Requirements Verification and Validation

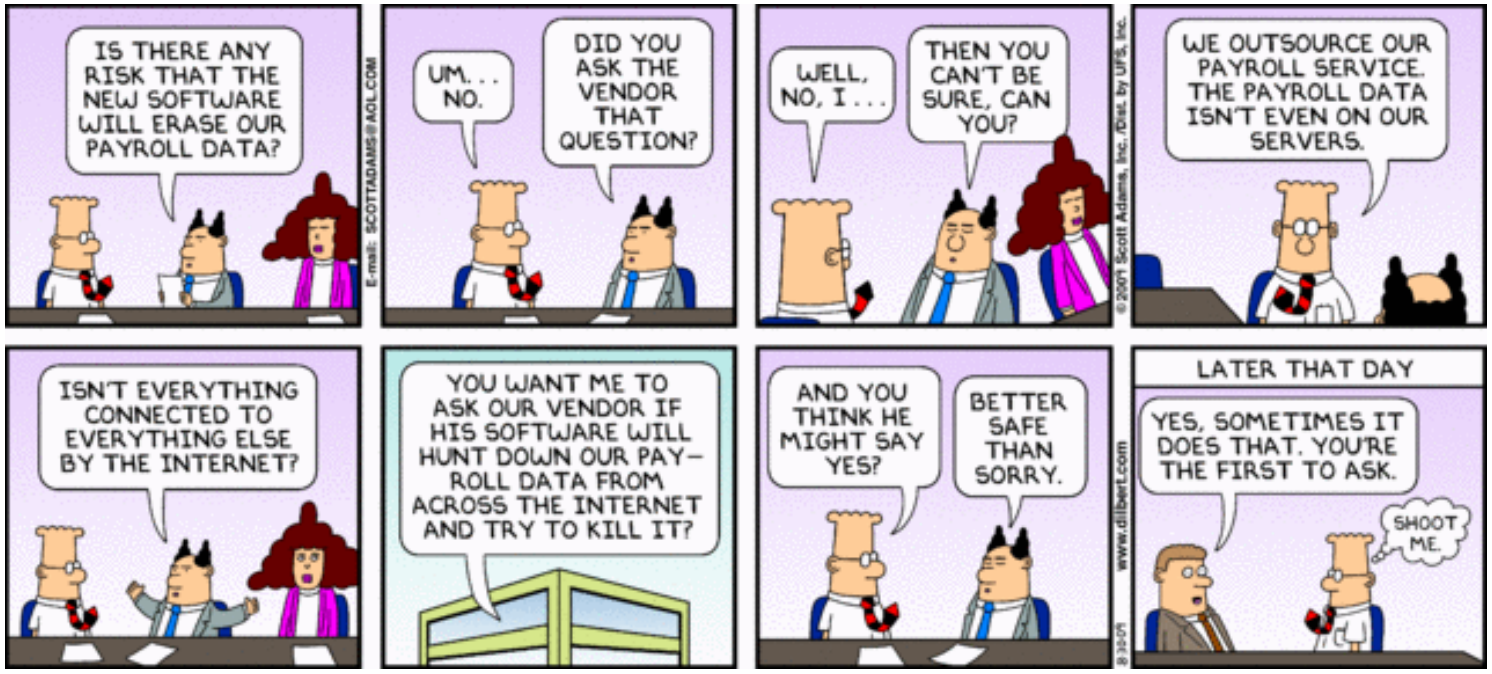
Gregor v. Bochmann, University of Ottawa

Based on Powerpoint slides by Gunter Mussbacher
with material from:
G. Kotonya and I. Sommerville, M. Jackson, P. Heymans,
S. Somé 2008, and D. Amyot 2008

Table of Contents

- The software is done. We are just trying to get it to work…[1]

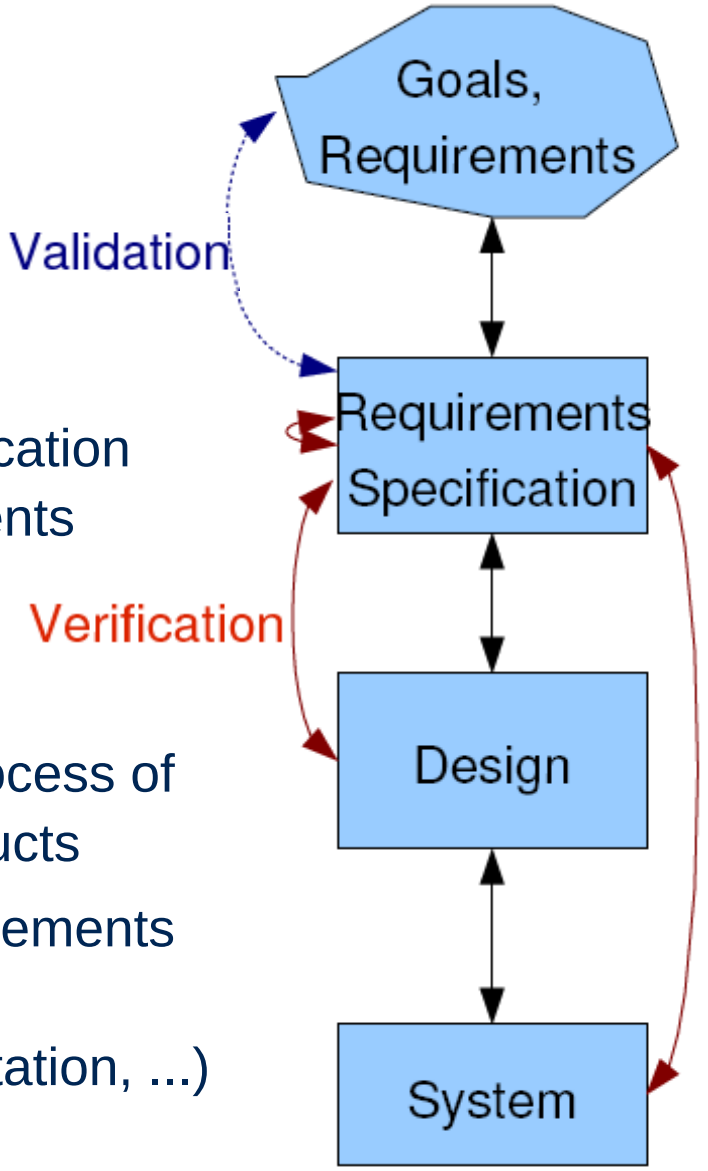
[1] Anonymous


IS THERE ANY RISK THAT THE NEW SOFTWARE WILL ERASE OUR PAYROLL DATA?
UM... NO.
DID YOU ASK THE VENDOR THAT QUESTION?
WELL, NO, I...
THEN YOU CAN'T BE SURE, CAN YOU?
WE OUTSOURCE OUR PAYROLL SERVICE. THE PAYROLL DATA ISN'T EVEN ON OUR SERVERS.
ISN'T EVERYTHING CONNECTED TO EVERYTHING ELSE BY THE INTERNET?
YOU WANT ME TO ASK OUR VENDOR IF HIS SOFTWARE WILL HUNT DOWN OUR PAY-ROLL DATA FROM ACROSS THE INTERNET AND TRY TO KILL IT?
AND YOU THINK HE MIGHT SAY YES?
BETTER SAFE THAN SORRY.
LATER THAT DAY
YES, SOMETIMES IT DOES THAT. YOU'RE THE FIRST TO ASK.
SHOOT ME.
E-mail: SCOTTADAMS@AOL.COM
© 2001 Scott Adams, Inc. /Dist. by UFS, Inc.
www.dilbert.com


I'LL DESIGN THE SYSTEM AS SOON AS YOU GIVE ME THE USER REQUIREMENTS.
BETTER YET, YOU COULD BUILD THE SYSTEM, THEN I'LL TELL YOUR BOSS THAT IT DOESN'T MEET MY NEEDS.
I DON'T MEAN TO FRIGHTEN YOU, BUT YOU'LL HAVE TO DO SOME ACTUAL WORK.
THAT'S CRAZY TALK.
www.dilbert.com scottadams@aol.com
© 2003 United Feature Syndicate, Inc.

Requirements Verification and Validation

- Requirements Validation
  - Check that the right product is being built
  - Ensures that the software being developed (or changed) will satisfy its stakeholders
  - Checks the software requirements specification against stakeholders goals and requirements
- Requirements Verification
  - Check that product is being built right
  - Ensures that each step followed in the process of building the software yields the right products
  - Checks consistency of the software requirements specification artefacts and other software development products (design, implementation, ...) against the specification



Requirements Verification and Validation (2)

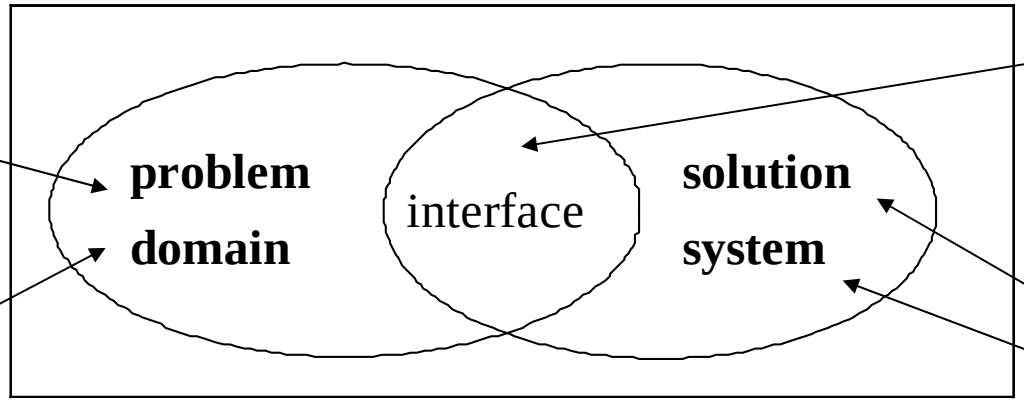
- Help ensure delivery of what the client wants
- Need to be performed at every stage during the (requirements) process
  - Elicitation
    - Checking back with the elicitation sources
    - “So, are you saying that . . . . . ?”
  - Analysis
    - Checking that the domain description and requirements are correct
  - Specification
    - Checking that the defined system requirement will meet the user requirements under the assumptions of the domain/environment
    - Checking conformity to well-formedness rules, standards…

The World and the Machine[1]

(or the problem domain and the system)

These 6 slides are taken from Introduction to Analysis

- Domain properties (D)
  these are assumptions about the environment of the system-to-be
- Requirements (R)



Specification (S)

Hardware (C)
Software (P)

- **Validation question** (do we build the right system?) : if the domain-to-be (excluding the system-to-be) has the properties D, and the system-to-be has the properties S, then the requirements R will be satisfied.

$$D \text{ and } S \Rightarrow R$$

- **Verification question** (do we build the system right?) : if the hardware has the properties H, and the software has the properties P, then the system requirements S will be satisfied.

$$C \text{ and } P \Rightarrow S$$

- Conclusion:

$$D \text{ and } C \text{ and } P \Rightarrow R$$

[1] M. Jackson, 1995

Example

- Requirement
  - (R) Reverse thrust shall only be enabled when the aircraft is moving on runway.
- Domain Properties
  - (D1) Deploying reverse thrust in mid-flight has catastrophic effects.
  - (D2) Wheel pulses are on if and only if wheels are turning.
  - (D3) Wheels are turning if and only if the plane is moving on the runway.
- System specification
  - (S) The system shall allow reverse thrust to be enabled if and only if wheel pulses are on.
- Does D1 and D2 and D3 and S $\Rightarrow$ R?
  - Are the domain assumptions (D) right? Are the requirement (R) or specification (S) what is really needed?

The assumption D3 is false because the plane may hydroplane on wet runway.

based on P. Heymans, 2005

Requirement specifications including assumptions

- Often the requirements for a system-to-be include assumptions about the environment of the system.
- The system specification S, then, has the form:

  $S = A \Rightarrow G$

  where A are the assumptions about the environment and G are the guarantees that the system will provide as long as A hold.
- If these assumptions (A) are implied by the known properties of the domain (D), that is $D \Rightarrow A$, and we can check that the domain properties (D) and the system guarantees (G) imply the requirements (R), that is D and $G \Rightarrow R$, then the "validation condition" D and $S \Rightarrow R$ is satisfied.

Specification with assumptions and guarantees (example)

Example: A power utility provides electricity to a client. The problem is that the monthly invoice is not related to the electricity consumption, because there is no information about this consumption.

- Idea of a solution: introduce an electricity counter.
- **Specification of the electricity counter**
  - **Inputs and outputs**
    - input power from utility (voltage, current) – voltage supplied by utility
    - output power to client (voltage, current) – current used by client
    - Reset button (input)
    - consumption (output - watt-hours of electricity consumption)

Example (suite)

- **Assumptions**
  - Input voltage < 500 Volts (determined by utility)
  - Output current < 20 Amps (determined by client)
- **Guarantees**
  - Output voltage = input voltage
  - Input current = output current
  - Consumption output shall indicate the consumption since the last reset operation, that is, the integral of (output voltage x output current) over the time period from the occurrence of the last reset operation to the current time instant.

- **Software example**

- Specification of a method providing the interface “List search(Criteria c.
  Assumption: c is a data structure satisfying the Criteria class properties.
  Guarantee: the returned result is a list satisfying the List class properties and includes all items from the database that satisfy c.

Formal Verification and Validation

- Evaluating the satisfaction of "D and S $\Rightarrow$ R" is difficult with natural language
  - Descriptions are verbose, informal, ambiguous, incomplete...
  - This represents a risk for the development and organization
- Verification of this "validation question" is more effective with formal methods (see below)
  - Based on mathematically formal syntax and semantics
  - Proving can be tool-supported
- Depending on the modeling formalism used, different verification methods and tools may be applied. We call this "Model-Based V&V"
  - In the case of the aircraft example above, we used Logic to write down statements about the model. This is a particular case of modeling formalism.

V&V vs. Analysis

- Both have several activities in common
  - Reading requirements, problem analysis, meetings and discussions...
- Analysis works with raw, incomplete requirements as elicited from the system stakeholders
  - Develop a software requirements specification document
  - Emphasis on "we have the right requirements"
- Requirements V&V works with a software requirements specification and with negotiated and agreed (and presumably complete) domain requirements
  - Check that this these specifications are accurate
  - Emphasis on "we have the right requirements well done"

Requirements V&V Techniques

Various Requirements V&V Techniques

- Simple checks
  - Traceability, well-written requirements
- Prototyping
- Functional test design
- User manual development
- Reviews and inspections
  - Walkthroughs
  - Formal inspections
  - Checklists
- Model-Based V&V
  - First-order logic
  - Behavioral models

Simple Checks

- Various checks can be done using traceability techniques
  - Given the requirements document, verify that all elicitation notes are covered
  - Tracing between different levels of requirements
    - Checking goals against tasks, features, requirements…
- Involves developing a traceability matrix
  - Ensures that requirements have been taken into consideration (if not there should be a reason)
  - Ensures that everything in the specification is justified
- Verify that the requirements are well written (according to the criteria already discussed)

Prototyping (1)

- Excellent for validation by users and customers
  - More accessible than specification
  - Demonstrate the requirements and help stakeholders discover problems
- Come in all different shapes and sizes
  - From paper prototype of a computerized system to formal executable models/specifications
  - Horizontal, vertical
  - Evolutive, throwaway

Prototyping (2)

- Important to choose scenarios or use cases for elicitation session

- Prototyping-based validation steps
  - Choose prototype testers
  - Develop test scenarios
    - Careful planning is required to draw up a set of test scenarios which provide broad coverage of the requirements
    - Users should not just play around with the system as this may never exercise critical system features
  - Execute test scenarios
  - Document problems using a problem reporting tool

Comment on next two techniques

- The two V&V techniques, namely Functional Test Design and User Manual Development, are not really V&V techniques.
- They are activities that must be performed anyway, and they are based on the specification document.
  - Through these activities, as for any other activities based on the specification document, errors and other problems with this document may be detected.

Functional Test Design

- Functional tests at the system level must be developed sooner or later...
  - Can (and should) be derived from the requirements specification
  - Each (functional) requirement should have an associated test
  - Non-functional (e.g., reliability) or exclusive (e.g., define what should not happen) requirements are harder to validate with testing
  - Each requirements test case must be traced to its requirements
  - Inventing requirements tests is an effective validation technique
- Designing these tests may reveal errors in the specification (even before designing and building the system)!
  - Missing or ambiguous information in the requirements description may make it difficult to formulate tests
- Some software development processes (e.g., agile methods) begin with tests before programming ➔ Test-Driven Development (TDD)

User Manual Development

- Same reasoning as for functional test design
  - Has to be done at some point
  - Reveals problems earlier
- Forces a detailed look at requirements
- Particularly useful if the application is rich in user interfaces / for usability requirements

- Typical information in a user manual
  - Description of the functionality
  - How to get out of trouble
  - How to install and get started with the system

Reviews and Inspections (1)

- A group of people read and analyze requirements, look for potential problems, meet to discuss the problems, and agree on a list of action items needed to address these problems
- A widely used requirements validation technique
  - Lots of evidence of effectiveness of the technique
- Can be expensive
  - Careful planning and preparation
  - Pre-review checking
  - Need appropriate checklists (must be developed if necessary and maintained)

Reviews and Inspections (2)

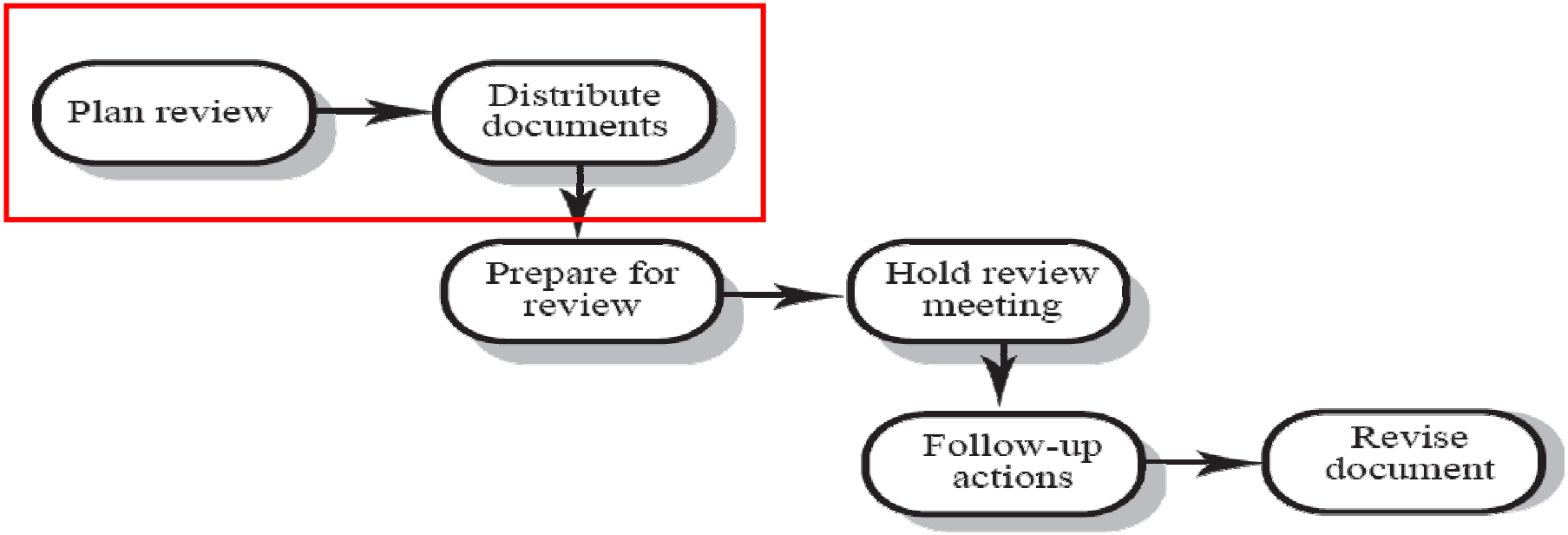
- Different types of reviews with varying degrees of formality exist (similar to JAD vs. brainstorming sessions)
  - Reading the document
    - A person other than the author of the document
  - Reading and approval (sign-off)
    - Encourages the reader to be more careful (and responsible)
  - Walkthroughs
    - Informal, often high-level overview
    - Can be led by author/expert to educate others on his/her work
  - Formal inspections
    - Very structured and detailed review, defined roles for participants, preparation is needed, exit conditions are defined
    - E.g., Fagan Inspection

Reviews and Inspections (3)

- Different types of reviews (cont'd)
  - Focused inspections
    - Reviewers have roles, each reviewer looks only for specific types of errors
  - Active reviews
    - Author asks reviewer questions which can only be answered with the help of the document to be reviewed

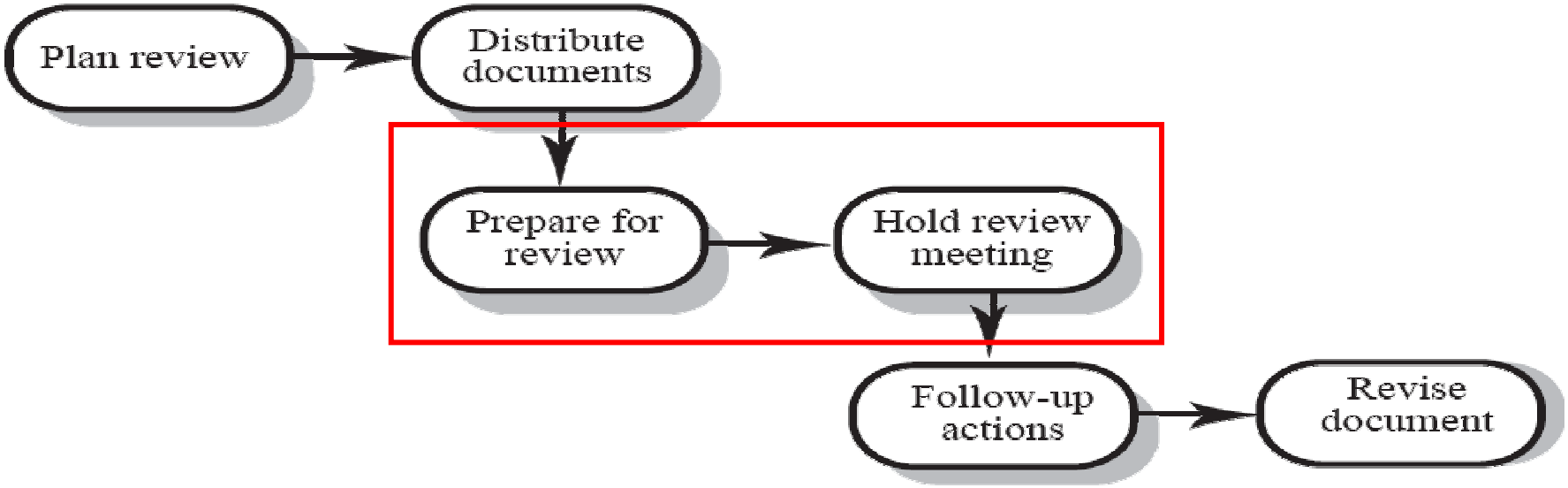
Typical Review / Inspection Steps (1)

- Plan review
  - The review team is selected and a time and place for the review meeting is chosen
- Distribute documents
  - The requirements document is distributed to the review team members



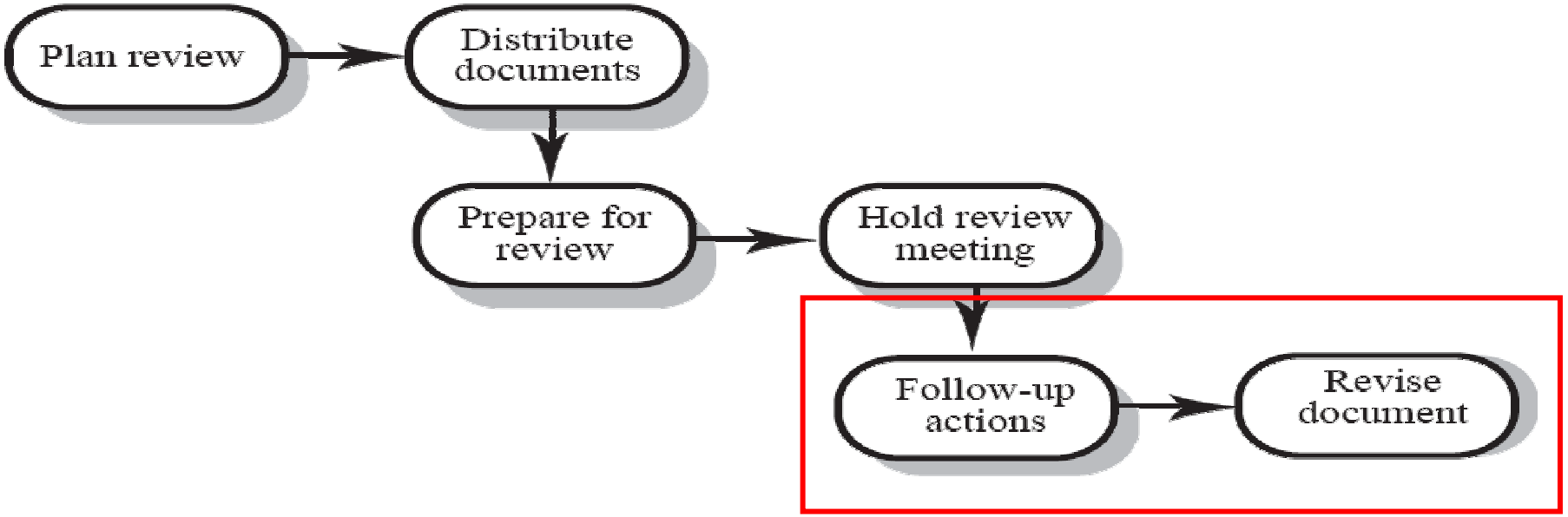
Typical Review / Inspection Steps (2)

- Prepare for review
  - Individual reviewers read the requirements to find conflicts, omissions, inconsistencies, deviations from standards, and other problems
- Hold review meeting
  - Individual comments and problems are discussed and a set of action items to address the problems is established



Typical Review / Inspection Steps (3)

- Follow-up actions
  - The chair of the review checks that the agreed action items have been carried out
- Revise document
  - Requirements document is revised to reflect the agreed action items
  - At this stage, it may be accepted or it may be re-reviewed



Review Team

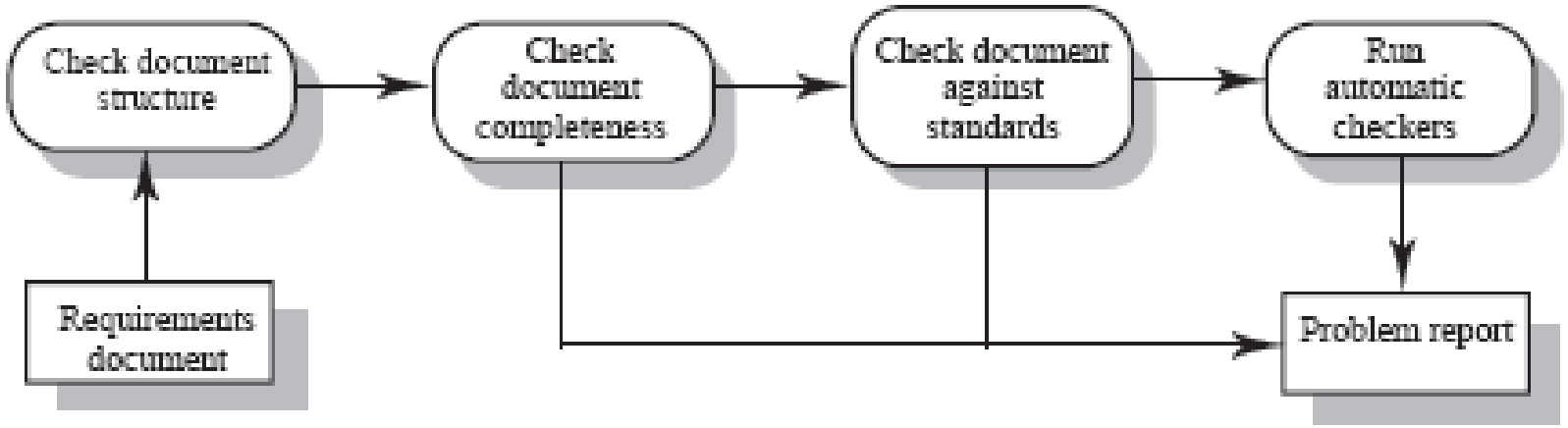
- Reviews should involve a number of stakeholders drawn from different backgrounds
  - People from different backgrounds bring different skills and knowledge to the review
  - Stakeholders feel involved in the RE process and develop an understanding of the needs of other stakeholders
  - Review team should always involve at least a domain expert and a user

Review – Problem Categorization

- Requirements clarification
  - The requirement may be badly expressed or may have accidentally omitted information which has been collected during requirements elicitation
- Missing information
  - Some information is missing from the requirements document
- Requirements conflict
  - There is a significant conflict between requirements
  - The stakeholders involved must negotiate to resolve the conflict
- Unrealistic requirement
  - The requirement does not appear to be implementable with the technology available or given other constraints on the system
  - Stakeholders must be consulted to decide how to make the requirement more realistic

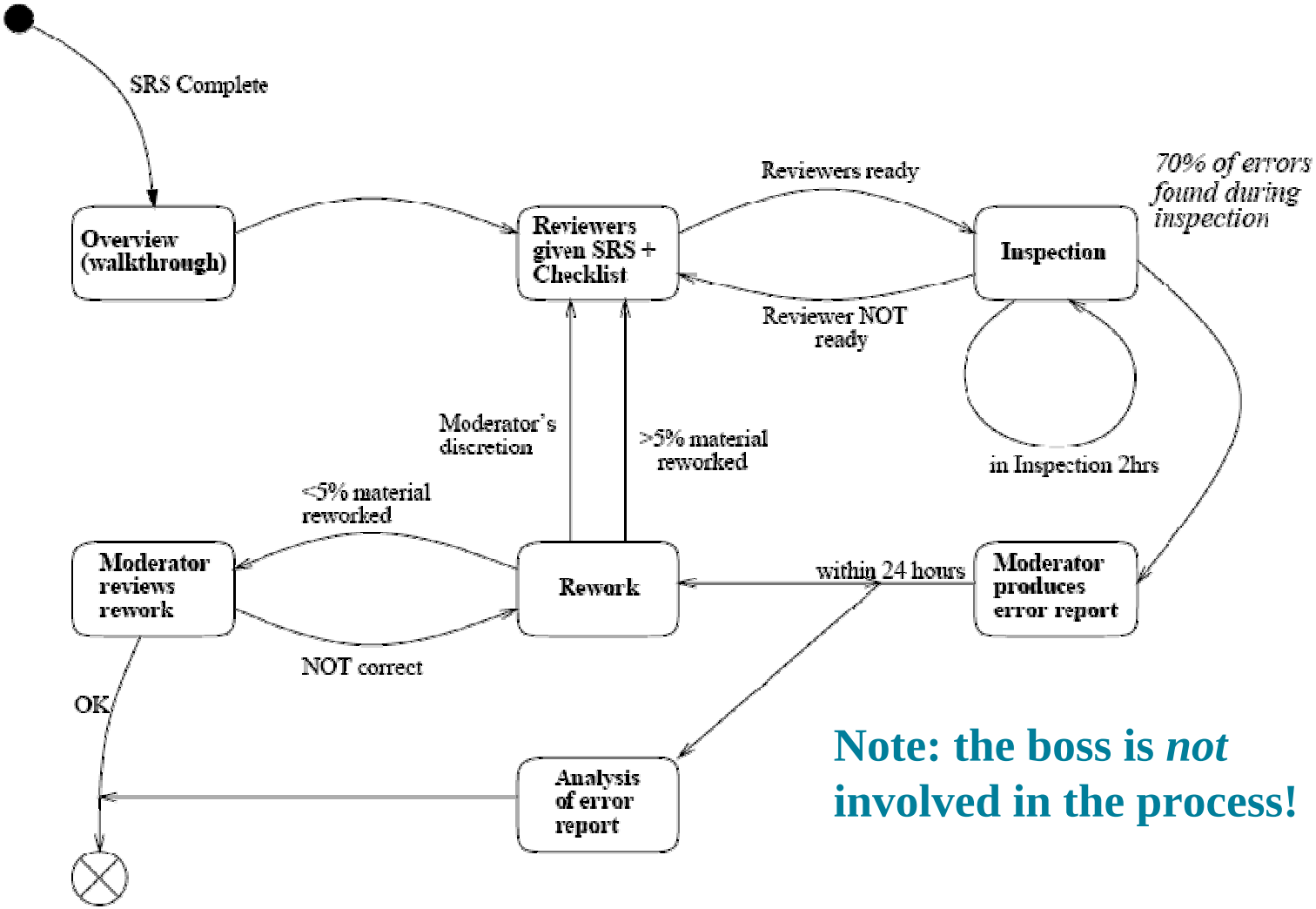
Pre-Review Checking

- Reviews can be expensive because they involve many people over several hours reading and checking the requirements document
- We can reduce this cost by asking someone to make a first pass called the pre-review
  - Check the document and look for straightforward problems such as missing requirements (sections), lack of conformance to standards, typographical errors, etc.



Fagan Inspection (1)

- Formal and structured inspection process



Note: the boss is *not* involved in the process!

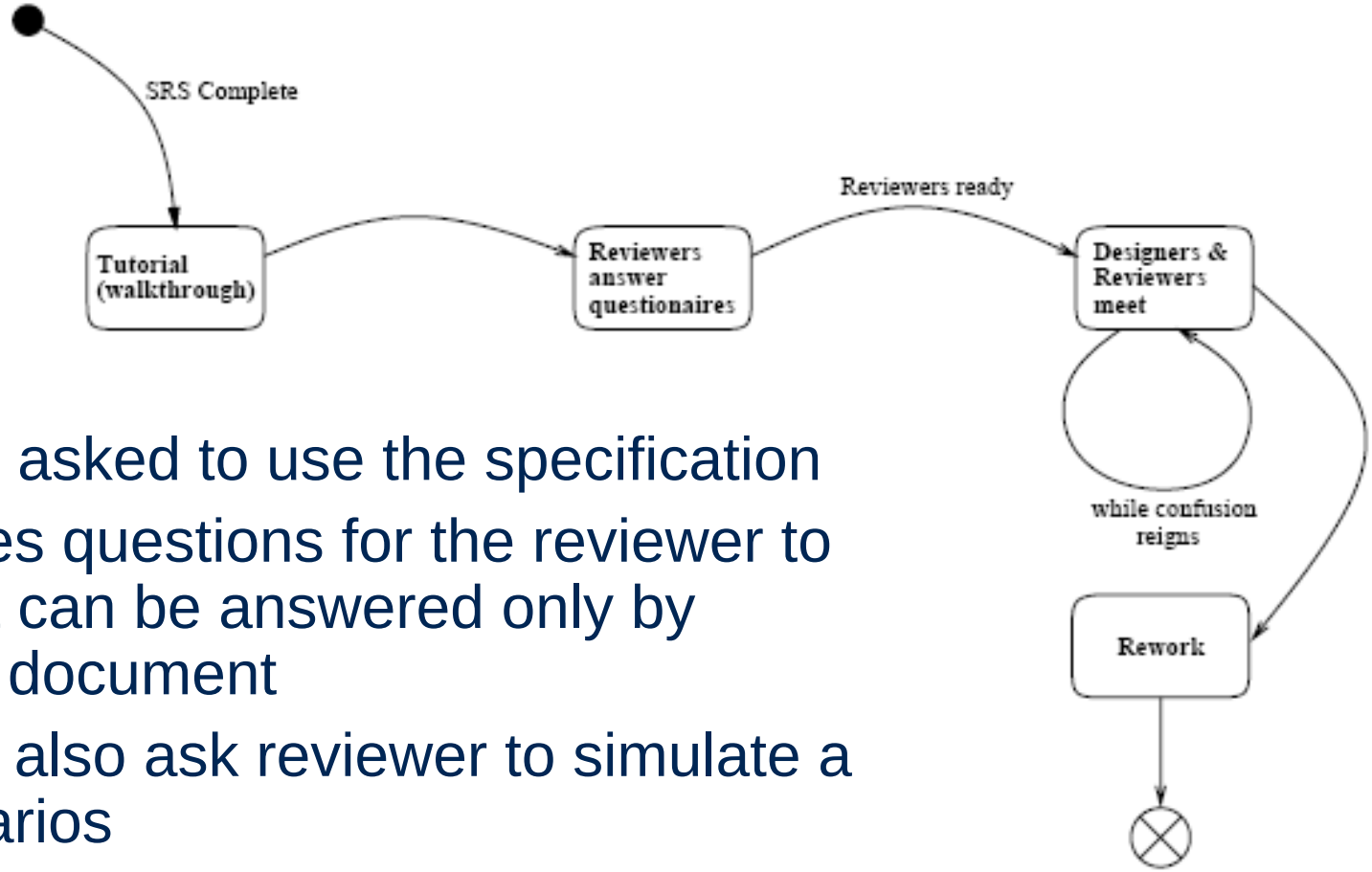
Fagan Inspection (2)

- Characterized by rules on who should participate, how many reviewers should participate, and what roles they should play
  - Not more than 2 hours at a time, to keep participants focused
  - 3 to 5 reviewers
  - Author serves as the presenter of the document
  - Metrics are collected
    - Important: the author's supervisor does not participate in the inspection and does not have access to data
    - This is not an employee evaluation
  - Moderator is responsible for initiating the inspection, leading the meeting, and ensuring issues found are fixed
  - All reviewers need to prepare themselves using checklists
  - Issues are recorded in special forms

Fagan Inspection (3)

- The inspection meeting is like a brainstorming session to identify (potential) problems
- Re-inspection if > 5% of the document change
  - Some variants are less tolerant... too easy to introduce new errors when correcting the previous ones!

Active Review



- Reviewer is asked to use the specification
- Author poses questions for the reviewer to answer that can be answered only by reading the document
- Author may also ask reviewer to simulate a set of scenarios

Requirements Review Checklists (1)

- Essential tool for an effective review process
  - List common problem areas and guide reviewers
  - May include questions an several quality aspects of the document: comprehensibility, redundancy, completeness, ambiguity, consistency, organization, standards compliance, traceability ...
- There are general checklists and checklists for particular modeling and specification languages
- Checklists are supposed to be developed and maintained

- See example on course website

Requirements Review Checklists (2)

- Sample of elements in a requirements review checklist
  - Comprehensibility – can readers of the document understand what the requirements mean?
  - Redundancy – is information unnecessarily repeated in the requirements document?
  - Completeness – does the checker know of any missing requirements or is there any information missing from individual requirement descriptions?
  - Ambiguity – are the requirements expressed using terms which are clearly defined? Could readers from different backgrounds make different interpretations of the requirements?
  - Consistency – do the descriptions of different requirements include contradictions? Are there contradictions between individual requirements and overall system requirements?

Requirements Review Checklists (3)

- Sample of elements (cont'd)
  - Organisation – is the document structured in a sensible way? Are the descriptions of requirements organised so that related requirements are grouped?
  - Conformance to standards – does the requirements document and individual requirements conform to defined standards? Are departures from the standards justified?
  - Traceability – are requirements unambiguously identified? Do they include links to related requirements and to the reasons why these requirements have been included?

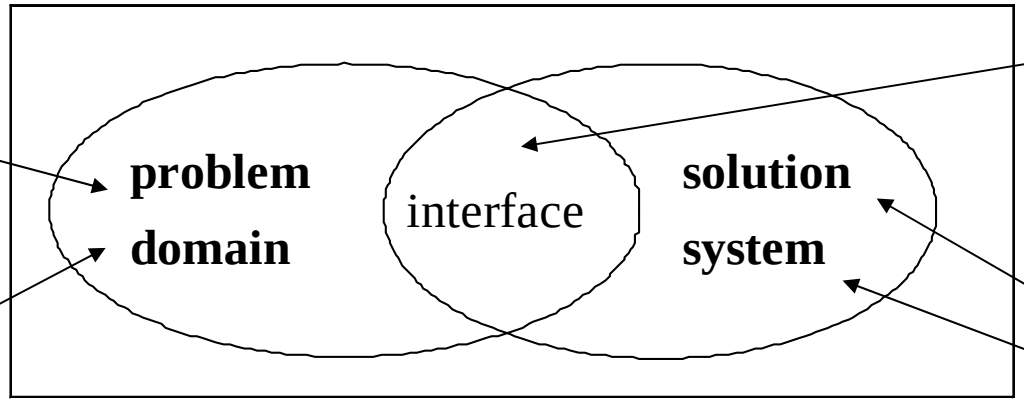
Comments on Reviews and Inspections

- Advantages
  - Effective (even after considering cost)
  - Allow finding sources of errors (not only symptoms)
  - Requirements authors are more attentive when they know their work will be closely reviewed
    - Encourage them to conform to standards
  - Familiarize large groups with the requirements (buy-in)
  - Diffusion of knowledge
- Risks
  - Reviews can be dull and draining (need to be limited in time)
  - Time consuming and expensive (but usually cheaper than the alternative)
  - Personality problems
  - Office politics…

Model-based (formal) Verification and Validation

The problem domain and the system – copied from Introduction to Analysis and Specification

- Domain properties (D)
  these are assumptions about the environment of the system-to-be
- Requirements (R)



Specification (S)

Hardware (C)
Software (P)

- **Validation question** (do we build the right system?) : if the domain-to-be (excluding the system-to-be) has the properties D, and the system-to-be has the properties S, then the requirements R will be satisfied.

$$D \text{ and } S \Rightarrow R$$

- **Verification question** (do we build the system right?) : if the hardware has the properties H, and the software has the properties P, then the system requirements S will be satisfied.

$$C \text{ and } P \Rightarrow S$$

- Conclusion:

$$D \text{ and } C \text{ and } P \Rightarrow R$$

[1] M. Jackson, 1995

Modeling paradigms

- Modeling paradigms
  - Entity-Relationship modeling – e.g. UML Class diagrams
  - Workflow modeling notations – there are many different “dialects”, such as UML Activity diagrams, UCM, BPML, Petri nets (a very simple formal model), Colored Petri nets
  - State machines – e.g. Finite State Machines (FSM – a very simple formal model), extended FSMs, such as UML State diagrams
  - First-order logic – notations such as Z, VDM, UML-OCL, etc.
    - Can be used as an add-on with the other paradigms above, by providing information about data objects and relationships (possibly in the form of “assertions” or “invariants” that hold at certain points during the dynamic execution of the model)
    - Can be used alone, expressing structural models and behavioral models (there are many examples of using Z for such purpose)

Formal V&V techniques and tools (i)

- Available V&V techniques will vary from one modeling paradigms to another and will also depend on the available tools (that usually only apply to a particular "dialect" of the modeling paradigm)
- The following functions may be provided through tools
  - Completeness checking – only according to certain syntax rules, templates
  - Consistency checking : given model M, show that M does not imply a contradiction and does not have any other undesirable general property (e.g. deadlock possibility)
  - Refinement checking : given two models M and M', show that the properties of M imply the properties of M'. This can be used for the validation of the system specification S, that is, showing that D and S $\Rightarrow$ R where D are the domain properties and R are the domain requirements (M = D and S; M' = R)
  - Model checking : given a model M and some properties P, show that any system implementation satisfying M will have the properties P
  - Generation of system designs or prototype implementations (from workflow or state machine models)
  - Generation of test cases
  - Performance evaluation

Formal V&V techniques and tools (ii)

- Consistency and Refinement checking
  - Logic models
    - Theorem proving
  - Workflow and State machine models
    - Simulated execution (prototype implementations)
    - Reachability analysis (determining all reachable states of a system consisting of a composition of several state machines, or of a workflow model). In contrast, simulated execution will only perform partial analysis – namely a certain number of test cases (note: one may consider a very large number of such cases, possibly randomly generated).
    - *These techniques have first be developed for distributed systems (communication protocols), see **Some notes on the history of protocol engineering** (G. v. Bochmann, D. Rayner and C. H. West) to be published in Computer Networks journal.* - http://www.site.uottawa.ca/~bochmann/dsrg/PublicDocuments/Publications/Boch10a-submitted.pdf

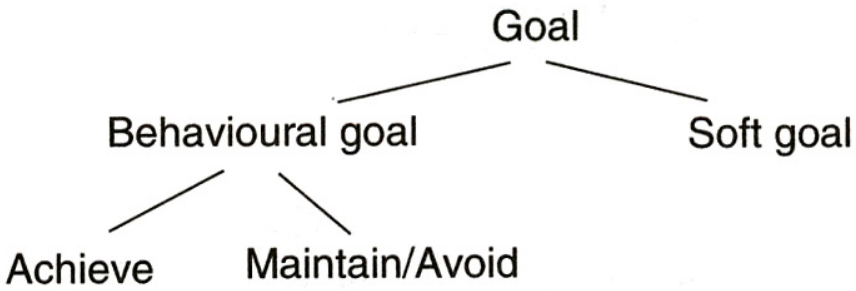
Consistency checking for state machines

- Different types of refinements
  - Refinement (also called Conformance) between two machines (for example, one abstract and the other one more concrete)
  - Reduction of non-determinism
  - Reduction of optional behavior (compliant, but some behaviors are not supported)
  - Extension (conformance, but some new events are treated and lead to new behaviors)
- Equivalence checking
  - Between two machines (for example, one abstract and the other one more concrete)
  - Several types of equivalence: trace equivalence (same traces of events can be observed), refusal equivalence (same blocking behavior), observational equivalence (equivalent states in both machines), etc.

Formal V&V techniques and tools (iii)

- Model checking: Is normally used for behavioral workflow and state machine models (however, the Alloy tool can also be used for checking structural Class diagram models).
  - Uses the approach of reachability analysis
  - The typical properties to be verified for a given model could be the following (note: can also be checked by simulated execution):
    - General properties (to be satisfied by most systems):
      - Absence of deadlocks in a system with concurrency
      - No non-specified messages, that is, for all events that may occur their handling is defined
      - All states can be reached and all transitions can be traversed
    - Specific properties (depending on this particular system): Such specific properties must be specified in some suitable notation, such as
      - Logic assertions or invariants
      - Temporal logic (extension of predicate calculus with two operators: always and eventually (corresponding to Maintain/Avoid goals and Achieve goals, respectively)

Different types of goals – copied from Goal-oriented modeling



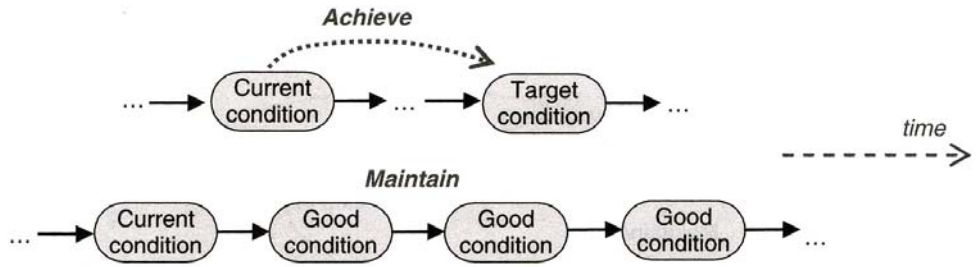


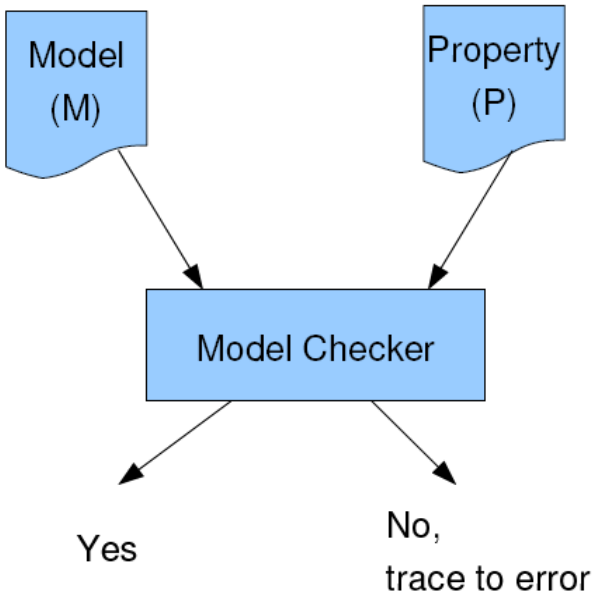
Figure 7.4 *Behavioural goals: Achieve and Maintain goals*

Figure 7.2 *A taxonomy of goal types*

- Behavioral goal: establishment of goal can be checked
  - Describes intended behavior declaratively
  - Implicitly defines a maximal set of admissible behaviors
    - Achieve: points to future (like "eventually" operator in Temporal Logic)
    - Maintain/Avoid: states property that always holds (like "always" operator)
- Soft-Goal: are more or less fulfilled by different alternatives of (external) design – often difficult to quantify – one says, some alternative may "satisfice" the goal

Model checking

- Verifies that the model satisfies temporal logic properties, for example:
  - If A occurs, B will occur in the future (eventually)
  - If C occurs, D will be true always in the future
- Traverse systematically all possible behaviors (execution paths) of the machine (reachability analysis)
  - Verification of properties done after reachability analysis or on the fly
- Model checker verifies M ⇒ P (if no trace of states and transitions leading to the violation of P is found) – otherwise a counter example trace is provided
- Major obstacle is state space explosion



Example tools:

SPIN (see http://spinroot.com/spin/whatispin.html) - for distributed systems with message passing

Alloy (see http://alloy.mit.edu/community/) – for OO Class diagrams with assertions

Performance modeling and evaluation

Performance Analysis

Recall URN Example I

- Which of the three wireless IN alternative architectures is the best for this scenario?
  - Service and Data in MsgSwitchingCenter
  - Service in MsgSwitchingCenter, Data in ServiceNode
  - Service and Data in ServiceControlPoint

- Different approaches to performance analysis
  - Informal: Qualitative analysis with GRL strategies
  - Counting the number of messages involved: e.g. transformations of workflow scenarios into sequence diagrams
  - Model-based performance evaluation
    - Queuing models : consider resources, service times and request queuing
    - Markov models : consider transition probabilities of state machine models

Performance modeling : Markov models

- Markov models
  - State machine model where each transition has a given rate of occurrence; this leads to an exponential distribution of the sejourn time in a given state.
  - This modeling paradigm is often used for modeling reliability, availability etc.
  - Example: Machine may be operational or failed. In the operational state, the rate of the failing transition is 0.001 per hour, in the failed state, the rate of the repaired transition (back to the operational state) is 1.0 per hour (the machine remains in the failed state a duration that has an exponential distribution with average 1 hour).

Performance modeling : Queuing models

- Queuing models
  - One considers: user requests, resources (servers), service times (for processing requests by resources) and request queuing
  - One talks about queueing networks – a kind of workflow model involving several resources providing various services and requests that flow between resources (closed system: users are also modeled as resources – open system: users are outside the “system”)
  - The performance of workflow models (UML Activity diagrams or UCMs) can be naturally modeled by queueing networks.
    - The jUCMNav provides for the automatic transformation into such a model using an intermediate representation called Core Senario Model (CSM)
  - The functional workflow model must be complemented with performance parameters in order to provide the necessary input data for performance modeling. This includes:
    - Performance data on resources: e.g. service times, queuing disciplines, etc.
    - Performance data on work load: e.g. number of requests per unit time, etc.

Performance evaluation tools

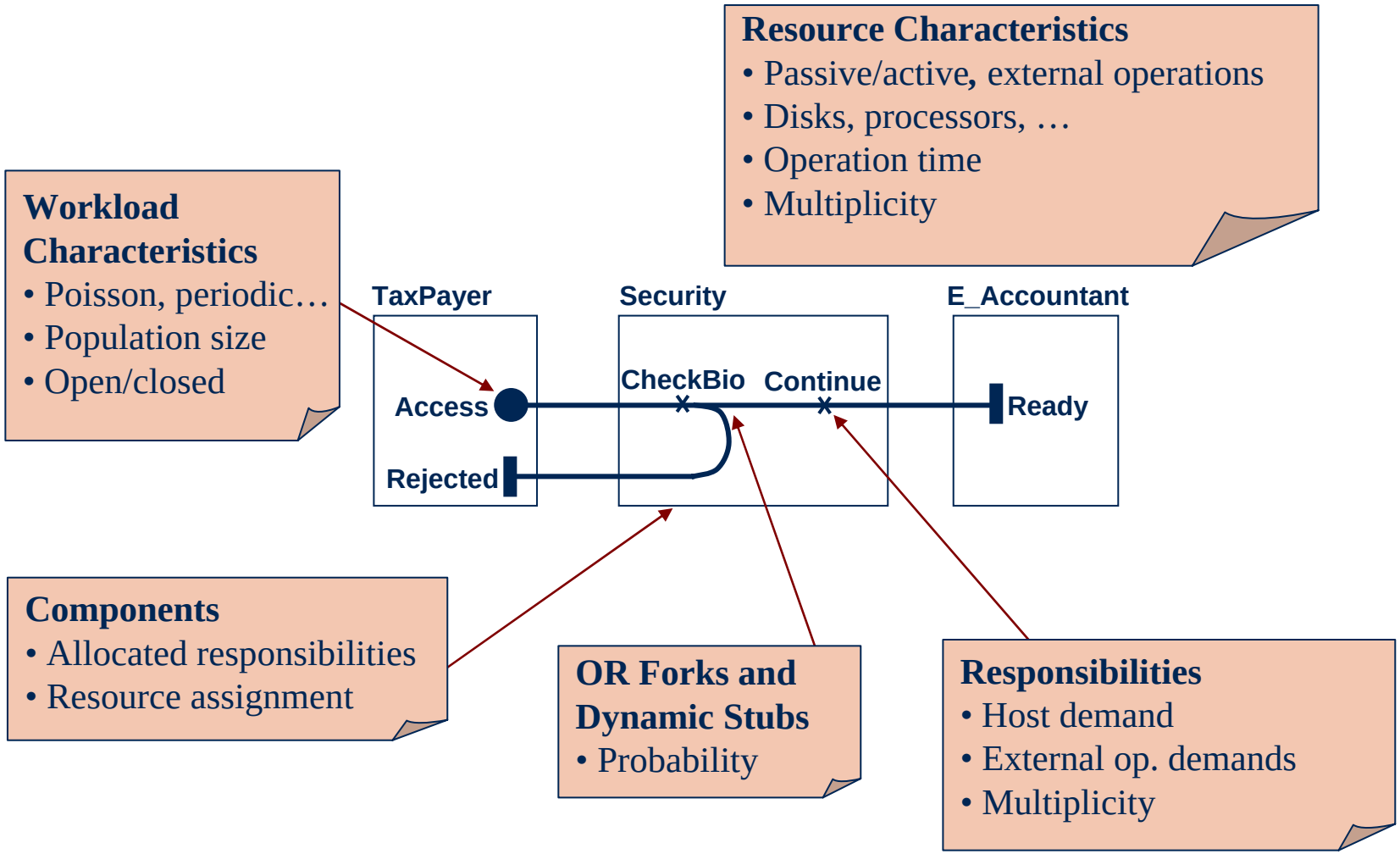
- For both, Markov and Queuing models, there are two basic approaches to performance evaluation:
  - Analytical formulas
  - Simulation studies

- Special versions of modeling paradigms
  - Layered Queuing Networks (LQN - using several layers of abstraction, like layered operating system functions) – developed by Dr. Woodside at Carleton University
  - Stochastic Petri nets (Markov's rate-based transitions applied to Petri nets)

Typical Performance Results from Queuing models

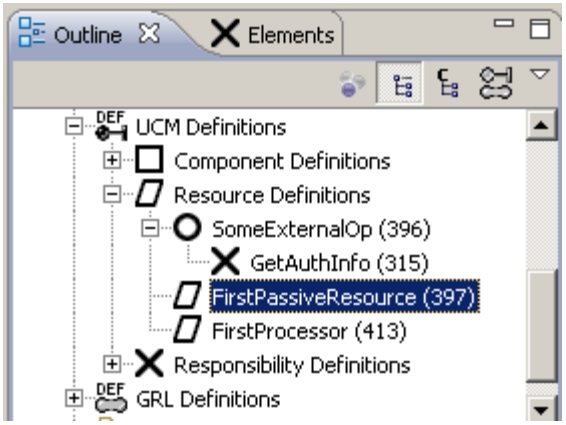
- General statistics
  - Elapsed time, system time…
- Measured quantities
  - Service demands, number of blocking and non-blocking calls, call delays, synchronization delays
- Service times
  - For every entry and activity, with confidence intervals and variances (where relevant)
- Throughputs and utilizations for every entry and activity, with confidence intervals
- Utilizations and waiting times for devices (by entry)

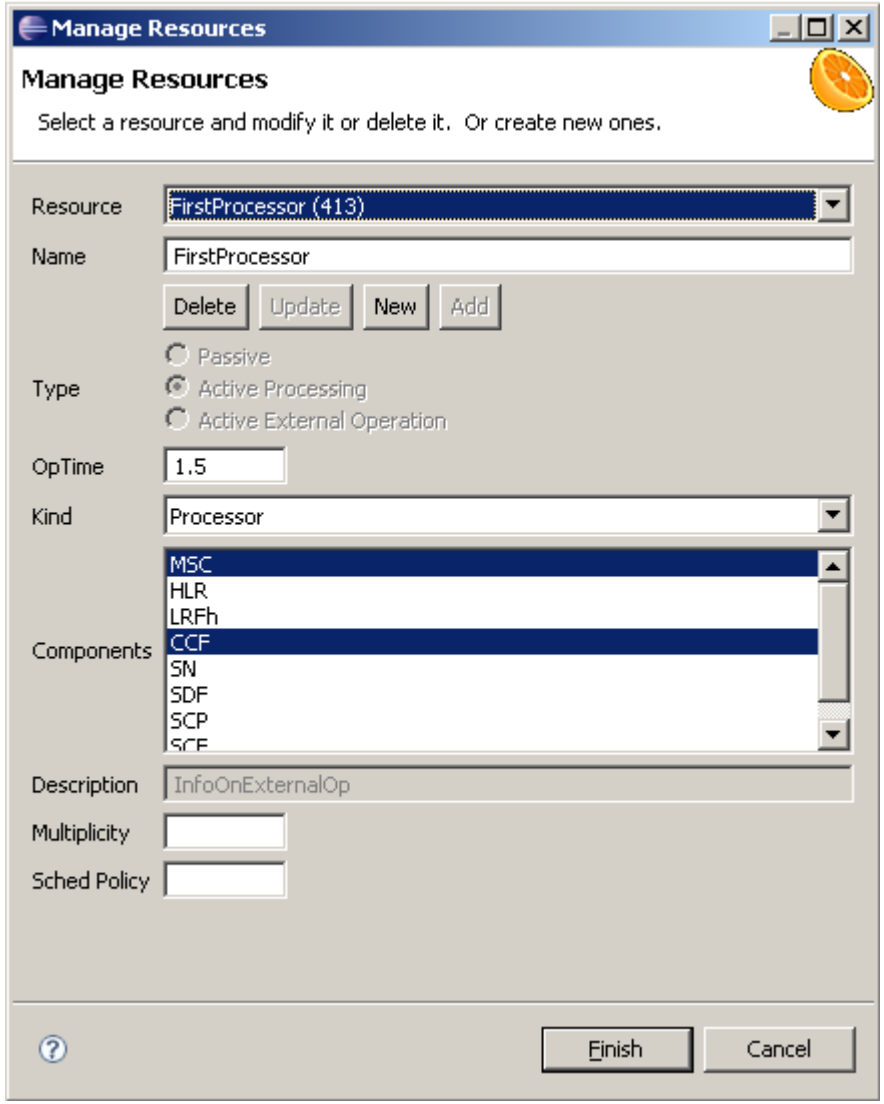
Example: Performance Annotations for UCMs



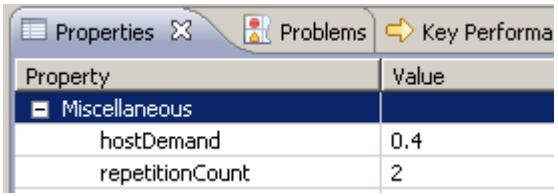
- Automated translation to *Core Scenario Model (CSM)* for analytical evaluations and simulations

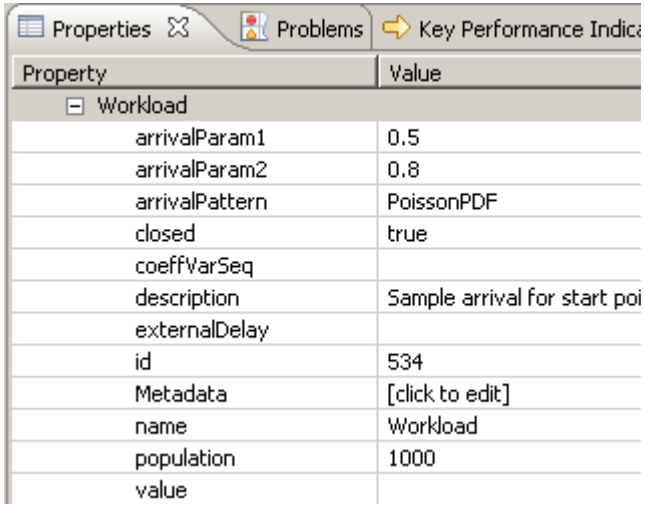
Resource Management

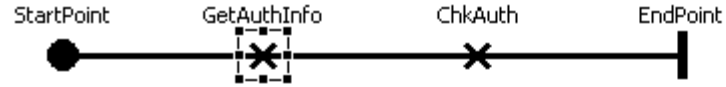


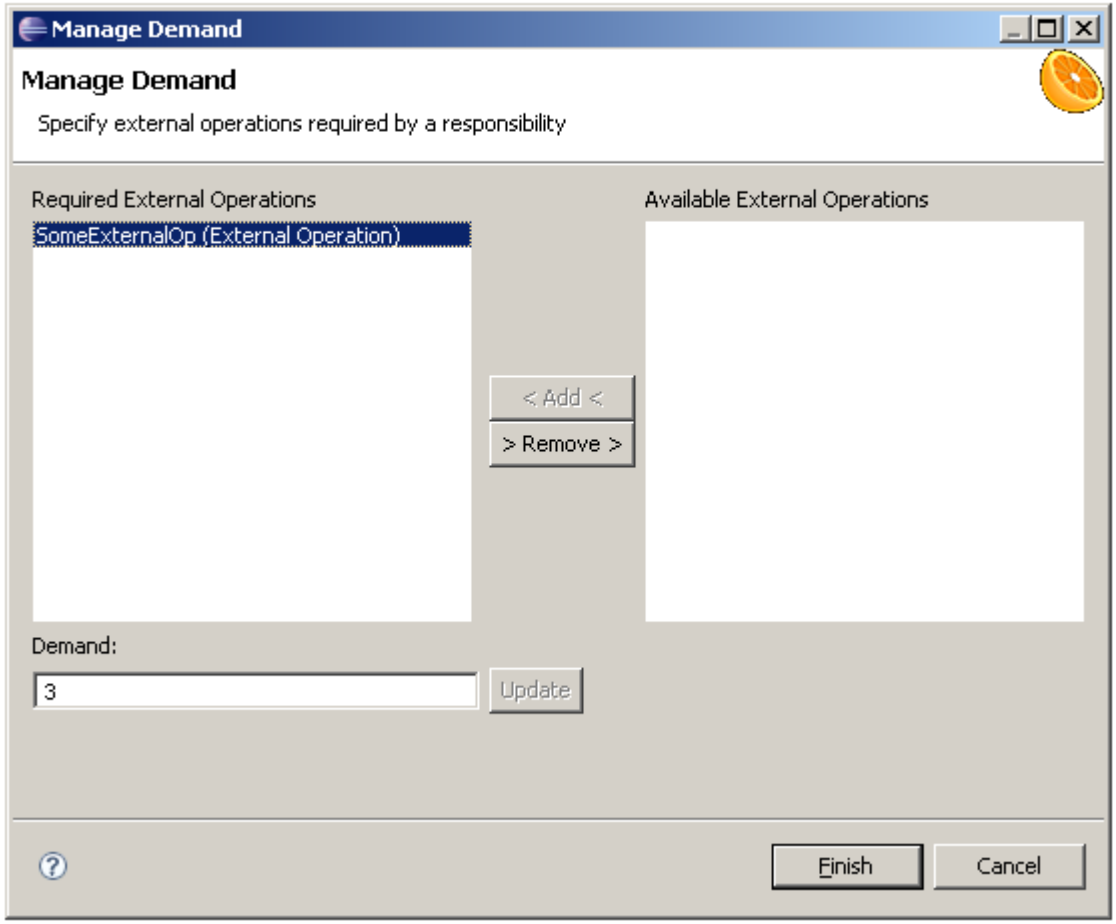


Demand and Workload Management



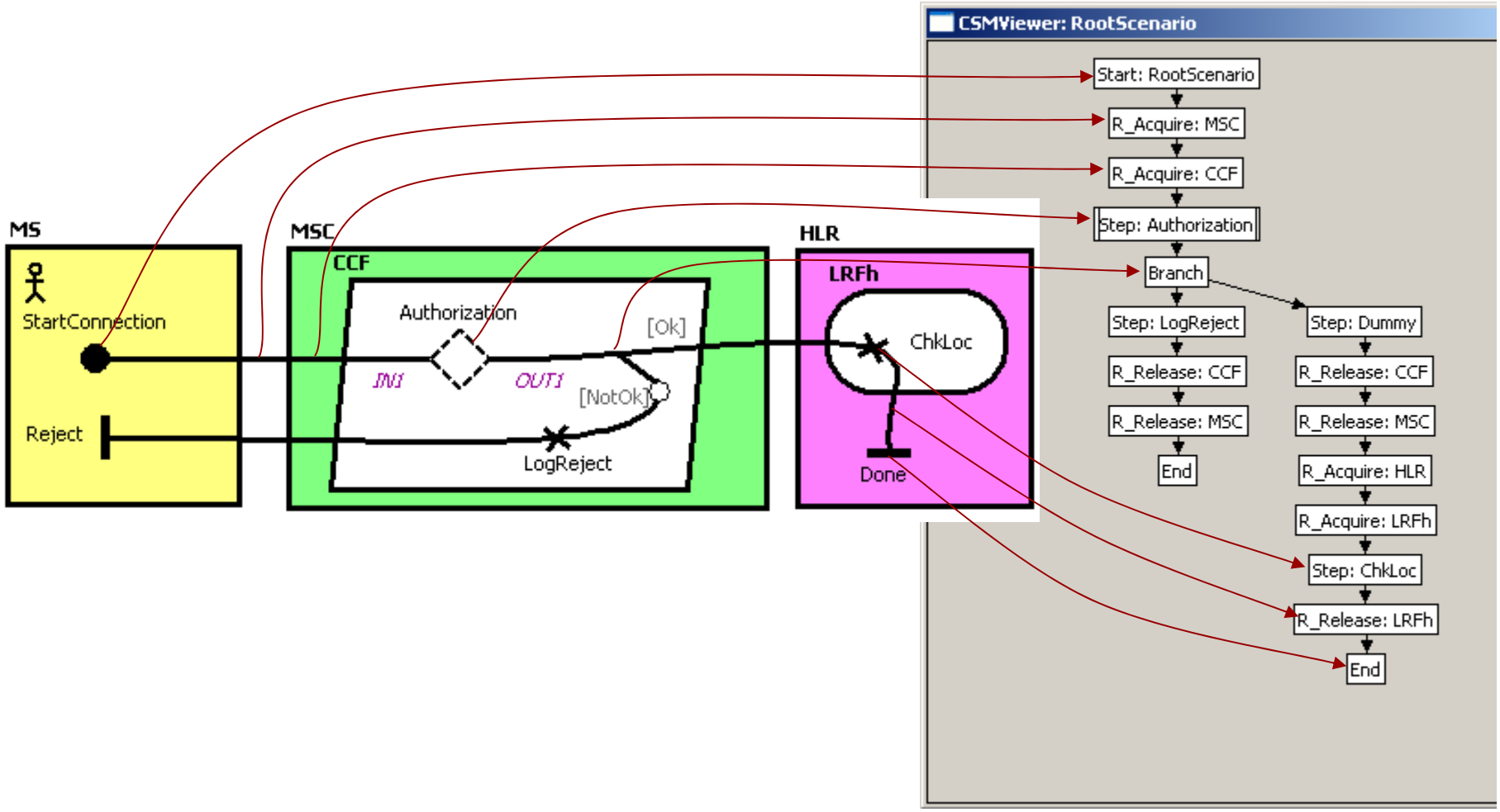






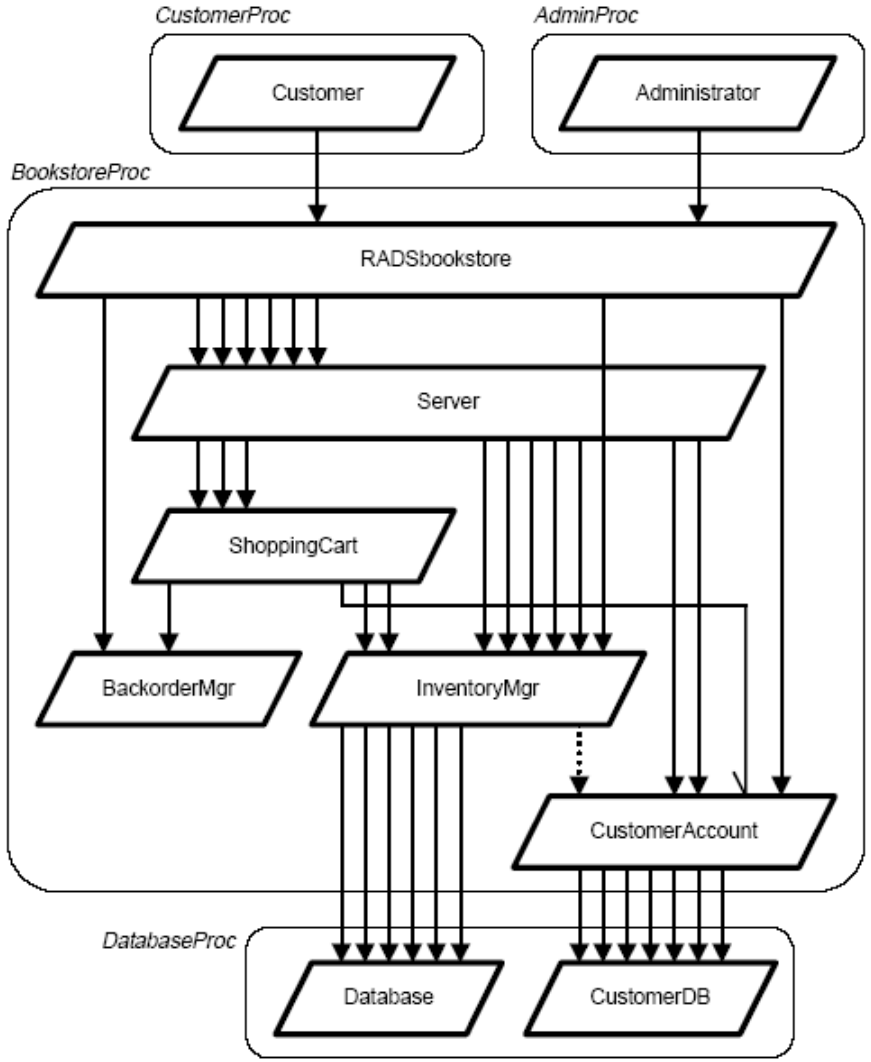
From UCM to Core Scenario Model (CSM)

- Export CSM (XML) from URN model
- Translation of CSM file to LQN, QN, stochastic Petri Nets…



LQN Generation from UCMs (2)

- Useful for various types of analyses
  - Sensitivity (importance or impact of parameters)
  - Scalability (what if there are more users/requests?)
  - Concurrency (what if there are more/fewer threads?)
  - Deployment and configuration (different hardware allocation)
- Quantitative evaluation of architecture!



Source: D.B. Petriu et al., 2003

Some industrial experience

- Bochmann did some consulting for DMR in the 1990ies.
- DMR is a big software and systems analysis consulting company based in Montreal.
- At that time, the DMR used internally a tool that allowed to define workflow models with resource constraints and to perform simulated execution of these models in order to evaluate the performance of such models depending on the available resources and scheduling constraints.
- The tool was used in the consulting assignments that were performed by the company.

Model transformations: Test case derivation

Model-based testing

- Behavioral models can be used for
  - Deriving test cases
  - Providing an oracle that predicts the correct output expected for given inputs. (However: if the behavioral model is non-deterministic – for a given input there may be different outputs – then this is quite difficult)
- This is black-box testing – the system implementation under test is observed only at its external interfaces – no internal view
- Test cases – two complementary coverage issues
  - Covering different control flows through the behavior
  - Covering different data parameter values
  - Question of executability of given control flow path with given data parameters

Coverage issues for black-box testing

- Issues of control flow coverage
  - All branches of the behavioral model will be exercised at least once
    - E.g. so-called transition tour for FSM model
  - All paths … (leads normally to too many test cases)
  - Covering all faults – one needs a fault model
    - Fault model for FSMs:
      - Output faults (wrong output produced): will be detected by transition tour
      - Transfer faults (wrong next state): difficult to detect - either introduce state visibiility, or use so-called state identification test sequences

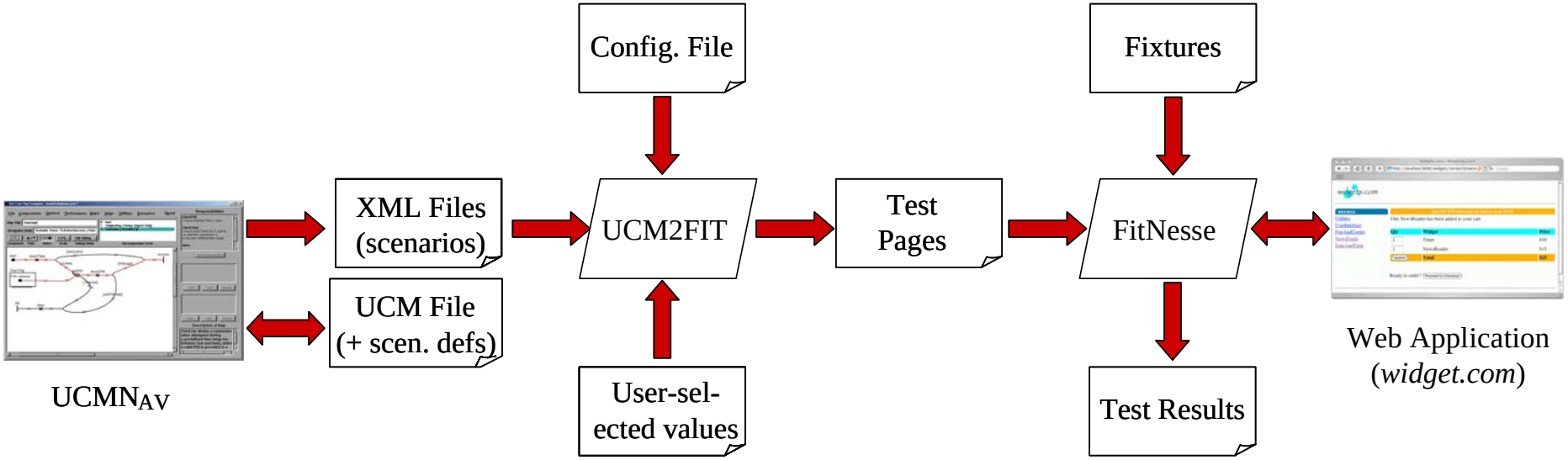
Automating test development from models ?

- FSM models:
  - There has been much work on deriving test suites (sets of test cases) from FSM models (for different coverage criteria)
- UCM models:
  - Deriving sequence diagram (test case – without data) for each scenario that can be realized from the given UCM
  - Automatic generation of scenarios and corresponding test cases (see next page)

Test Generation for Web Applications

There are some partial results available…

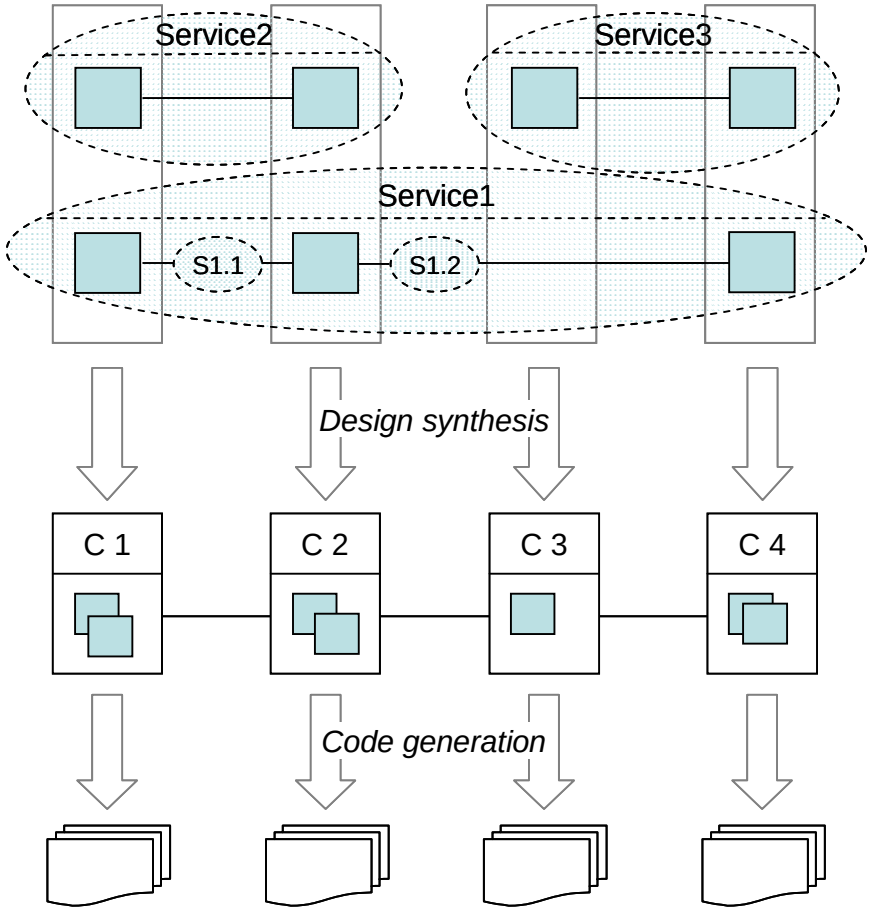
- Use of jUCMNav, scenario definitions, and *Fitnesse* to generate executable test cases for a typical Web application



Source: Amyot, Roy, and Weiss, 2005

Model transformations: Deriving distributed system designs

The problem



Service models

Design models

Implementations

This is an ongoing research project by Bochmann's group

Type of applications

- Communication services
  - telephony features (e.g. call waiting)
  - teleconference involving many parties
  - Social networking
- Workflows
  - Intra-organization, e.g. banking application, manufacturing
  - inter-organisations, e.g. supply-chain management
  - Different underlying technologies:
    - Web Services
    - GRID computing
    - Cloud computing
  - Dynamic partner selection: negotiation of QoS – possibly involving several exchanges

The problem

(early phase of the software development process)

- Define
  - Global functional requirements
  - Non-functional requirements
- Make high-level architectural choices
  - Identify system components
  - Define underlying communication service
- Define behavior of system components:
  - Locally performed functions
  - Communication protocol
    - Required messages to be exchanged and order of exchanges
    - Coding of message types and parameters

Issues

- Define
  - Global functional requirements
  - Non-functional requirements
- Make high-level architectural choices
  - Identify system components
  - Define underlying communication service
- Define behavior of system components
  - Local functions
  - Protocol:
    - Required messages to be exchanged and order of exchanges
    - Coding of message types and parameters

What language / notation to use for defining global requirements (dynamic behavior)

Architectural choices have strong impact on performance

Automatic derivation of component behaviors ? e.g. [Bochmann 2007]

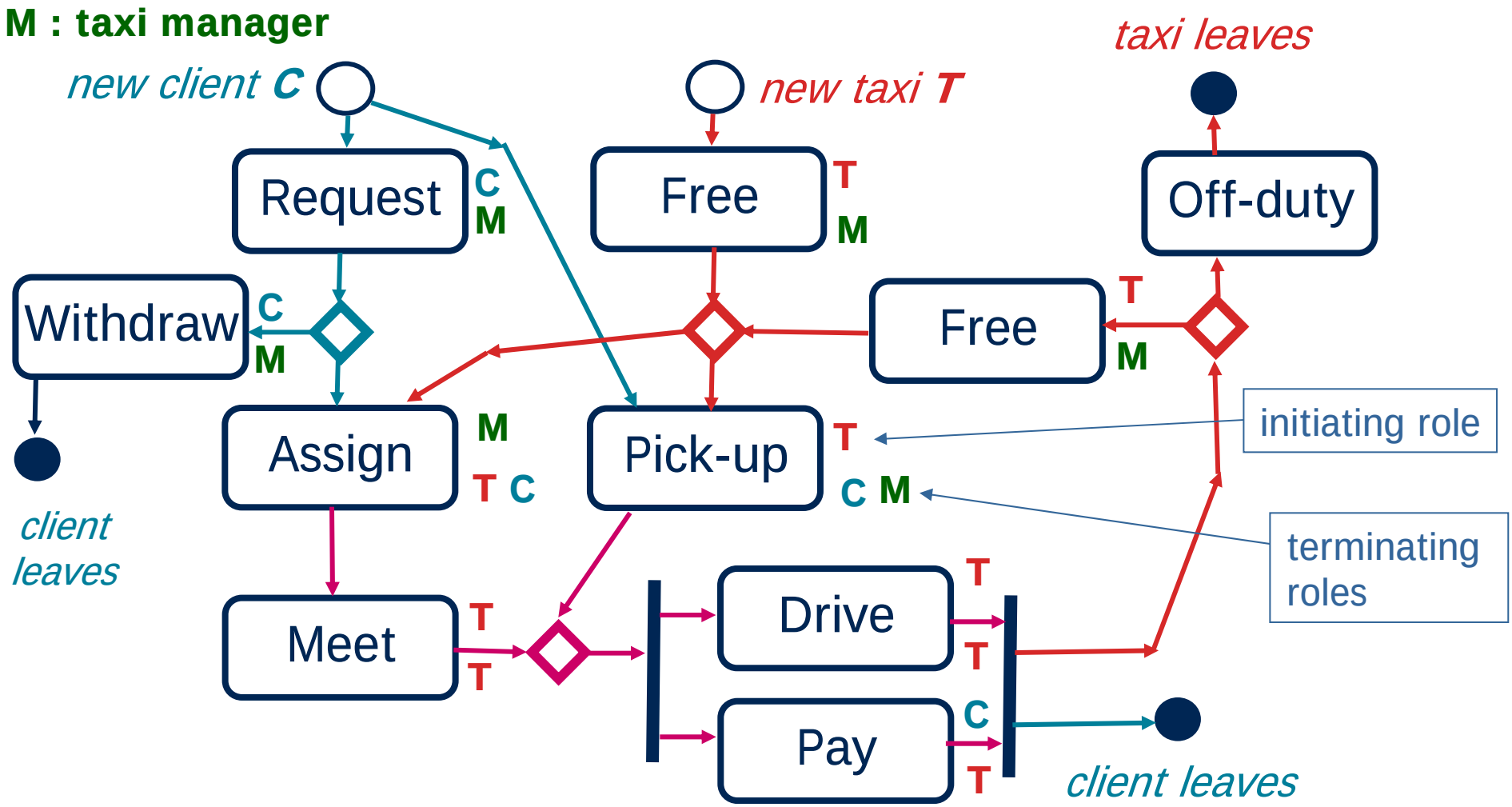
Performance prediction – based on component behavior

- Response time, Throughput, Reliability

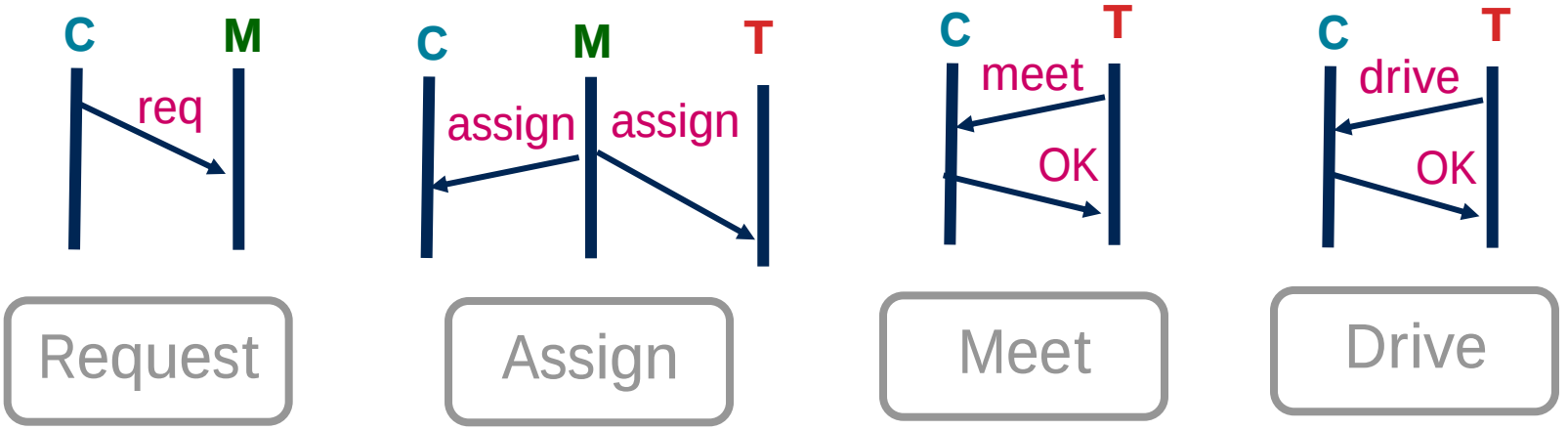
Choice of middleware platform for inter-process communication

- E.g. Java RMI, Web Services, etc.

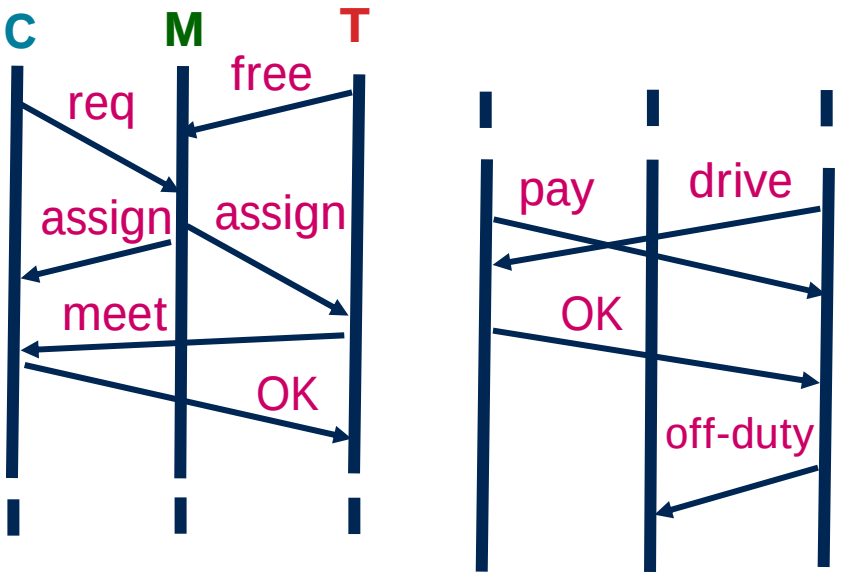
Example: Taxi system (an activity diagram - each activity is a collaboration between several roles: client, taxi, manager)



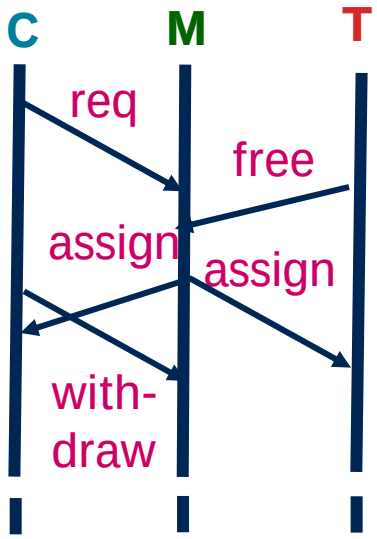
Detailed definitions of collaborations



Example scenario
(sequence diagram)

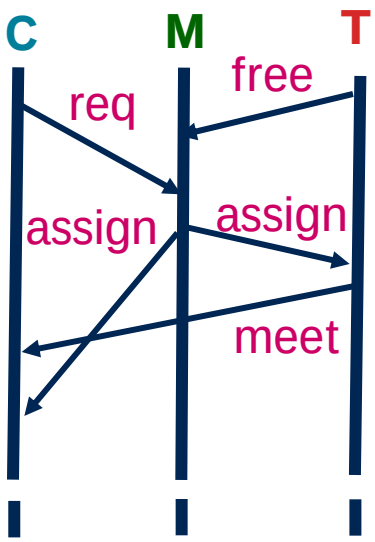


Taxi System : Problematic scenarios



non-local
choice

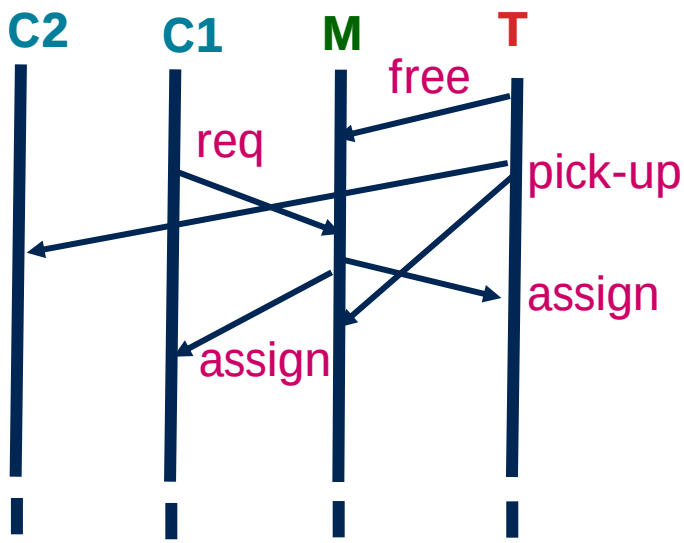
[Gouda 84] suggests:
define different priorities
for different roles



race
condition

"implied scenario":

[Alur 2000] component behaviors
that realize the normal scenario
will also give rise to implied scenarios



non-local
Choice
(conflict over taxi)

Model transformations: Deriving prototype implementations

Code generation from behavioral models

- This has been explored in research projects since the 1980ies for extended FSM languages and commercial tools have been around since the 1990ies, in particular for SDL, Statecharts and other notations – now also for UML State machines.