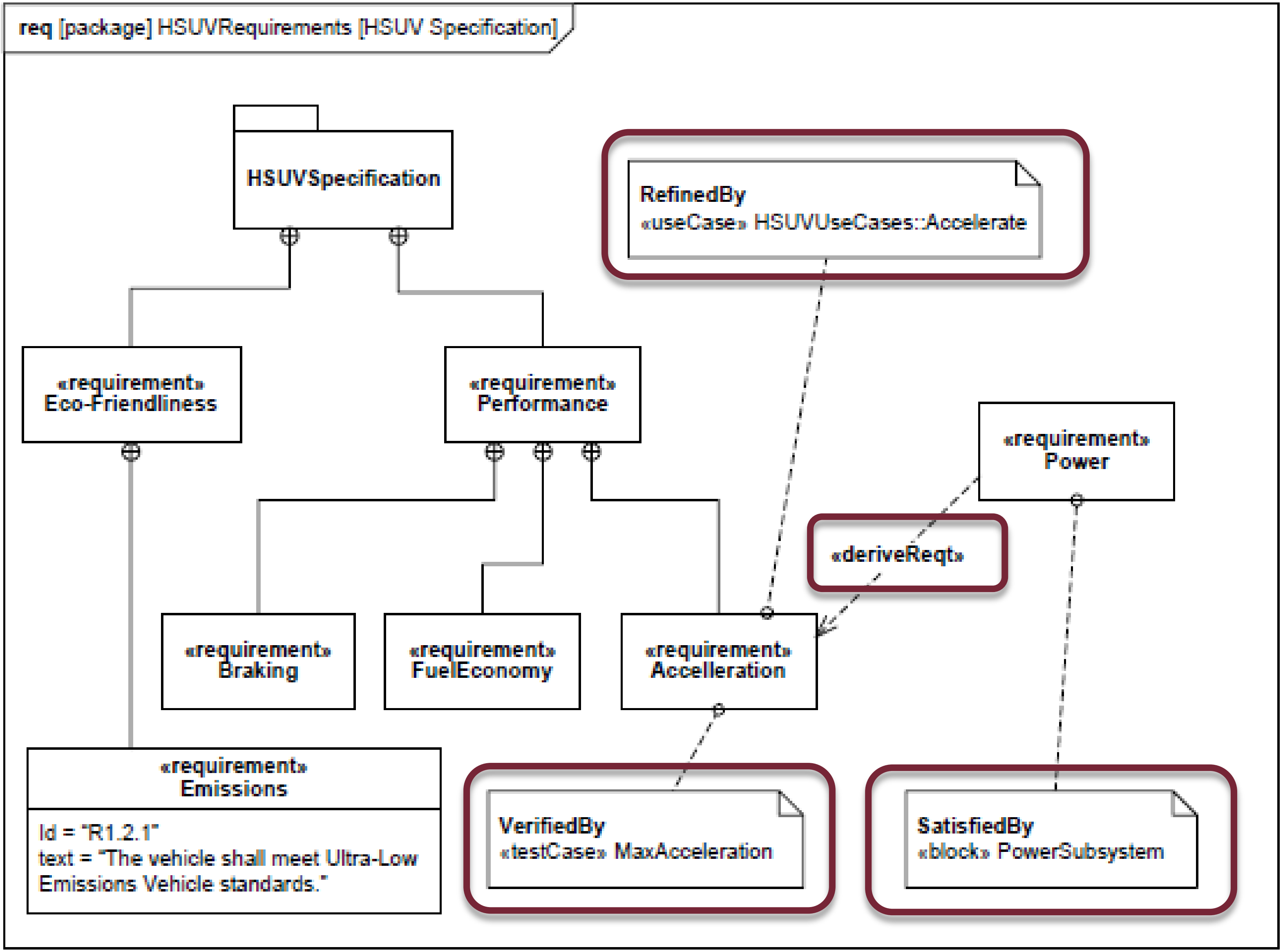


Requirements diagram: Decisions

- Special comments (with predefined stereotype) can be assigned to any model element:
  - <<problem>>: Problem or proposal that needs decision
  - <<rationale>>: Rationale, solution, explanation

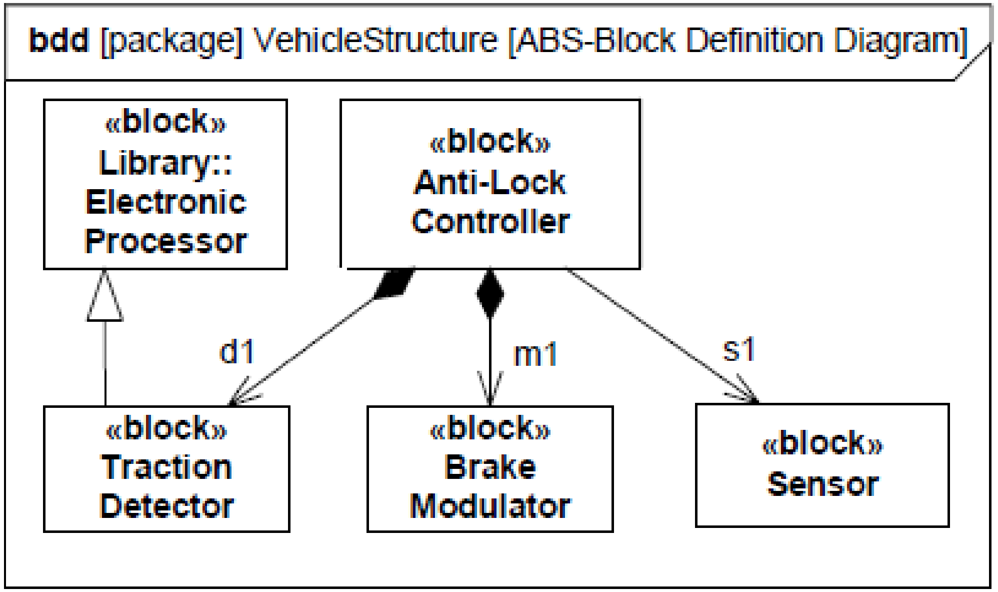


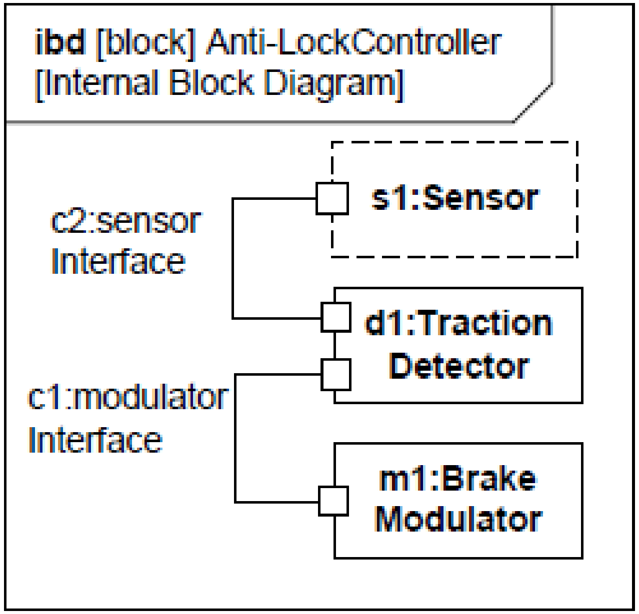
Example requirements diagram: Relations



Block diagram

- Block: Element of the structure (black / white box)
  - Component (not only software)
  - In SysML: Based on UML 2.0 classes
- Block definition diagram: Types of blocks
- Internal block diagram: Concrete roles of block types





Parametric diagram

- Goal: Verifiable quantitative requirements (constraints) expressed using attributes
  - Non-functional requirements
  - Supporting analysis (e.g., performance, reliability)
- ConstraintBlock: Specifying interrelations
  - Formal (e.g., MathML, OCL), or informal (textual)
  - Adapted to analysis tool (not SysML specific)
- Parametric diagram: Concrete application
  - Application of Constraint blocks in a given context
  - Binding between values

Parametric diagram: Example

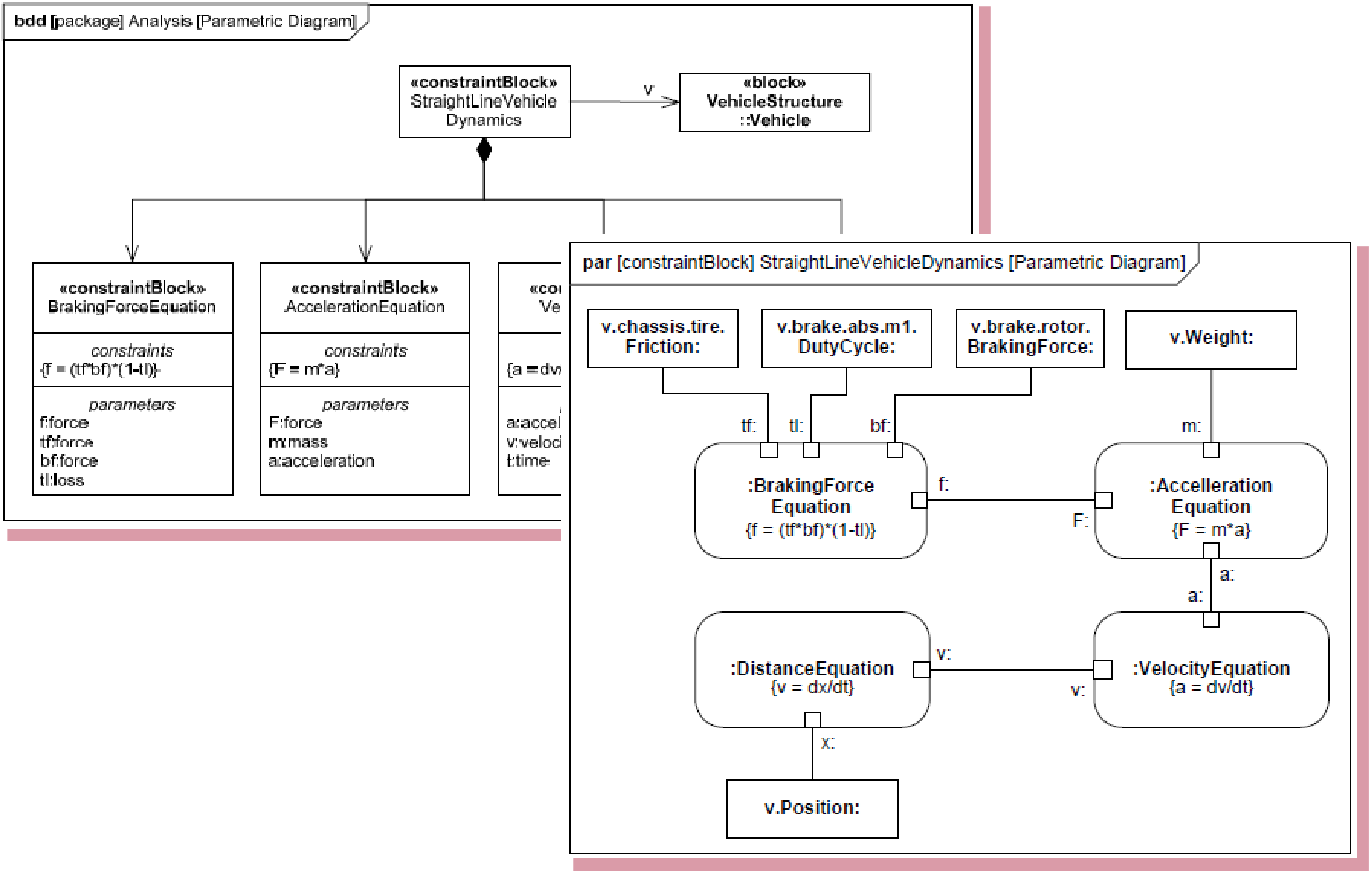
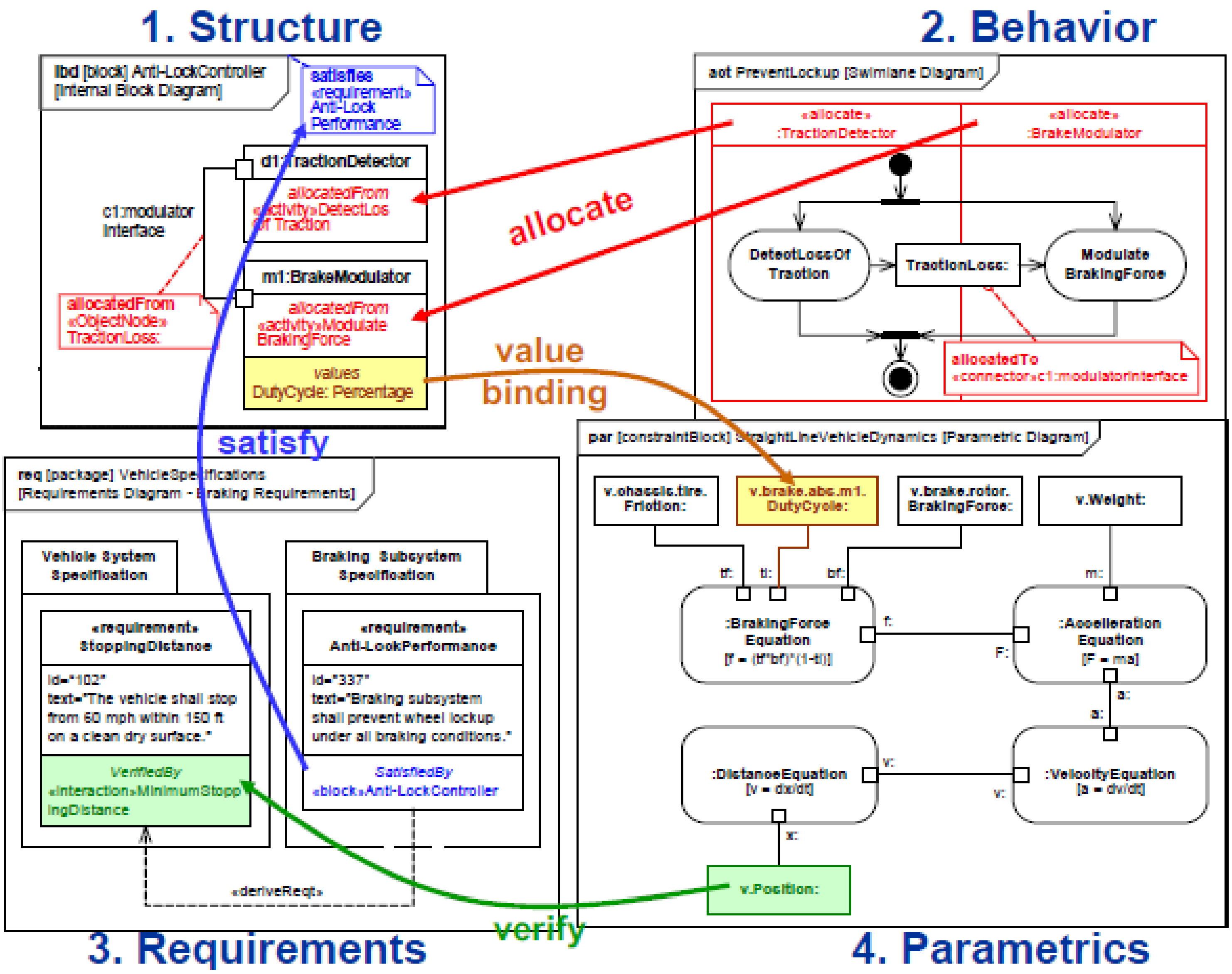


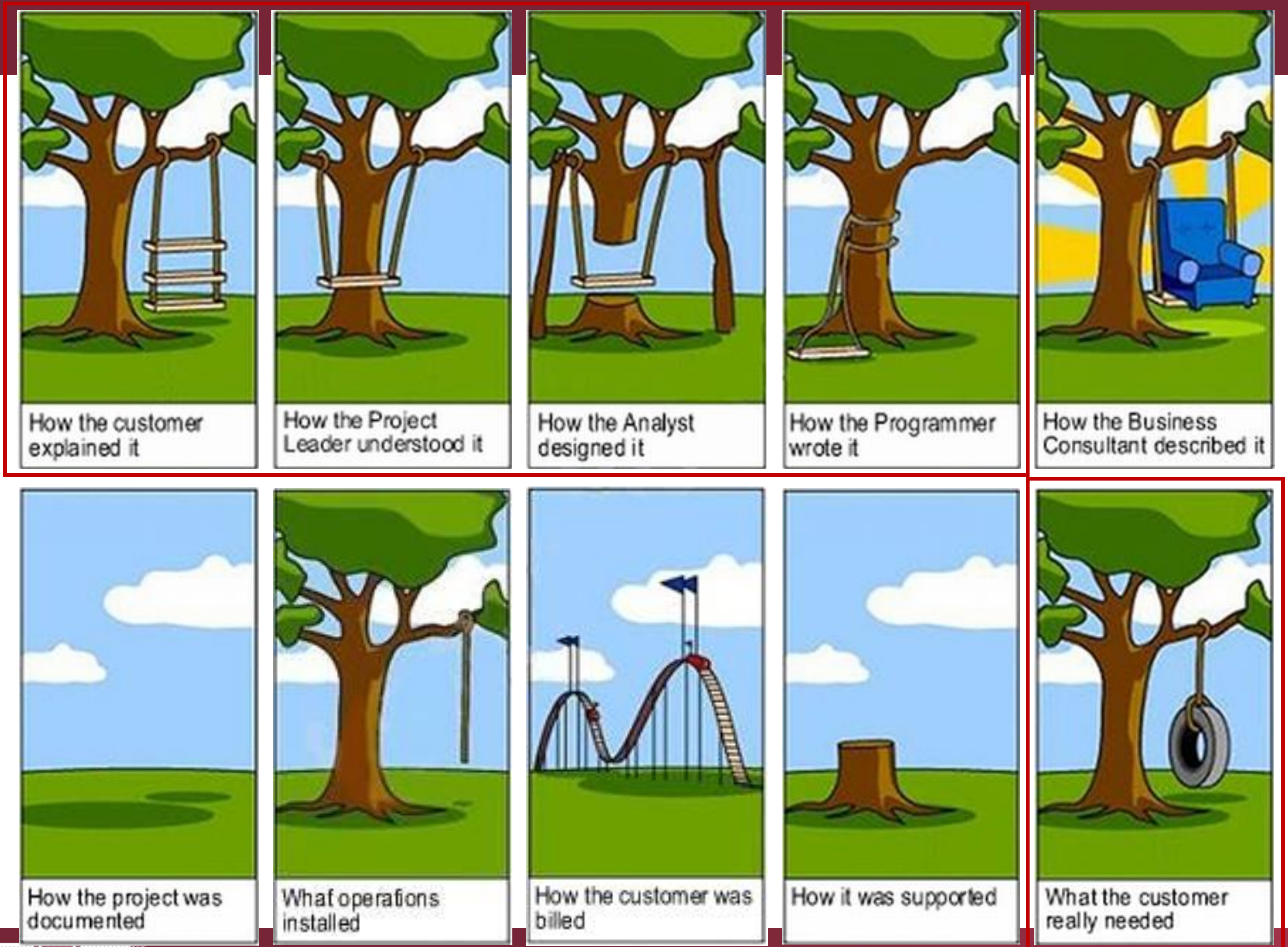
Illustration of the relations among diagrams



Verification tasks

General aspects and verification techniques

Verifying completeness and consistency


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

General criteria for a good specification

- Complete
  - Specified functions, references, tools, …
- Consistent
  - Internal and external consistency
  - Traceability
- Verifiable
  - Specific
  - Unambiguous
  - Quantifiable (if possible)
- Feasible
  - Resources
  - Usability
  - Maintainability
  - Risks: budget, technical, environmental

Example: Good specification on the basis of IEEE 830-1998

Correct

- Every requirement stated therein is one that the software shall meet
- Consistent with external sources (e.g. standards)

Unambiguous

- Every requirement has only one interpretation
- Formal or semi-formal specification languages can help

Complete

- For every (valid, invalid) input there is specified behavior
- TBD only possible resolution

Consistent

- No internal contradiction, well-defined terminology

Ranked for importance and/or stability

- Necessity of requirements

Verifiable

- Can be checked whether the requirement is met

Modifiable

- Not redundant, structured

Traceable

- Source is clear, effect can be referenced

Example: Good specification on the basis of IEEE 29148-2011

Necessary

- If it is removed or deleted, a deficiency will exist, which cannot be fulfilled by other capabilities

Implementation-free

- Avoids placing unnecessary constraints on the design

Unambiguous

- It can be interpreted in only one way; is simple and easy to understand

Consistent

- Is free of conflicts with other requirements

Complete

- Needs no further amplification (measurable and sufficiently describes the capability)

Singular

- Includes only one requirement with no use of conjunctions

Feasible

- Technically achievable, fits within system constraints (cost, schedule, regulatory…)

Traceable

- Upwards traceable to the stakeholder statements; downwards traceable to other documents

Verifiable

- Has the means to prove that the system satisfies the specified requirement

Techniques for verification

- Static analysis
  - Checking documents, code or other artifacts
  - Without execution
- Basis for static analysis: Checklists
  - Examples: Criteria for good specification
  - Completeness of the checklist is always questionable
- Implementation of static analysis
  - Manual review (all aspects)
  - Tool-support (esp. for checking consistency)


Manual review: Terminology and steps

Types of review:

- Informal review
  - No formal process
  - Peer or technical lead reviewing
- Walkthrough
  - Meeting led by author
  - May be quite informal
- Technical review
  - Review meeting with experts
  - Pre-meeting preparations for reviewers
- Inspection
  - Formal (well-documented) process
  - Led by a trained moderator

Steps of a review:

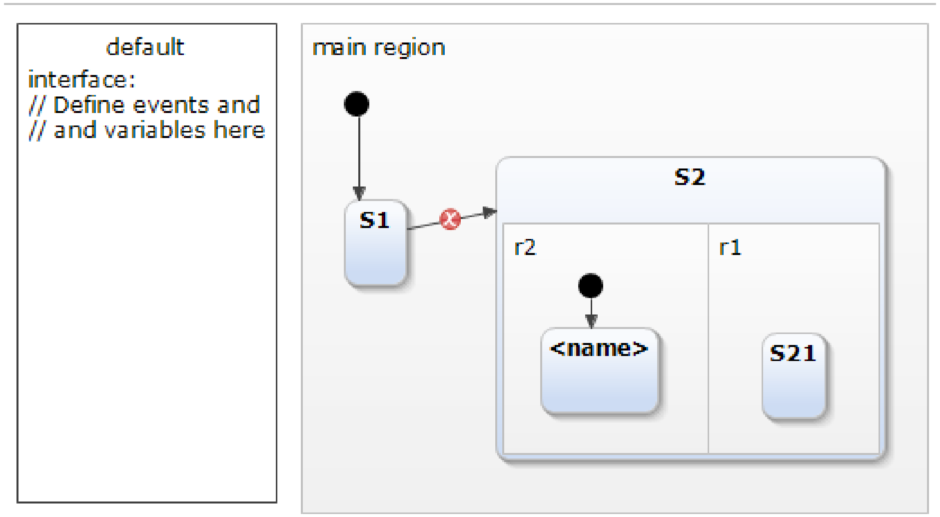
1. Planning
   - Defining review criteria
   - Allocating roles
2. Kick-off
   - Distributing documents
   - Explaining objectives
3. Individual preparation
   - Reviewing artifacts
   - Collecting defects, questions
4. Review meeting
   - Discussing and logging results
   - Making decisions
5. Rework
   - Fixing defects
   - Recording updated status
6. Follow-up
   - Checking fixes
   - Checking exit criteria

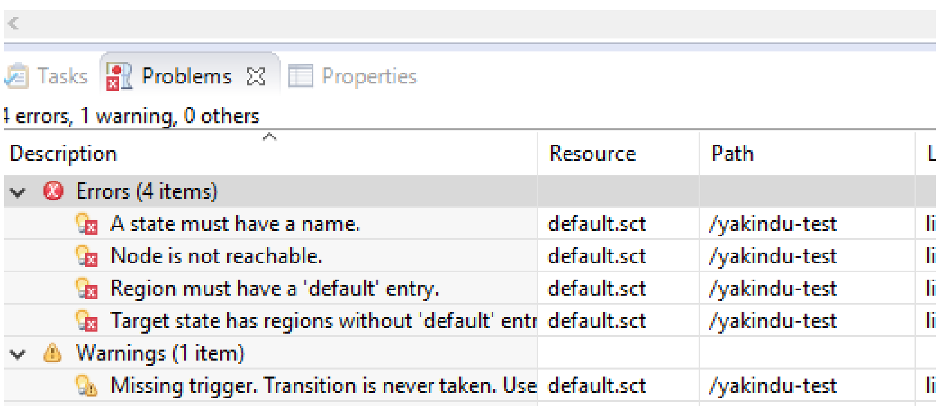
Tool support for verification of the specification

- Natural languages
  - No precise syntax and semantics
  - Static analysis by manual review
- Semi-formal languages
  - Precise syntax, but informal semantics
  - Automated checking of syntax and well-formedness (missing or contradictory elements)
- Formal languages
  - Mathematically precise syntax and semantics
  - Automated checking of syntax / well-formedness
  - Automated checking of behavior
    - Operational semantics: Reachable states of computation (e.g., model checking, equivalence/refinement checking)
    - Axiomatic semantics: Properties of computation (e.g., theorem proving for invariants, post-conditions)

Tool support: Checking state machines

Yakindu Statechart Tools



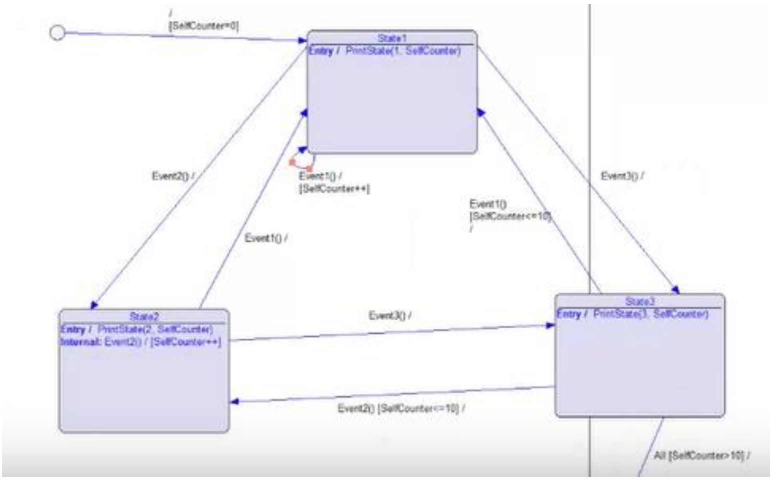

Tasks | Problems | Properties

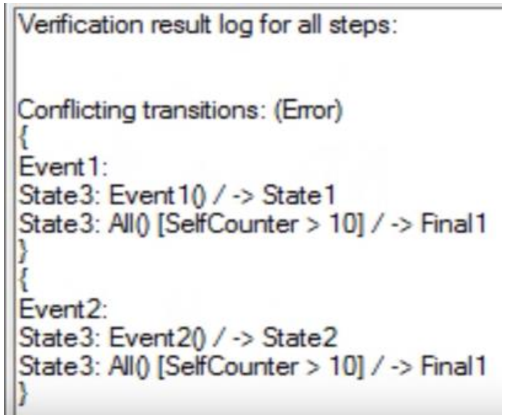
4 errors, 1 warning, 0 others

Description	Resource	Path	L
Errors (4 items)			
A state must have a name.	default.sct	/yakindu-test	li
Node is not reachable.	default.sct	/yakindu-test	li
Region must have a 'default' entry.	default.sct	/yakindu-test	li
Target state has regions without 'default' entr	default.sct	/yakindu-test	li
Warnings (1 item)			
Missing trigger. Transition is never taken. Use	default.sct	/yakindu-test	li

https://www.youtube.com/watch?v=uO6MASCBPrg

IAR visualSTATE




Verification result log for all steps:

Conflicting transitions: (Error)
{
Event1:
State3: Event1() / -> State1
State3: All() [SelfCounter > 10] / -> Final1
}
{
Event2:
State3: Event2() / -> State2
State3: All() [SelfCounter > 10] / -> Final1
}

https://www.youtube.com/watch?v=05lTlymLugM

Verifying completeness and consistency

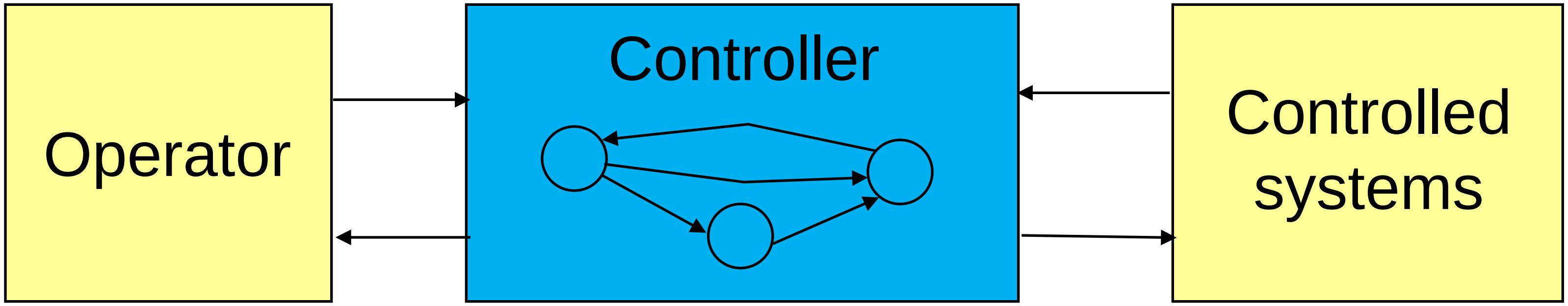
Incompleteness or inconsistency: major source of failures

- Statistics of faults found during the system testing of Voyager and Galileo spacecraft:
  78% (149/192) faults resulting from specification problem
  - 23%: missing state transitions (stuck in dangerous state)
  - 16%: missing time constraints for data validity
  - 12%: missing reaction to external event
  - 10%: missing assertions to check input values
- 60-70% of IT project failures can be traced back to insufficient requirements – Meta Group (2003)
- "Significantly more defects were found per page at the earlier phases of the software life cycle."
  - Inspection of 203 documents
  - An analysis of defect densities found during software inspections (JSS, DOI: 10.1016/0164-1212(92)90089-3)

Example: Review criteria for reactive systems

Groups of criteria (developed by N. Leveson, Safeware)

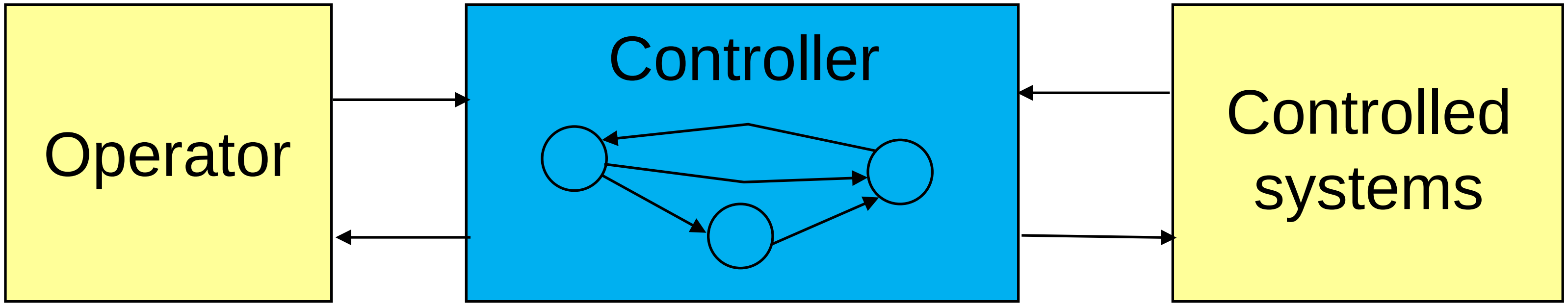
- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface



Example: Review criteria for reactive systems

- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface

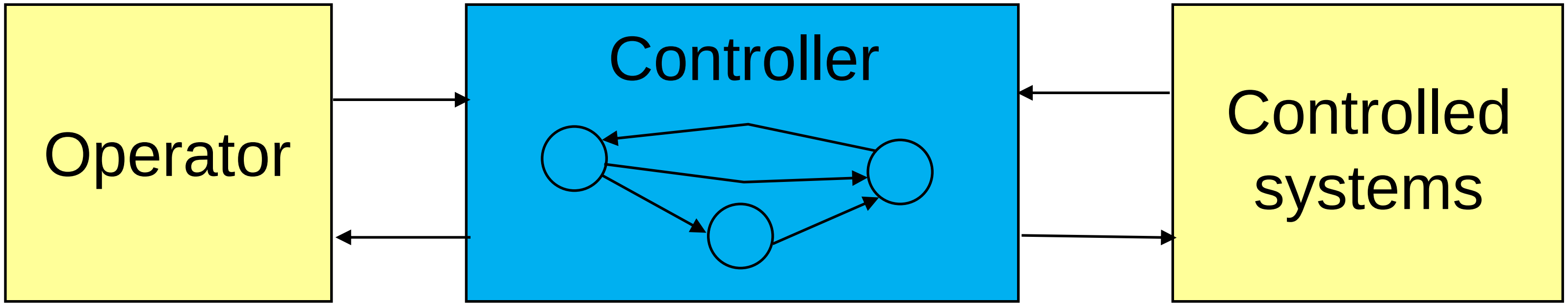
- Initial state is safe
- In case of missing input there is a timeout, and no action is allowed



Example: Review criteria for reactive systems

- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interfac

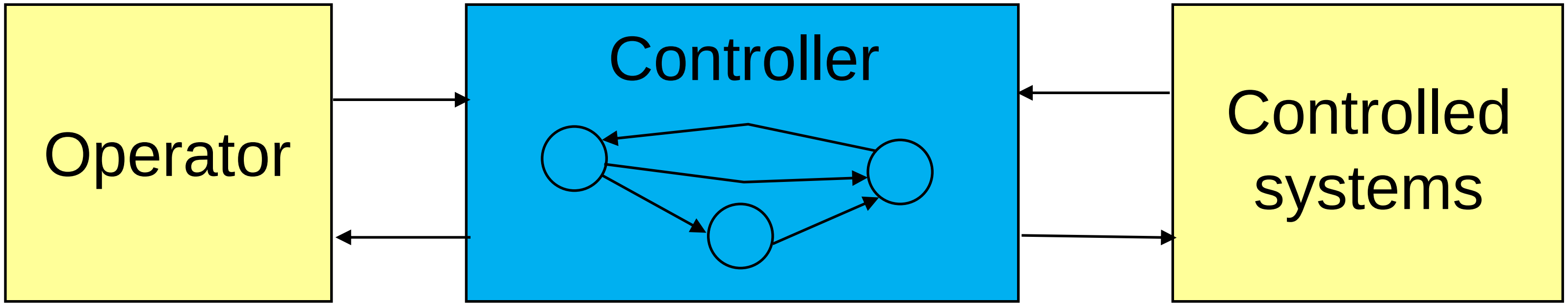
- For every input in every state there is a specified behavior
- Reactions are unambiguous (deterministic)
- Input is checked (value, timeliness)
- Handling of invalid inputs is specified
- Rate of interrupts is limited



Example: Review criteria for reactive systems

- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface

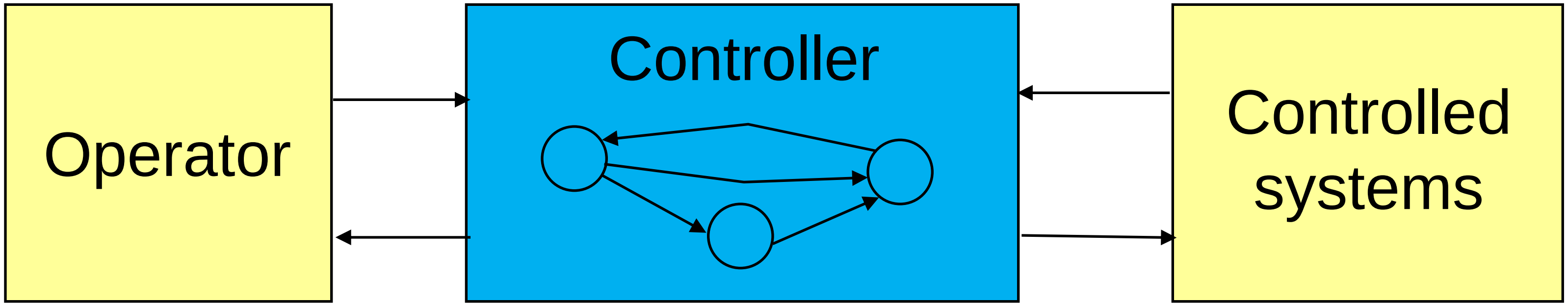
- Credibility checks are specified
- There is no unused output
- Processing capability of the environment is respected



Example: Review criteria for reactive systems

- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface

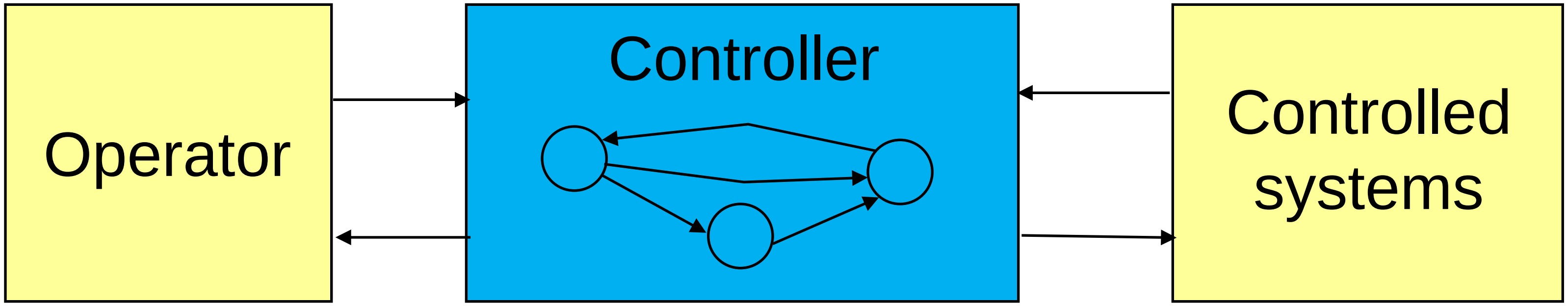
> - Effect of outputs is checked through the inputs
> - Control loop is stable



Example: Review criteria for reactive systems

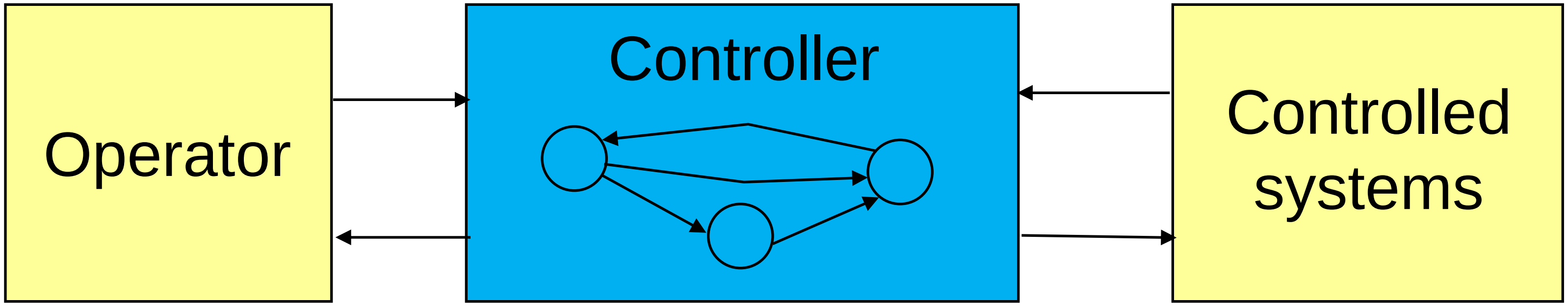
- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface

- Every state is reachable statically (incoming path)
- Transitions are reversible (there is a way back)
- More than one transitions from dangerous to safe states
- Confirmed transitions from safe to dangerous states



Example: Review criteria for reactive systems

- State definition
- Inputs (events)
- Outputs
- Outputs and triggers
- Transitions
- Human-machine interface



Managing requirements

Traceability

Basic tasks and tool support

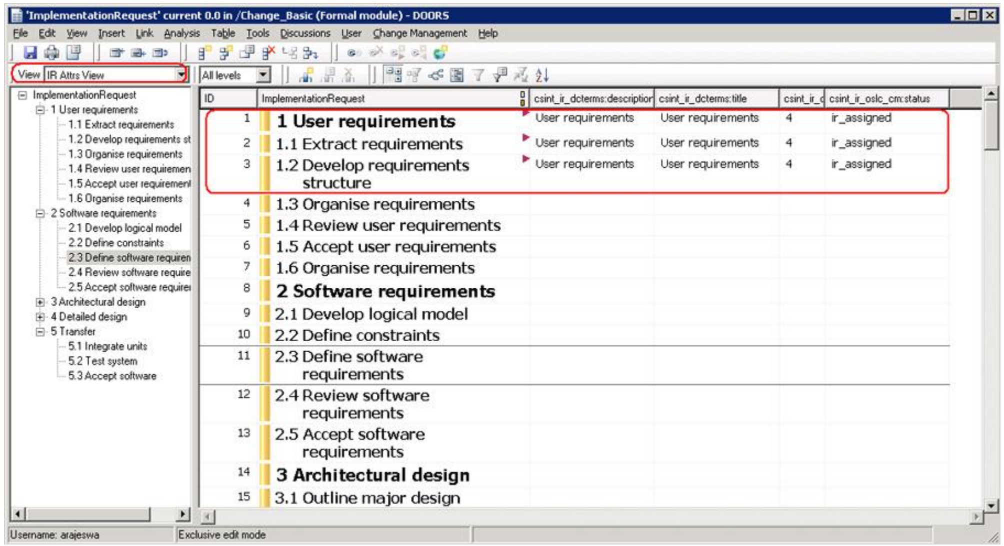
The role of traceability

- Traceability of requirements: Managing links among requirements and design artifacts
  - Among various levels of requirements: User –> System –> Module
  - Among requirements and design artifacts:
    Req. specification –> Architecture design –> Module design –> Source code –> Test –> Test result
- Analysis possibilities based on traceability links
  - Impact analysis: handling the changes
    - What is affected by a changed requirement?
  - Derivation analysis: handling utility and rewards
    - Why is this artifact here? What is the related requirement?
  - Coverage analysis: handling the status of development
    - What requirements are refined / implemented / tested?

Typical tasks of requirement management tools

Storing the requirements:	Hierarchic grouping
Handling the lifecycle and changes of requirements:	Using versions, attributes, timestamps, showing timeline of changes
Storing the relations: Support traceability:	Several types: Composition, derivation, refinement, implementation, .. Requirements – Design (models) – Source code – Test – Test results
Navigation on relations:	Forward: e.g., impact analysis Backward: e.g., derivation analysis
Generation of coverage lists:	Identify uncovered requirements or extra functionality
Handling authorization:	Defining roles and allowed activities
Sending notifications:	Messages in case of changes
Assuring integrity:	Detecting unintentional changes

Requirement management tools



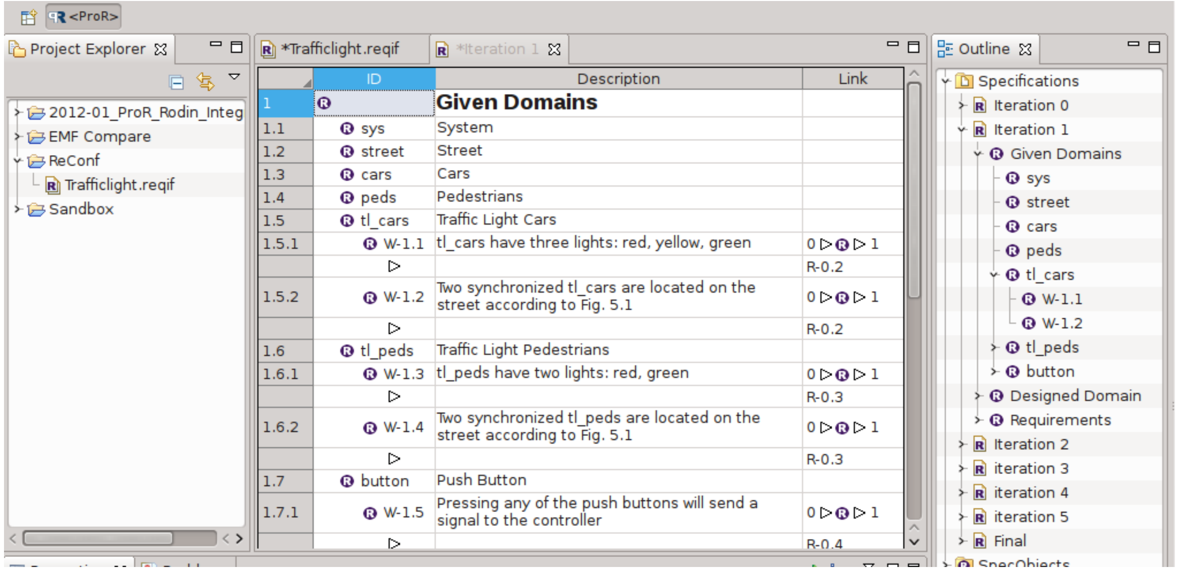
IBM Rational DOORS Next Generation

https://www.youtube.com/watch?v=qYK7_g4Fy44

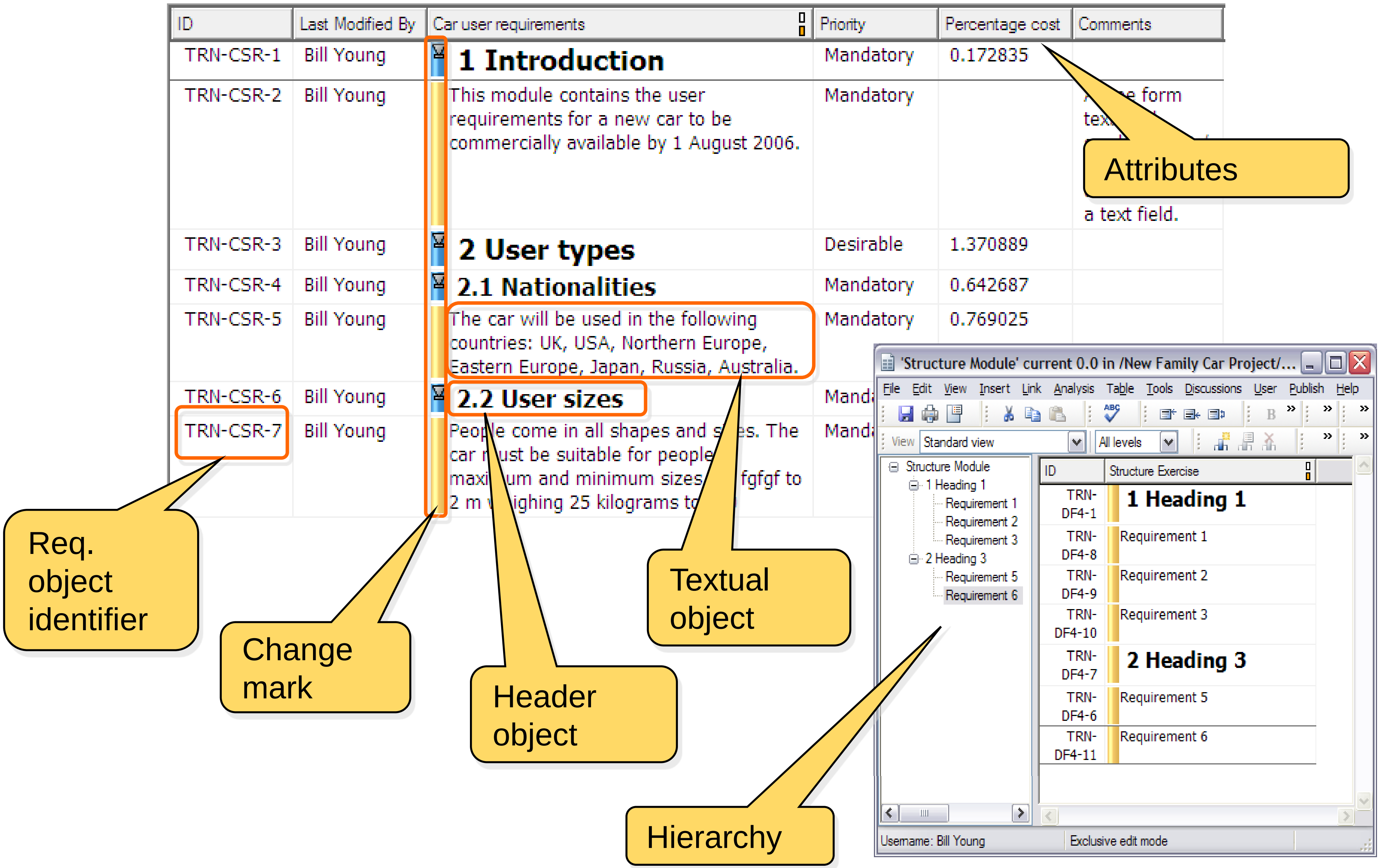
ProR

ReqIF Requirements Interchange Format

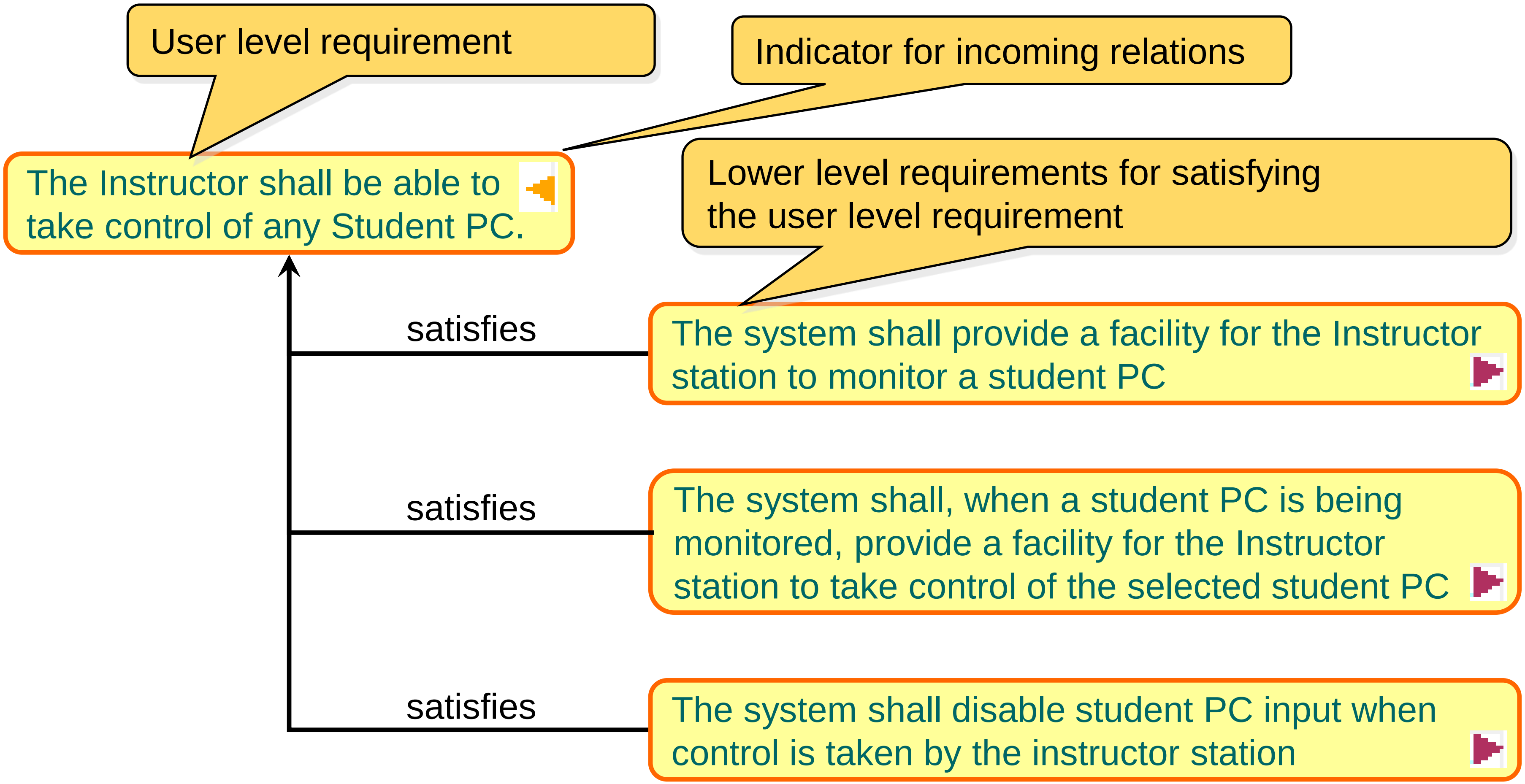
https://www.youtube.com/watch?v=YC_NrseqWcc



Example: IBM Rational DOORS



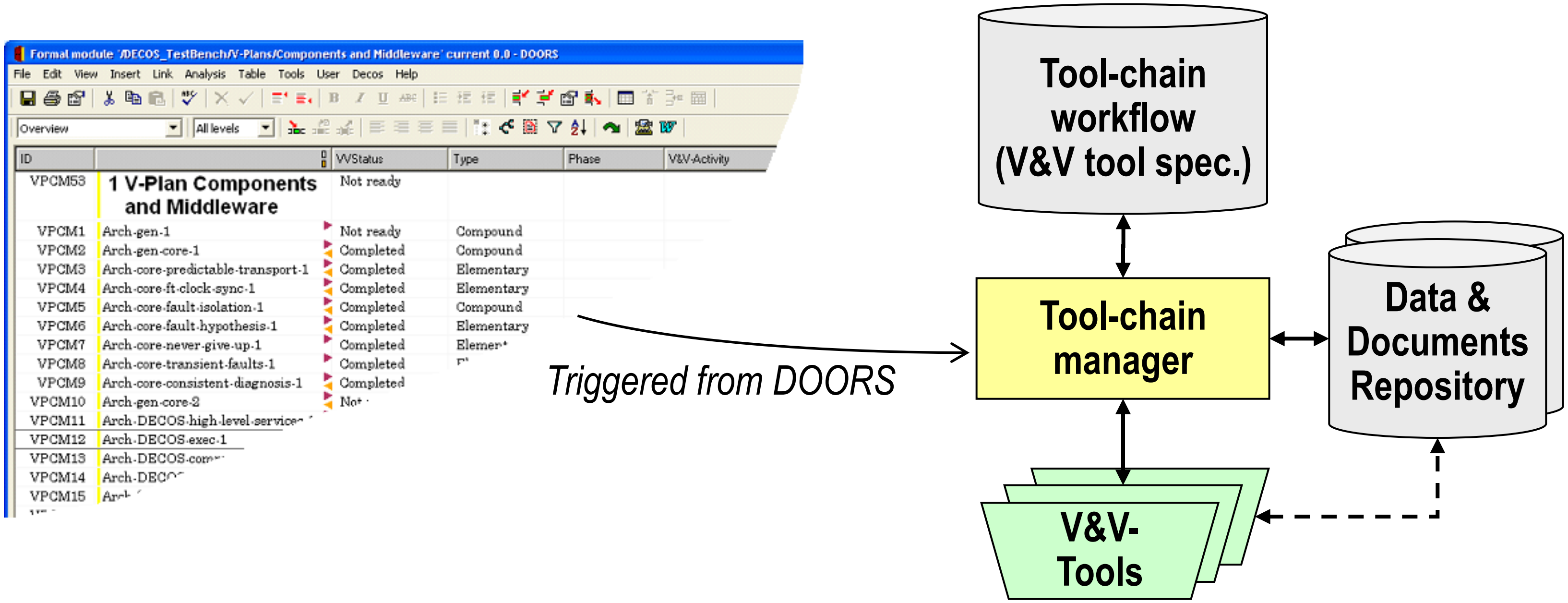
Example: IBM Rational DOORS



Requirement based verification tool-chains

- Assigning verification activities to requirements
  - Checking satisfaction of the req., collecting evidences
  - Standard-based techniques and measures (e.g., for safety case)
- Verification tool-chains (typically external)
  - Analysis: Generating analysis model, performing analysis, post-processing or visualization of results
  - Testing: (Model based) test case generation, test execution, providing test verdict
  - Measuring: Configuring measurements, executing measurements, data analysis
- Verification tool-chains can be started from the requirement management tool
  - Scripts with triggers (verifiable requirement)
- Registering the status of verification
  - Successfully verified requirement + repository of evidences

Example: Starting verification tool chain from DOORS



Example tools:

- ITEM (Hazard and risk analysis)
- RACER (Formal verification)
- SCADE MTC (Simulation)
- LDRA (Testing)
- PROPANE (Fault injection)
- EMI Test Bench

Summary

- Inputs and outputs of the phase
- Preparing the requirements specification
  - Formal languages
  - Semi-formal and structured methods
- Verification tasks
  - General aspects and verification techniques
  - Verifying completeness and consistency
- Managing requirements
  - Traceability
  - Basic tasks and tool support