



Software Engineering and Formal Specification

CSIT - 31012

Dr. Nadeem Akhtar

Assistant Professor

Dept. of Computer Science & IT

The Islamia University of Bahawalpur

Course Outline (1/7)

Software Engineering and Formal Specification

- Course code: CSIT-31012 - (Credit Hours: 3)
- Dr. Nadeem Akhtar

Objectives:

1. At the end of the course students would be able to:
2. Describe the costs and benefits of formal methods
3. Verify attributes of formal models
4. Demonstrate formal correctness of simple procedure
5. Construct formal models of sequential software systems

Course Outline (2/7)

6. Implement sequential software systems based on formal models
7. Represent software systems with both state-based and process algebra models
8. Specify software systems formally, and reason about specifications, and verify their properties.
9. Connect specifications to programs through refinement and decomposition.
10. Use theorem proving tools.
11. Use model checking tools.

Course Outline (3/7)

Course Outline:

- Software Engineering: A Preview
- Introduction
- The software Life-cycle
- Software: Its Nature and Qualities
- Software Representative Qualities
 - Correctness, Reliability, and Robustness
 - Performance, Efficiency, Usability, Verifiability, Maintainability,
 - Repairability, Evolvability, Reusability, Understandability
 - Interoperability, Productivity

Course Outline (4/7)

Software Engineering Principles

- Rigor and Formality
- Separation of Concern
- Modularity
- Abstraction
- Anticipation of Change
- Generality
- Incrementality

Course Outline (5/7)

- Formal methods, formal specifications, formal verification.
- Formal languages, methods and techniques.
- Introduction to the use of mathematical models for specification and validation.
- Finite state machine models, models of concurrent systems, verification of models, and limitations.
- Transformational development
- Specification analysis and proof
- Program verification
- Objects and types: Sets and set types
- Tuples and Cartesian product types

Course Outline (6/7)

- Bindings and schema types, Relations and functions
- Properties and schemas, Generic constructions, Syntactic conventions, Schema references.
- Schema texts, Predicates, Schema expressions, Generics, Sequential Systems.
- Analyzing correctness (e.g. static analysis, simulation, model checking, etc.).
- Formal analysis.
- Analyzing well-formedness (e.g. completeness, consistency, robustness, etc.).
- Comparative Formal Methods.
- Formal Proofs.
- Introduction to Coloured Petri Nets (CP-Nets)
- Introduction to Event-B

Course Outline (7/7)

Reference Materials:

Fundamentals of Software Engineering, Second edition

Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli

Software Engineering: A Practitioner's Approach, Eighth edition

[Roger Pressman](#)

Publisher: McGraw-Hill. 2015

Artificial Intelligence, A Modern Approach, Third edition

Stuart J. Russell and Peter Norvig

Publisher: Pearson

Modern Formal Methods and Applications

Hossam A. Gabbar

Publisher: Springer-Verlag 2006

Software Engineering

–Software Engineering deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers

- Multiple versions
- Are in service for many years
- Undergo changes during their lifetime to fix defects, to enhance existing features, to add new features, to remove old features, or to be adapted to run in a new environment

Software Engineering

- [IEEE Std 610.12-1990's]

*The application of a **systematic, disciplined, quantifiable** approach to the development, operation, and maintenance of software*

- **Classical Engineering:**
 - Tools and mathematical training to specify the properties of the product separately from those of the design.

Example: An Electrical engineer relies on mathematical equations to verify that a design will not violate power requirements.

- **Software Engineering:**
 - **Mathematical tools are not well-developed**
 - **Relies more on experience and judgment rather than mathematical techniques.**

Role of a Software Engineer

- Good programmer – data structures and algorithms – fluent in one or more programming languages.
- Familiar with several design approaches
- **Be able to translate vague requirements into precise specifications.**
- Able to converse with the user of a system in terms of application.

Role of a Software Engineer

- Ability to move among **several levels of abstraction** at different stages of the project, from specific application procedures and requirements, to abstractions for the software system, to a specific design for the system and finally to the detailed coding level.

Role of a Software Engineer

- **Develop skills to build a variety of models and to reason about those models.**
- **Different models are used in the requirement phase, in the design of the software architecture, and in the implementation phase.**

Role of a Software Engineer

- Model might be used to answer questions about both the behavior of the system and its performance

Software Life Cycle (1/6)

- **Requirement analysis and specification.**
 - After Feasibility study (precise costs and benefits of the software system)
 - Identify and document the exact requirements for the system.
 - Produce User manuals, plans for the system test.

What the problem is

Software Life Cycle (2/6)

- **System design and specification**
 - ***Architectural or High-level design:*** defines the overall organization of the system in terms of high-level components and interactions among them.
 - ***Detailed design:*** components are decomposed into lower level modules with precisely defined interfaces.
- All design levels are documented in specification documents that keep track of the design decisions.

How to solve the problem

Software Life Cycle (3/6)

- *The Requirements phase attempts to specify What the problem is*
- *The purpose of design phase is to specify a particular software system that will meet stated requirements*

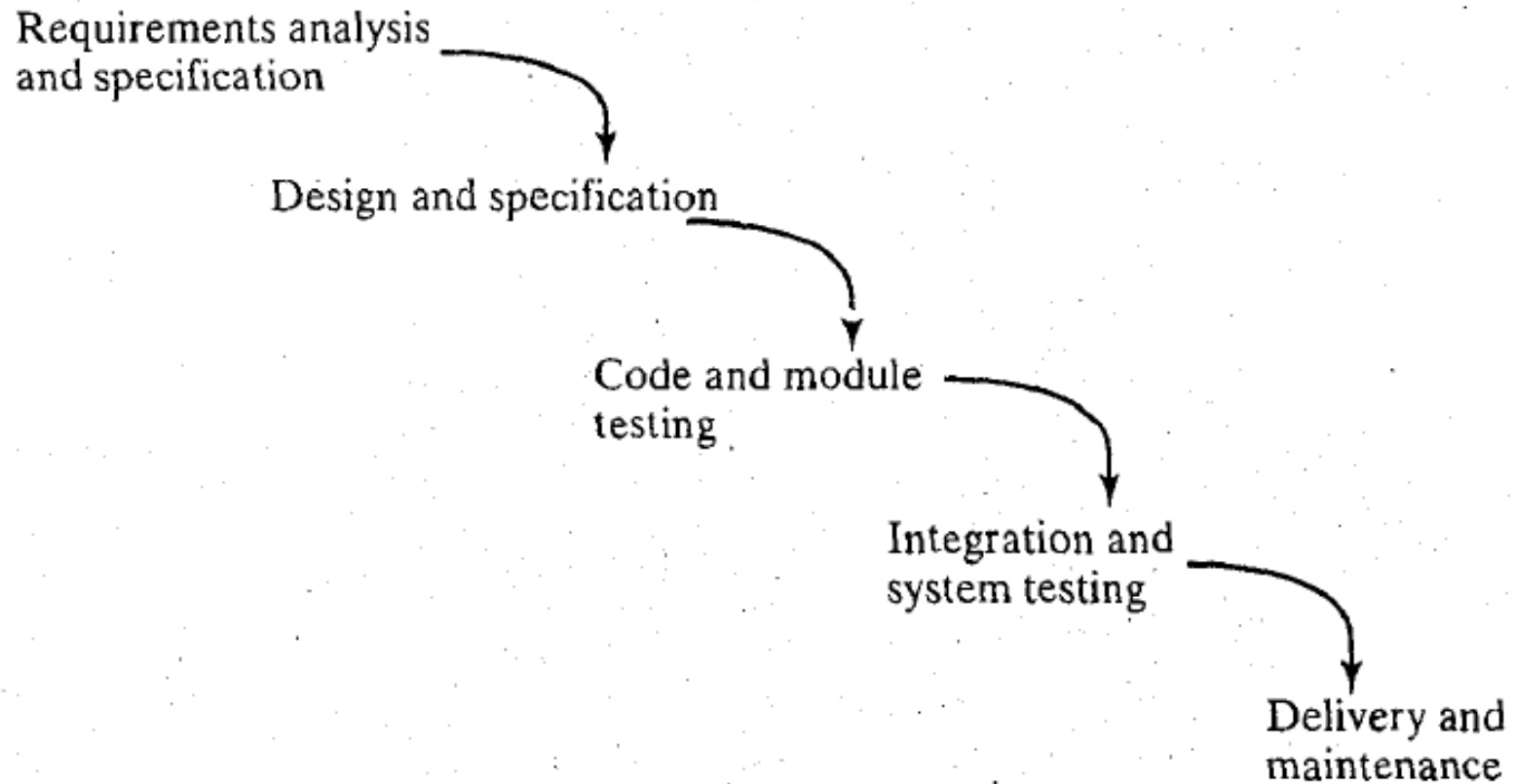
Software Life Cycle (4/6)

- **Coding and module testing**
 - *Code that will be delivered to the customer as the running system.*
 - *Individual modules developed in coding phase are tested before delivered to the next phase*
 - *Other phases of the life cycle may also develop code, such as that for **prototypes, tests, and test drivers.***

Software Life Cycle (5/6)

- **Integration and system testing**
 - *Modules developed and tested individually are put together-integrated- in this phase and are tested as a whole system.*
- **Delivery and maintenance**
 - The system passes all the tests are delivered to the customer and enters the maintenance phase.

Software Life Cycle (6/6)



Correctness (1/5)

- Program provide functions specified in its **functional requirements specifications**
- A program is *functionally correct* if it behaves according to its stated functional specifications.

Correctness (2/5)

- **Assumption:** Requirement specifications are available and it is possible to determine unambiguously whether a program meets the specifications.

Correctness (3/5)

- *If requirement specification does exist, it is usually written in an informal style using natural language. Therefore it is most likely to contain many ambiguities.*
- *The Solution is to use formal methods to write mathematical based, concise and precise specifications which are well-defined.*

Correctness (4/5)

- *Correctness is a mathematical property that establishes the equivalence between the software and its specification*
- *Correctness can be assessed more systematically and precisely depending upon how rigorous we are in specifying functional requirements.*

Correctness (5/5)

- *Correctness of a program may be assessed through a variety of methods*
 - *Experimental approach e.g. testing*
 - *Analytic approach e.g. inspections or formal verification*
 - *Tools supporting extensive static analysis.*
 - *Using standard proven algorithms or libraries of standard modules.*

Reliability (1/2)

- *Software is reliable if user can depend on it (Dependability).*
- *Reliability is the probability that the software will operate as expected over a specified time interval.*

Reliability (2/2)

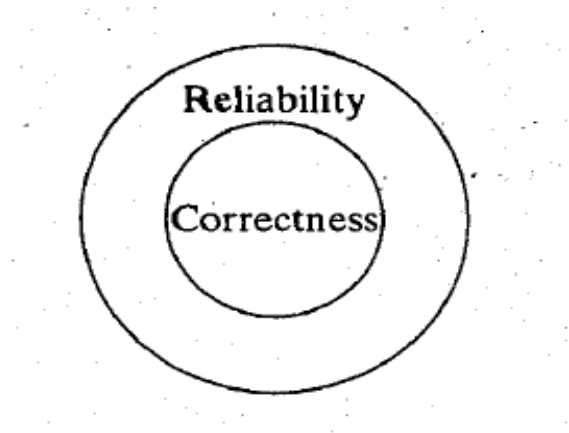
- *Correctness is a mathematical absolute quality: any deviation from its requirements makes a system incorrect. (deviation: minor or serious)*
- *Reliability is relative: If the consequence of a software error is not serious, the incorrect software may still be reliable.*

Reliability and Correctness (1/2)

- **Assumption:** functional requirement specifications captures all the desirable properties of the application and that no undesirable properties are erroneously specified in it.
- The set of all reliable programs includes the set of correct programs, but not vice versa.

Reliability and Correctness (2/2)

- Idealized situation requirements are assumed to be correct.
- Not all incorrect behaviors signify equally serious problems; that is, some incorrect behaviors may be tolerable.



Robustness (1/5)

- A program is robust if it behaves “reasonably” even in circumstances that were not anticipated in the requirements specification

Example

When a program encounters incorrect input data or some hardware malfunction (i.e. a disk crash).

Robustness (2/5)

- **A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command is not robust.**
- A program might be correct, though, if the requirements specification does not state what the program should do in the face of incorrect input.

Robustness (3/5)

- Robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its “reasonable” behavior completely. **Thus, robustness would become equivalent to correctness.**

Robustness (4/5)

- **Example:**

Two bridges connecting two sides of the same river are both “correct” if they satisfy the stated requirements.

If however during an unexpected, unprecedented earthquake, one collapses and the other one does not, we can call the latter more robust than the former.

Robustness (5/5)

Lesson learned from the collapse of the bridge will probably lead to more complete requirements for future bridges, establishing resistance to earthquakes as a correctness requirement.

Robustness and Correctness

In conclusion, we can see that robustness and correctness are strongly related, without a sharp dividing line between them.

If we put a requirement in the specification, its accomplishment becomes an issue of correctness; if we leave the requirement out of the specification, it may become an issue of robustness. The border line between the two qualities is the specification of the system.

Correctness, Robustness & Reliability

In relation to the software production process.

A process is robust, for example, if it can accommodate unanticipated changes in the environment, such as new release of the operating system or the sudden transfer of half the employees to another location.

Correctness, Robustness & Reliability

A process is reliable if it consistently leads to the production of high-quality products.

Performance (1/3)

Efficiency is an internal quality and refers to how economically the software utilizes the resources of the computer.

Performance on the other hand, is an external quality based on user requirements.

For example: A telephone switch may be required to be able to process 10,000 calls per hour.

Efficiency affects, and often determines, the performance of a system

Performance (2/3)

Performance is important because it affects the usability of the system.

If a software system is too slow, it reduces the productivity of the users, possibly to the point of not meeting their needs.

If a software system uses too much disk space, it may be too expensive to run.

If a software system uses too much memory, it may affect the other applications that run on the same system (run slowly as OS tries to balance the memory usage of different applications).

Performance (3/3)

Performance is important because it affects the scalability of a software system.

An algorithm that is quadratic may work on small inputs, but not work at all on larger inputs.

A compiler that uses register allocation algorithm whose running time is the square of the number of program variables will run more and more slowly as the length of the program being compiled increases.

Techniques of Performance Evaluation

- *Analyze the Complexity of Algorithms used in the software.*
 - Average or Worst case behavior of algorithms – in terms of resource requirements e.g. time and space – in case of distributed systems in terms of number of messages exchanges.
- *Measurement Analyses*
 - Actual performance measurement by hardware or software monitors that collect data while the system is running.

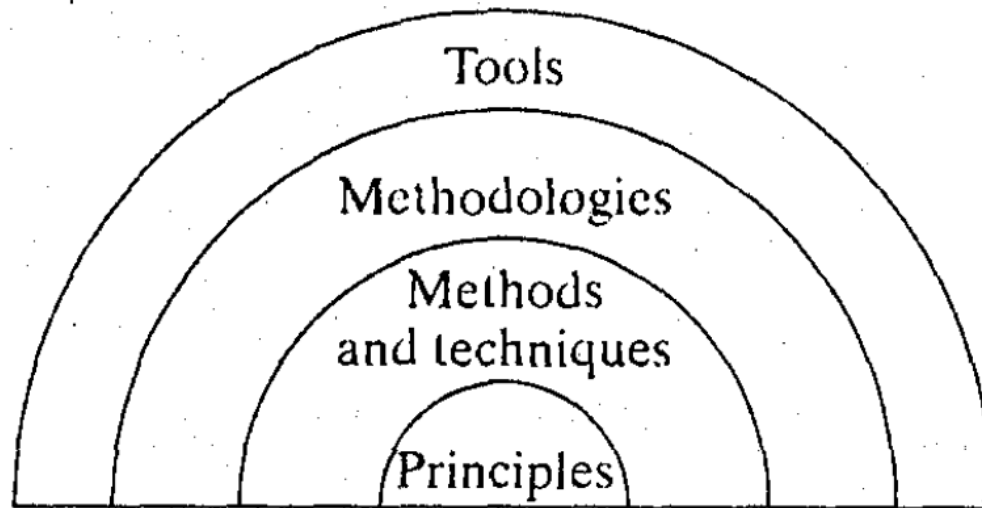
Techniques of Performance Evaluation

- *Build a model of the product and analyze it.*
- *Build a model that simulates the product.*
 - **Analytical model** – easier to build but less accurate
 - **Simulation models** – costly to build but are more accurate.
 - Combine the two techniques – At start an analytic model to identify performance critical areas – simulation model of the performance critical areas

Software Engineering Principles

- **Principles** are general enough to be applicable throughout the process of software construction and management.
- **Principles** are not sufficient alone to drive software development.
- **Principles** are general and abstract statements describing desirable properties of software processes and products.

Relationships between principles, methods and techniques, methodologies, and tools



Principles

- 1- Rigor and Formality**
- 2- Separation of Concerns**
- 3- Modularity**
- 4- Abstraction**
- 5- Anticipation of Change**
- 6- Generality**
- 7- Incrementalism**

Methods and Techniques

- To apply principles, the software engineer should be equipped with **methods** and **techniques** that help incorporate the desired properties into processes and products.

Methods and Techniques

- **Methods** are general guidelines that govern the execution of some activity; they are **rigorous, systematic, and disciplined** approaches.
- **Techniques** are more technical and mechanical than methods. They have more restricted applicability.

Principles

Rigor and Formality (1/8)

Rigor defined as **precision** and **exactness**.

Rigorous approach – that can repeatedly produce reliable products, control their costs, and increase our confidence in their reliability.

Highest degree of Rigor is **formality**.

Rigor and Formality (2/8)

- Formality is stronger requirement than rigor. It requires the software process to be driven and evaluated by mathematical laws.
- Formality implies rigor, but the converse is not true. One can be rigorous and precise even in an informal setting.

Rigor and Formality (3/8)

- Example

Engineer can rely on past experience and rules of thumb to design a small bridge, to be used temporarily to connect the two sides of a creek.

If the bridge is large and permanent one, engineer should use a mathematical model to verify whether the design is **safe**.

Engineer would use a more sophisticated mathematical model if the bridge is exceptionally long or if it is built in an area of seismic activity.

Rigor and Formality (4/8)

Engineer (mathematician) must be able to identify and understand the level of rigor and formality that should be achieved depending on the difficulty and criticality of the task. The level may even vary for different parts of the same system.

Rigor and Formality (5/8)

- **Example:**

Critical parts - such as the scheduler of a real-time operating systems kernel.

Security component of an electronic commerce system.

Must have a formal description of their intended functions and a formal approach to their assessment.

Well-understood and standard parts would require simpler approaches.

Rigor and Formality (6/8)

- The description of what a program does may be given in a rigorous way by using natural language;
- It can also be given formally by providing a formal description in a language of logical statements.

Rigor and Formality (7/8)

- The advantage of formality over rigor is that formality may be basis of mechanization of the process.

For example:

One can use the formal description of the program to create the program (if the program does not exist yet) or to show that the program corresponds to the formal description (if the program and its formal specification exist)

Rigor and Formality (8/8)

- Programming:

Programs are formal objects.

They are written in a language whose **syntax** and **semantics** are fully defined.

Programs are formal descriptions that may be automatically manipulated by compilers; They are checked for formal correctness.

Separation of Concerns (1/8)

- Allows us to deal with different aspects of a problem, so that we can concentrate on each individually.
- It is a common-sense practice that we try to follow in our everyday life to overcome the difficulties we encounter.

Separation of Concerns (2/8)

- *Software development – to master inherent complexity of software*
 - Decisions concern features of product; functions to offer, *expected reliability, efficiency with respect to space and time, user interfaces*
 - Decision concern the development process: *the development environment, the organization and structure of teams, scheduling, control procedures, design strategies, error recovery mechanisms.*
 - Decision concerns *economic and financial matters.*

Separation of Concerns (3/8)

- Separate concerns in time

— For example:

University Professor i.e. Scheduling teaching-related activities such as holding classes, seminars, department meetings from 9 a.m. to 2 p.m. Engaging in research the rest of the time.

Separation of Concerns (4/8)

Example:

- Separate concerns in terms of *Qualities* of software that should be treated separately
- In case of software *Correctness* and *Efficiency* are dealt separately
 - *Correctness* is given priority (checked by formal correctness proofs and Complexity analysis) then restructure the program to improve its *Efficiency*.

Separation of Concerns (5/8)

Example:

- Separate concerns in terms of different *Views* of the software to be analyzed separately
- During requirement analysis of software
 - One view of **Flow of data** from one activity to another in a system.
 - Another view of **Flow of Control** that govern the way different activities are synchronized.

Separation of Concerns (6/8)

Example:

- Separate concerns in terms of *Size* allows to deal with *parts* of the same system separately
 - Separation by using *Modularity*

Separation of Concerns (7/8)

Example:

- Separate **Problem-domain** concerns from **Implementation-domain**.
- Problem-domain properties hold in general regardless of the implementation-domain.
 - **Employees Management System:** Problem-domain issues about employees in general
 - **Implementation-domain:** implementation of an employee as a data structure or object. Employee A reports to employee B. One object pointing to another.

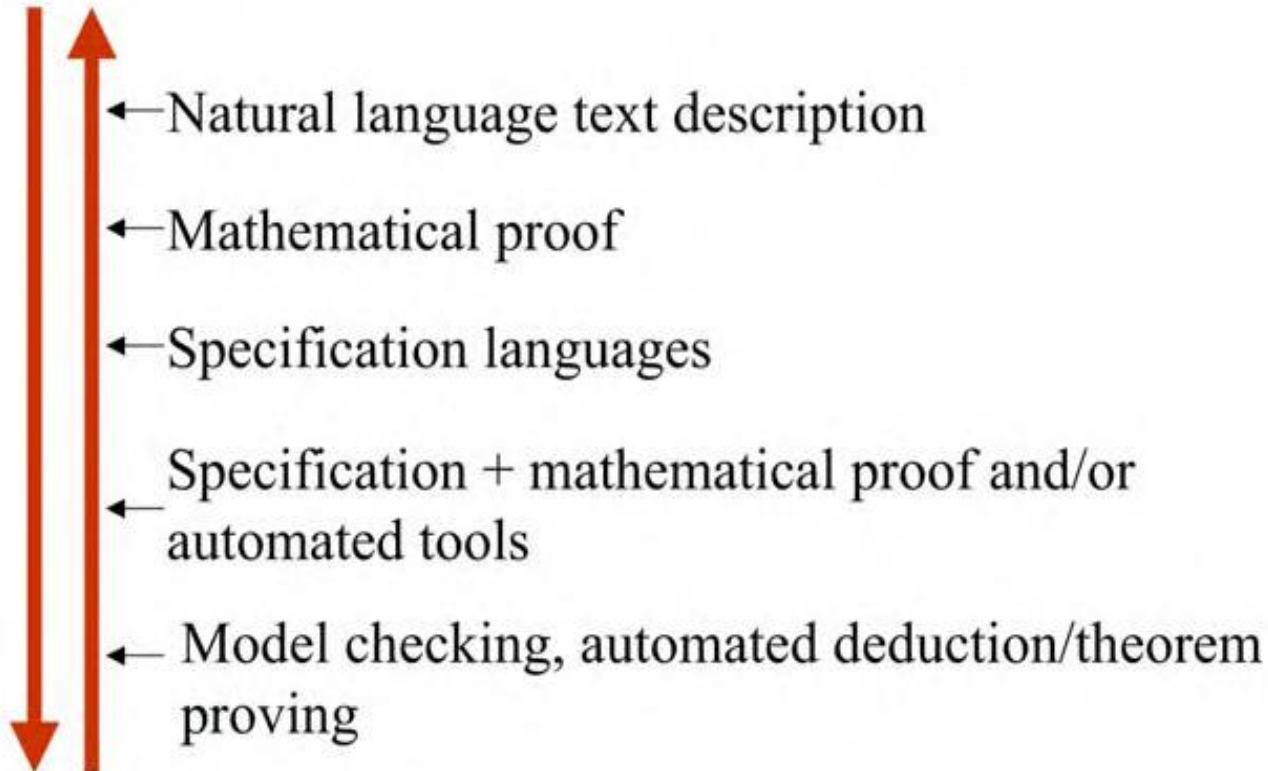
Separation of Concerns (8/8)

Example:

- Separate **Managerial** and **Technical** issues in the software process.
- An analyst concentrating separately on **functional** and **non-functional** system requirements.

Formalization Spectrum

Less Formal



More formal

Formal Methods (1/11)

- Formal methods encompass a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable to *specify*, *develop*, and *verify* a computer-based system by applying a *rigorous, mathematical notation*.

Formal Methods (2/11)

“Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a Systematic, rather than adhoc manner” [Mar01]

[Mar01] Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, 2nd edition, Wiley, 2001.

Formal Methods (3/11)

- The desired properties of a formal specification - *consistency*, *completeness*, and *lack of ambiguity* - are the objectives of all specification methods.

Mathematically based specification language used for formal methods results in a much higher likelihood of achieving these properties

Formal Methods (4/11)

- The ***formal syntax*** of a specification language enables requirements or design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g. English) or a graphical notation (e.g. UML) must be interpreted by a reader.

Formal Methods (5/11)

- The descriptive facilities of *set theory* and *logic notation* enable a clear statement of requirements. To be consistent, requirements stated in one place in a specification should not be contradicted in another place.
- Consistency is achieved by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification.

Formal Methods (6/11)

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms.

Ambiguity, incompleteness, and inconsistency
can be discovered and corrected through the application of mathematical analysis.

Formal Methods (7/11)

When formal methods are used during ***design***, they serve as a basis for program verification and therefore enable to discover and correct errors that might otherwise go undetected.

Formal Methods (8/11)

- Models developed using formal methods are described using a formal syntax and semantics that specify system *function* and *behavior*.
- The specification in mathematical form (e.g., predicate calculus can be used as the basis for a formal specification language)

Formal Methods (9/11)

– Why formal?

- **Correctness** of software systems can be improved by formalizing different components and processes in the life-cycle.

Formal Methods (10/11)

- ▶ Are based on solid mathematical foundation:
- ▶ The objectives to have formal foundations are:
 - 1) to improve the quality of the whole development process;
 - 2) to enable rigorous analysis of the system properties;
 - 3) to be as certain as possible that the specifications, transformations and implementation is property-preserving and error-free;
 - 4) to provide a foundation for the adaptation and evolution process;
and
 - 5) to improve documentation and understanding of specifications.

Formal Methods (11/11)

- ▶ These are all based on mathematical representation and analysis of software
- ▶ Formal methods include:
 - ▶ Formal specification
 - ▶ Formal languages
 - ▶ Formal verification
 - ▶ Formal transformations from one model to another
 - ▶ Formal architecture

Formal Specification (1/4)

- ▶ Formal specifications are based on Formal methods.
- ▶ Formal specifications are *precise and unambiguous*. Techniques for the unambiguous specification of software.

Formal Specification (2/4)

- ▶ They remove area of doubts in a specification and avoid some of the problems of language misinterpretation.
- ▶ Non-specialists may find formal specifications difficult to understand.

Formal Specification (3/4)

- ▶ A formal software specification is a specification expressed in a language whose *vocabulary*, *syntax* and *semantics* are formally defined.
- ▶ This need for a formal definition means that the specification languages must be based on mathematical concepts whose properties are well understood.

Formal Specification (4/4)

- ▶ The branch of mathematics used are:
 - ▶ *Discrete mathematics* with the mathematical concepts drawn from set theory, logic and algebra;
 - ▶ *Calculus* with the mathematical concepts drawn from π -Calculus.

Use of formal methods

- ▶ Formal methods principal benefits are in reducing the number of errors in systems, so their main area of applicability is **critical systems**. (i.e. in this area, the use of formal methods is most likely to be cost-effective)
- ▶ The principal value of using formal methods in the software process is that it forces **an analysis of the system requirements at an early stage**. Correcting errors at this stage is cheaper than modifying a delivered system.

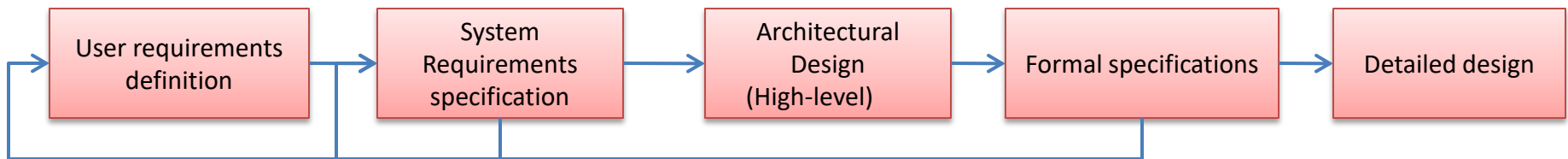
Formal specifications in the software process

- ▶ Analysis and design are intermingled (i.e. there are no well-defined boundaries)
- ▶ Architectural specifications play important role in formal software development process.
- ▶ Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and design

Increasing contractor involvement

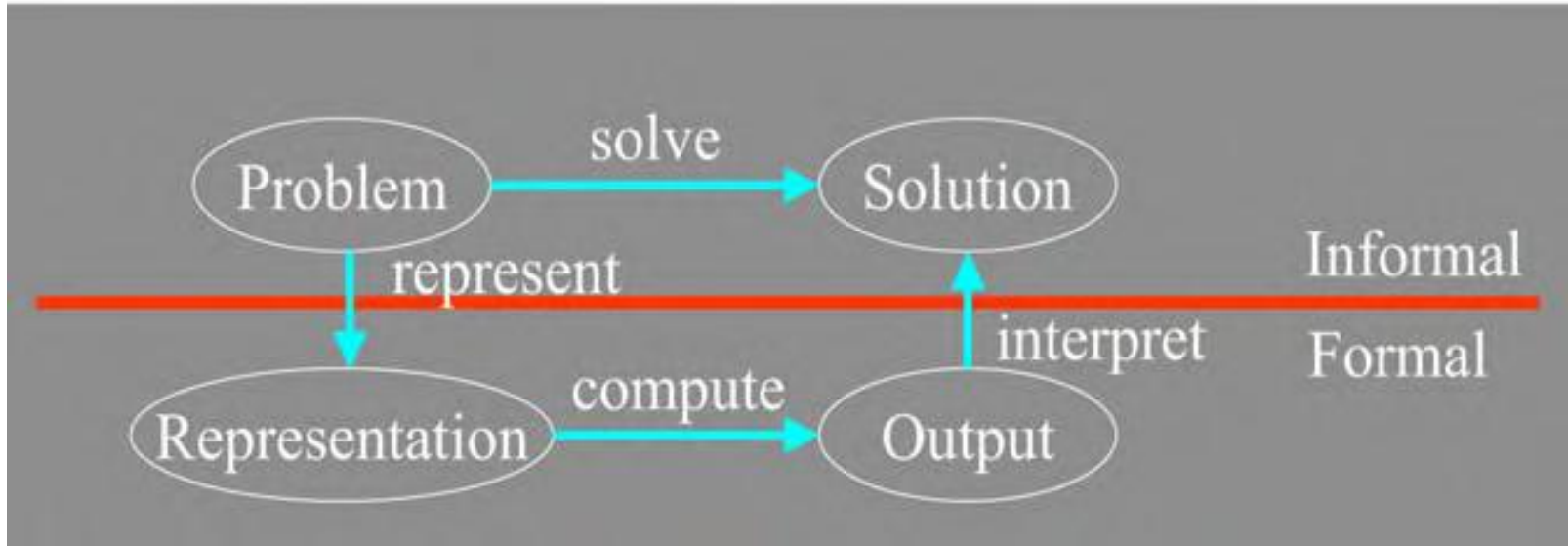
Decreasing client involvement



Analysis specification

Design specification

Problem Solving Framework



Formal Methods

Mathematically based techniques for the **specification, development and verification** of the underlying system.

Formal Methods

One of the first definitions of formal methods is:

*A broad view of formal methods includes all applications of (primarily) **discrete mathematics** to software engineering problems. This application usually involves modeling and analysis where the models and analysis procedures are derived from or defined by an underlying mathematically-precise foundation.*

Formal Methods

Formal methods support **precise** and **rigorous** specifications of those aspects of the underlying system capable of being used to manage the system throughout its life cycle.

Formal methods - Graphical

- Formal methods can include graphical languages.

For example, Data Flow Diagram (DFD) is a well-known graphical technique for specifying the function of a system. DFDs can be considered a semi-formal method

- Graphical languages in a completely formal manner.
 - Finite State Machines
 - Petri-nets [Peterson 77]

Formal methods

- Formal models perform *a rigorous analysis of the system (thereby improving the quality)* and allow to reuse the specification in the development of an implementation.
- In a formal development approach, we transfer some efforts from the test phase (i.e. verification of the implementation) to the specification phase (where the specification in relation to the requirements is verified).

Formal methods

Test reasoning versus model (blueprint) reasoning. (1/2)

- With the construction of complex discrete systems a very important activity in terms of time and money, is that of verifying that the final implementations are operating in a *correct* fashion.
- The validation of a discrete system by means of such testing i.e. “*laboratory executions*” is complicated.

Formal methods

Test reasoning versus model (blueprint) reasoning. (2/2)

- Nowadays a complex system construction concludes the construction process with a long and heavy testing phase.
- Program testing used as a validation process in almost all programming projects is far from being a complete process, because of the impossibility of achieving a total cover of all executing cases.
- There is *incompleteness* as a consequence of the *lack of oracles* which would give, *beforehand* and independently of the tested objects, the expected results of a future testing session.

Formal methods

Complex discrete systems. (1/2)

- Testing does not involve any kind of sophisticated reasoning. It rather consists of *always postponing any serious thinking* during the specification and design phase.
- The construction of the system will always be re-adapted and re-shaped according to the testing results (i.e. trial and error).
- In conclusion, testing always gives a shortsighted operational view of the system under construction: that of execution.

Formal methods

Complex discrete systems. (2/2)

- In other technologies, i.e. avionics, people eventually do test what they are constructing, but the testing is just the *routine confirmation* of a sophisticated design process rather than a fundamental phase in it.
- Most of the reasoning is done *before* the very construction of the final object.
- It is performed on various blueprints, in the broad sense of the term, by applying to them some well-defined practical theories. (Abrial, Modeling in Event-B System and Software Engineering, 2010)

Logic or Propositional Calculus (1/3)

Logic or propositional calculus is based on statements, which have **truth values** (true or false).

A proposition, or statement, is any declarative sentence, which is either true (T) or false (F).

We refer to T or F as the truth value of the statement.

The calculus provides a means of determining the truth values associated with statements formed from “atomic” statements.

Logic or Propositional Calculus (2/3)

If p stands for “pressure is high in pipe P_1 ” and q for “pipe P_1 is leaking” then we may form statements such as shown in table below

Table 1-1. Symbolic Statements

Symbolic Statement	Translation
$p \vee q$	p or q
$p \wedge q$	p and q
$p \Rightarrow q$	p logically implies q
$p \Leftrightarrow q$	p is logically equivalent to q
$\neg p$ (also $\sim p$)	Not p

Logic or Propositional Calculus (3/3)

Note that \vee , \wedge , \Rightarrow , \Leftrightarrow , are all binary connectives. They are sometimes referred to, respectively, as the symbols for ***disjunction***, ***conjunction***, ***implication*** and ***equivalence***.

Also \neg is unary and is the symbol for negation.

First-order Predicate Calculus (1/17)

- *Propositional logic* provide us with the means to assess the truth value of compound statements from the truth values of the “building blocks”
- **Rules** are required to do this.
- **For example**, the calculus states that “ $p \vee q$ ” is true if either p is true or q is true (or both are true).
- Similar rules apply for all the ways in which the building blocks can be combined. The language of predicate calculus requires:
 - Variables and Constants

First-order Predicate Calculus (2/17)

The language of predicate calculus requires:
Variables and Constants

Symbol	Meaning
\vee	or
\wedge	and
\neg	not
\Rightarrow	logically implies
\Leftrightarrow	logically equivalent
\forall	for all
\exists	there exists

First-order Predicate Calculus (3/17)

Quantification is non-logical constants that include names and entities.

For example:

$\forall x.\text{man}(x) \Rightarrow \text{mortal}(x)$, means all men are mortal.

$\exists x.\text{Tank}(x)$, means there is at least one tank.

First-order Predicate Calculus (4/17)

- It is possible to form a new proposition from old one.
- For example, p : "There is Pump with 300 rpm in Plant Model Plant-1."
- The negation of p is $\neg p$, which is defined as: "There is no Pump with 300 rpm in Plant Model Plant-1."
- Another example: if p : " $1 + 4 < 5$ ", q : " $1 + 4 = 5$ ", then $\sim p \wedge \sim q$: " $1 + 4 > 5$ ".

First-order Predicate Calculus (5/17)

- **Truth table** provides basic definition of predicate calculus.

(a) Negation

p	$\neg p$
T	F
F	T

(b) Disjunction

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

(c) Conjunction

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

First-order Predicate Calculus (6/17)

There are set of more complex expressions, that can be further simplified to validate the underlying system. Conditional and bidirectional are other forms of predicate calculus. For example, if p, then q can also be represented as: p implies q, and we write $p \Rightarrow q$. This can be represented as $(\neg p) \vee q$. Table 1-4 shows the truth table of such logic.

Table . $(\neg p) \vee q$.

p	q	$p \Rightarrow q$	$\neg p$	$(\neg p) \vee q$
T	T	T	F	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

First-order Predicate Calculus (7/17)

Table . Examples of Conditional Statements

If p , then q . p implies q .

q follows from p . Not p unless q .

q if p . p only if q .

Whenever p , q . q whenever p .

p is sufficient for q . q is necessary for p .

p is a sufficient condition for q . q is a necessary condition for p .

First-order Predicate Calculus (8/17)

The biconditional $p \Leftrightarrow q$, which we read "p if and only if q" or "p is equivalent to q," is defined by the truth table, shown in table .

Table . Biconditional Statements.

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

First-order Predicate Calculus (9/17)

Modus Ponens or Direct Reasoning can be defined as: $[(p \Rightarrow q) \wedge p] \Rightarrow q$, which can be represented as:

$p \Rightarrow q$	(if p then q, or p implies q)
p	(p, which is true or false)

$\therefore q$	

Similarly,

$(p \wedge q) \Rightarrow (r \wedge s)$
$(p \wedge q)$

$\therefore (r \wedge s)$

First-order Predicate Calculus (10/17)

Modus Tollens or Indirect Reasoning can be defined as:

$$[(p \Rightarrow q) \wedge \neg q] \Rightarrow \neg p$$

In words, if p implies q , and q is false, then so is p .

$$p \Rightarrow q$$

$$\neg q$$

$$\therefore \neg p$$

First-order Predicate Calculus (11/17)

Disjunctive Syllogism or One-or-the-Other:

$$[(p \vee q) \wedge (\neg p)] \Rightarrow q$$

$$[(p \vee q) \wedge (\neg q)] \Rightarrow p$$

Distributive Laws:

$$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$$

$$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$$

First-order Predicate Calculus (12/17)

Applying Modus Ponens

1. $(p \vee q) \Rightarrow (r \wedge \neg s)$
2. $\neg r \Rightarrow s$
3. $p \vee q$

Statement $(p \vee q)$ appears twice in lines (1) and (3). Looking at Modus Ponens, we see that we can deduce $(r \wedge \neg s)$ from these lines. Thus, we can enlarge our list as follows:

- | | |
|---|------------------|
| 1. $(p \vee q) \Rightarrow (r \wedge \neg s)$ | Premise |
| 2. $\neg r \Rightarrow s$ | Premise |
| 3. $p \vee q$ | Premise |
| 4. $r \wedge (\neg s)$ | 1,3 Modus Ponens |

First-order Predicate Calculus (13/17)

Summary of Rules of Inference

- T1 Any tautology that appears on the list at the end of the last section can be used as a rule of inference.
- T2 We can add any tautology that appears in the list of tautologies at the end of the last section as a new line in our list of true statements.
- S (Substitution): We can replace any part of a compound statement with a tautologically equivalent statement.
- C (Conjunction): If A and B are any two lines in a proof, then we can add the line AB to the proof.
- P (Premise): We can write down a premise as a line in a proof.

First-order Predicate Calculus (14/17)

Predicates

Upper case Roman letters, plus square brackets:

Examples:

$H[a]$: a is happy

$R[a, b]$: a respects b

$S[a, b, g]$: a sold b to g

$H[a, b, g, d]$: a is happy that b sold g to d

First-order Predicate Calculus (15/17)

Function Signs

Lower case Roman letters, plus parentheses:

Examples:

$m(a)$: the mother of a

$s(a,b)$: the sum of a and b

$s(a,b,g)$: the sum of a , b , g

Variables: lower case Roman letters: z , y , x , ...

First-order Predicate Calculus (16/17)

- Logic is the study of good reasoning - and good reasoning is of considerable importance in many subjects.
- Elementary logic covers certain aspects of logic, which are regarded as fundamental.

In elementary logic, compound terms, such as $x + y$, are not used where the focus is on the part of logic concerned with purely logical rules rather than with rules deriving from mathematical practice, or from some other specific domain. This restriction enables the use of a fairly simple syntax, and to make certain parts of the underlying logic practice fairly efficient.

First-order Predicate Calculus (17/17)

In general, elementary logic can include the following:

sentences S

noun phrases N

predicates $N^k \rightarrow S$

function signs $N^k \rightarrow N$

connectives $S^k \rightarrow S$

quantifiers $V + S \rightarrow S$

description operator $V + S \rightarrow N$

Specification techniques

▶ Algebraic approach

- ▶ The system is specified in terms of its operations and their relationships

▶ Model-based approach

- ▶ The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state.

Also referred as Model checking. Model checking can be viewed as exhaustive simulation, i.e. exhaustively exploring the system state space to prove certain correctness properties

Formal specification Languages

	Sequential	Concurrent
Algebraic	Larch (Gutting et al., 1993) OBJ (Futatsugi et al., 1985)	Lotos (Bolognesi and Brinksma, 1987) FSP (Finite State Process)
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) CCS (Milner, 1980) Petri Nets (Peterson, 1981) LTS (Magee and Kramer, 2006) π -ADL (Flavio et al., 2004) UPPAAL

CSP (Communicating Sequential Processes)

CCS (Calculus of Communicating Systems)

Use of formal specification

- ▶ Formal specification involves investing more effort in the early phases of software development
- ▶ This reduces requirements errors as it forces a detailed analysis of the requirements
- ▶ Incompleteness and inconsistencies can be discovered and resolved
- ▶ Hence, savings as made as the amount of rework due to requirements problems is reduced

FSP (Finite State Processes)

- ▶ FSP is a process algebra notation in the form of Finite State Processes used for the concise description of component behavior particularly for the concurrent systems.
- ▶ It provides the means to formalize the specification of software components and architecture.
- ▶ Each component consists of processes and each process has a finite number of states and is composed of one or more actions.

FSP (Finite State Processes)

- ▶ FSP has strong artifacts for parallel constructions; therefore it is used in particular for parallel and concurrent systems. Concurrency exists between elementary calculator activities, for which there is a need to manage the interactions, communication, and synchronization between processes

- ▶ **Component - Processes - States - Actions**

Labeled Transition Systems

A labeled transition system $T = (Q, I, A, \delta)$ is given by
a (finite or infinite) set of *states* Q ,
a set $I \subseteq Q$ of *initial states*,
a set A of *actions* (action names), and
a *transition relation* $\delta \subseteq Q \times A \times Q$.

An action $A \in A$ is *enabled* at state $q \in Q$ iff $(q, A, q') \in \delta$ for some $q' \in Q$.

A run of T is a (finite or infinite) sequence $\rho = q_0 \rightarrow A_0 q_1 \rightarrow A_1 q_2 \dots$ where $q_0 \in I$
and $(q_i, A_i, q_{i+1}) \in \delta$ holds for all i .

A state $q \in Q$ is *reachable* iff it appears in some run ρ of T .

Convention. We assume that A contains a special *.stuttering.* action τ with $(q, \tau, q') \in \delta$ iff $q' = q$. Every finite run can then be extended to an infinite run by “infinite stuttering”.

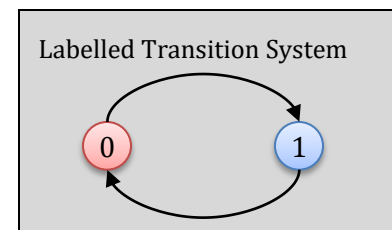
We say that T is *deadlocked* at q if no action except τ is enabled at q .

LTS verification

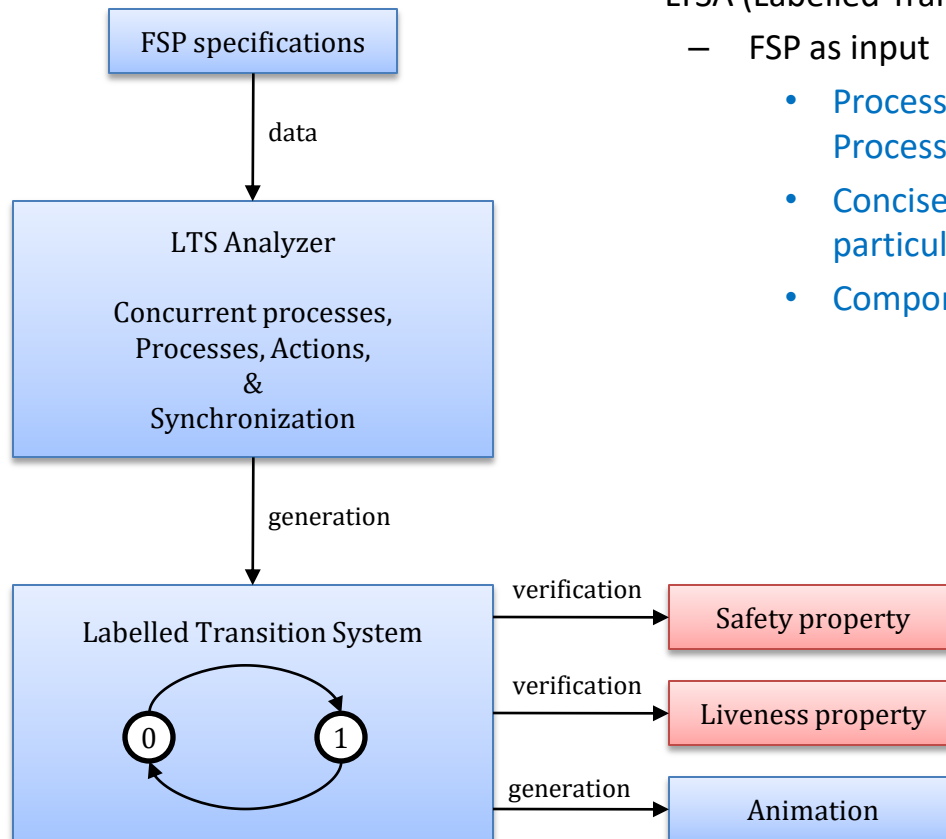
► LTS (Labelled Transition System)

(Magee and Kramer, 2006)

- Formal verification and validation of finite-state concurrent systems
- Compositional reachability analysis
- State machines that have a graphical form to analyze, display and animate behaviour
- Parallel composition of asynchronous processes, interleaving interaction-shared actions, process labelling and action relabeling
- Safety: Property automata



LTS verification



- LTSA (Labelled Transition System Analyzer)
 - FSP as input
 - Process algebra notation in the form of Finite State Processes
 - Concise description of component behaviour particularly for concurrent systems.
 - Component - Processes - States - Actions

Safety & Liveness

▶ Safety property

- Invariant which asserts that “something bad does never happen”.

▶ Liveness property

- It asserts that “something good happens” that describe the states of system that an agent must bring about given certain conditions.

Example: Elevator in a multiple story building

Safety & Liveness

► For example

Consider an elevator moving from one floor to the next floor of a multi-story building. At each floor, if the elevator stops, the door of elevator must open exactly in front of the floor door. In other words the door should not open when the elevator is moving and between two floors. This is a safety property. If the elevator moves between floors without ever opening its doors then its liveness property is satisfied, but it has no safety property. But now the elevator is of no use. So liveness and safety complement each other and both are necessary.

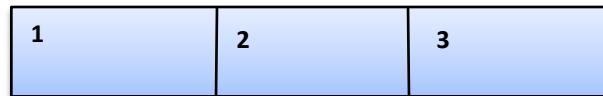
NoLoss: Safety property

```
const N=2      /// Number of carrier agents
const Min=1    /// First road partition / Load road partition
const Max=3    /// Last road partition / Unload road partition

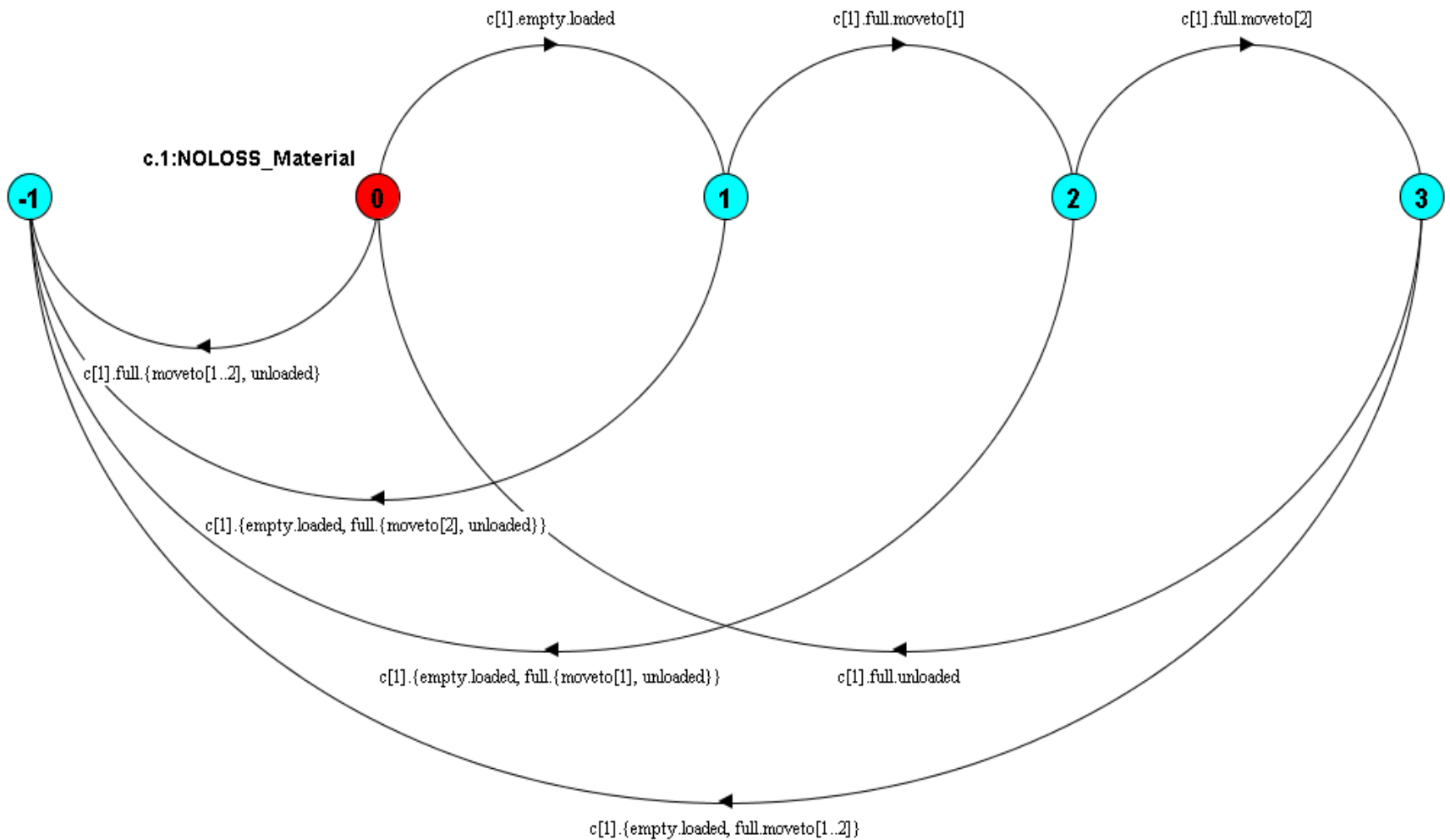
property NOLOSS_Material = (empty.loaded -> ONTHEWAY[1]),

ONTHEWAY[part:Min..Max] = (
    when(part>=Min && part<Max) full.moveto[part] -> ONTHEWAY[part+1]
    | when(part==Max) full.unloaded -> NOLOSS_Material).

||NOLOSS = (c[1..N]:NOLOSS_Material).
```



NoLoss: Safety property



1	2	3
---	---	---

Event-B

- Event-B is a notation for formal modelling based around an abstract machine notation.
- It is formal method that builds the complete model of a system in a rigorous way. It is ideally suited for system-level modelling and analysis.
- It uses set theory as a modelling notation; refinement to represent systems at different abstraction levels; and use of mathematical proof to verify consistency between refinement levels.

Event-B

- Event-B performs a simulation, by constructing a ***mathematical model*** which will be analyzed by doing ***proofs***.
- It involves modelling and formal reasoning.
- Initial model of a program describes the properties that the program must fulfil. Modeling is accompanied by reasoning. Model of a program also contains proofs that are related to the properties of the program.
- Event-B is an evolution of B. Incorporating formal methods in the analysis and design of big-data models is a challenge.

Event-B

- Distributed systems are complex systems, in that they are made of many parts interacting with a highly evolving and sometimes hostile environment. They require a high degree of correctness.
- Complex models can be written when the method of step-wise refinement is used i.e. a model is built by successive enhancement of an original simple “sketch” carefully transforming it into more concrete representations.

Event-B

An Event-B model consists of *contexts* and *machines*

- ***Contexts*** contain the static parts of the model. These are *constants* and *axioms* that describe the properties of these constants.
- ***Machines*** contain the dynamic parts of the model. A machine is made of a *state*, which is defined by means of *variables*. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers etc. They are constrained by *invariants* $I(v)$ where v are the variables of the machine. Invariants are supposed to hold whenever variable values change.

Event-B

- In addition to state, a machine contains a number of *events* which specify how the state may evolve. Each ***event*** is composed of a *guard* and an *action*.
- The ***guard*** is the necessary condition under which the event may occur.
- The ***action***, determines the way in which the state variables are going to evolve when the event occurs.

Event-B

- An event may have parameters that are local to that event. Parameters can serve different functions, for instance, to model arrays of events or as communication channels in composition of machines.
- An event may be executed only when its guard holds. Events are atomic and when the guards of several events hold simultaneously, then at most one of them may be executed at any one moment. The choice of event to be executed is non-deterministic

Event-B

- Event-B can be used for the construction of ***complex discrete systems***. Event-B models system in a discrete fashion.
- The behavior of most of the software models is ultimately continuous, the systems operate most of the time in a *discrete fashion*. This means that their behavior can be ***abstracted*** by a succession of steady states intermixed with jumps that cause sudden state changes.

Event-B

- The number of such possible changes is enormous, and they are occurring in a concurrent fashion at an unthinkable frequency.
- But this number and this high frequency do not change the very nature of the problem: such systems are intrinsically discrete. They fall under the generic name of ***transition systems***.

Event-B

State and transitions

- A discrete model is made of a state represented by some constants and variables at a certain level of abstraction with regard to the real system under study. Besides the state, the model also contains a number of transitions that can occur under certain circumstances. Such transitions are called here “events.”

Event-B

State and transitions

- Each event is first made of a guard, which is a predicate built on the state constants and variables. It represents the necessary conditions for the event to occur. Each event is also made up of an action, which describes the way certain state variables are modified as a consequence of the event occurrence.

Event-B

Operational interpretation

- As can be seen, a discrete dynamical model thus indeed constitutes a kind of state transition machine. We can give such a machine an extremely simple operational interpretation.
- First of all, the execution of an event, which describes a certain observable transition of the state variables, is considered to take no time. Moreover, no two events can occur simultaneously. The execution is then the following:
- When no event guards are true, then the model execution stops; it is said to have deadlocked.
- When some event guards are true, then one of the corresponding events necessarily occurs and the state is modified accordingly; subsequently, the guards are checked again, and so on.

Event-B

Operational interpretation

- This behavior clearly shows some possible non-determinism (called external nondeterminism) as several guards may be true simultaneously.
- We make no assumption concerning the specific event which is indeed executed among those whose guards are true. When only one guard is true at all times, the model is said to be deterministic.
- The fact that a model eventually finishes is not at all mandatory. Most of the systems we study never deadlock; they run for ever. (Abrial, Modeling in Event-B System and Software Engineering, 2010)

Event-B

Formal reasoning

- There are two kinds of discrete model properties.
- The first kind of properties to prove about models, and hence ultimately about real systems, are ***invariant properties***. An invariant is a condition on the state variables that must hold permanently. *In order to achieve this, it is just required to prove that, under the invariant in question and under the guard of each event, the invariant still holds after being modified according to the action associated with that event.*
- Consider more complicated forms of reasoning, involving conditions which, in contrast to the invariants, do not hold permanently. The corresponding statements are called modalities. In our approach we consider a very special form of modality, called reachability. Reachability property we would like to prove that *an event whose guard is not necessarily true now will nevertheless certainly occur within a certain finite time.*

Event-B

Managing the complexity of closed models

- The models built in Event-B will not just describe the control part of our intended system, they will also contain a certain representation of the environment within which the system is supposed to behave. In fact, we shall quite often essentially construct closed models, which are able to exhibit the actions and reactions taking place between a certain environment and a corresponding, possibly distributed, controller.
- In doing so, we shall be able to insert the model of the controller within an abstraction of its environment, which is formalized as yet another model. The state of such a closed system thus contains physical variables, describing the environment state, as well as logical variables, describing the controller state.
- And, in the same way, the transitions will fall into two groups: those concerned with the environment and those concerned with the controller. We shall also have to put into the model the way these two entities communicate.

Event-B

- The number of transitions in the real systems under study is enormous. The number of variables describing the state of such systems is also extremely large. How to practically manage such complexity? The answer to this question lies in three concepts: ***refinement***, ***decomposition***, and ***generic instantiation***.
- These concepts are linked together. A model is refined to later decompose it, and, more importantly, we decompose it to further refine it more freely. And, finally, a generic model development can be later instantiated, thus saving the user from redoing similar proofs.

Event-B

1. Refinement

- Refinement allows us to build a model *gradually* by making it more and more precise, so that it is closer to the reality. In other words, we are not going to build a single model representing once and for all our reality; this is clearly impossible due to the size of the state and the number of its transitions.
- It would also make the resulting model very difficult to master.
- We are rather going to construct an ordered sequence of embedded models, where each of them is supposed to be a refinement of the one preceding it in a sequence. This means that a refined, more concrete, model will have more variables than its abstraction; such new variables are a visible consequence of a view of our system from closer range.

Event-B

- A refined model is spatially larger than its previous abstractions.
- Analogously to this *spatial extension*, there is a corresponding *temporal extension*. This is because the new variables are now able to be modified by some transitions, which could not have been present in the previous abstractions, simply because the concerned variables did not exist in them. Practically, this is realized by means of *new events*, involving the new variables only. Such new events refine some implicit events doing nothing to the abstraction. Refinement will thus result in a discrete observation of our reality, which is now performed using a *finer time granularity*.
- Refinement is also used in order to modify the state so that it can be implemented on a computer by means of some programming language. This second usage of refinement is called *data refinement*. It is used as a second technique, once all the important properties have been modeled.

Event-B

2. Decomposition

- Refinement does not solve completely the problem of the complexity. As a model is more and more refined, the number of its state variables and that of its transitions may augment in such a way that it becomes impossible to manage them as a whole. At this point, it is necessary to cut our single refined model into several almost independent pieces.
- Decomposition is precisely the process by which a single model can be split into various component models in a systematic fashion. In doing so, we reduce the complexity of the whole by studying, and thus refining, each component model independently of the others. Such a decomposition implies that independent refinements of the component models could always be put together again to form a single model that is guaranteed to be a refinement of the original one. This decomposition process can be further applied to the components, and so on.

Event-B

3. *Generic development*

- Any model development done by applying refinement and decomposition is parameterized by some carrier sets and constants defined by means of a number of properties. Such a generic model may be instantiated within another development in the same way as a mathematical theory, e.g. group theory, can be instantiated in a more specific mathematical theory. This can be done providing that we have been able to prove that the axioms of the abstract theory are mere theorems in the second one. The interest of this approach of generic instantiation is that it saves us redoing the proofs already done in the abstract development.

Rodin

- Rodin (Abrial, et al., November 2010) (Jastram & Butler, 2014) platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. It integrates modelling and proving. It is open source, and is extensible with plugins. Rodin has made a large contribution in making Theorem proving a practical tool for software engineering.

Rodin

- In order for formal modelling to be used safely and effectively in engineering practice, good tool support is of critical importance.
- In Rodin, there is no need for the user to start processes like compilation i.e. a program is written and then run or debugged without compiling it.
- Rodin tool for Event-B (Abrial, 2010) provides techniques used in programming, formal modelling to formal verification.
- Instead of compilation, Rodin is interested in proof obligation generation and automatically discharging trivial proof obligations.
- Instead of running a program our approach in Rodin is to reason about models and analyze them.

ProB (Leuschel & Butler, 2003)

- The ProB (Leuschel & Butler, 2003) is an animator and model checker.
- The ProB tool supports automated consistency checking of B machines via model checking.
- For exhaustive model checking, the given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine.

ProB (Leuschel & Butler, 2003)

- ProB explores the state space non-exhaustively and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage.
- ProB generates and graphically display counter-examples when it discovers a violation of the invariant. ProB can also be used as an animator of a B specification.
- So, the model checking facilities are still useful for infinite state machines, not as a verification tool, but as a sophisticated debugging and testing tool. ProB is available as a plug-in for Rodin.

UML-B (Snook & Butler, 2006)

- The UML-B (Snook & Butler, 2006) is a profile of UML that defines a formal modelling notation. It has a mapping to the Event-B language.
- UML-B consists of class diagrams with attached state charts, and an integrated constraint and action language, called UML-B, based on the Event-B notation.
- It provides a diagrammatic, formal modelling notation based on UML.
- Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations.

UML-B (Snook & Butler, 2006)

- The UML-B (Snook & Butler, 2006) plug-in converts UML-B models into Event-B models.
- Translation from UML-B into Event-B enables the Rodin proof obligation generator and provers to be utilized.
- Since the Event-B language is not object-oriented, class instances must be modelled explicitly in the generated Event-B. Attributes and associations are represented as variables whose type is a function from the class instances to the attribute type or associated class.
- Operation behavior may be represented textually in Event-B, as a state chart attached to the class, or as a simultaneous combination of both.

Key points

- ▶ Formal system specification complements informal specification techniques
- ▶ Formal specifications are precise and unambiguous, They remove areas of doubt in a specification
- ▶ Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system

Key points

- ▶ Formal specification techniques are most applicable in the development of critical systems and standards.
- ▶ Model-based techniques model the system using sets and functions. Operations are defined by modifications to the system's state.

Reference Material

Fundamentals of Software Engineering (Second Edition)

Authors: Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli

Modern Formal Methods and Applications

Authors: Hossam A. Gabbar, Springer-Verlag 2006

Software Engineering: A Practitioner's Approach - 7th International edition (2009)

Author: [Roger Pressman](#)

Publisher: McGraw-Hill 7th edition (2009)

Reference Material

Software Reliability Methods, Doron A. Peled, 2001
Springer-Verlag

Logic in Computer Science, Modelling and Reasoning about Systems, 2nd Edition, Michael Huth, Imperial College of Science, Technology and Medicine, London Mark Ryan, University of Birmingham, 2004

Principles of Model Checking, Christel Baier and Joost-Pieter Katoen, MIT Press, 2008.