



Software Engineering and Formal Specification

CSIT - 31012

Dr. Nadeem Akhtar

Assistant Professor

Dept. of Computer Science & IT

The Islamia University of Bahawalpur

System Validation (1/12)

Safety Critical Systems

- *Mobile telephone systems*, are used in various circumstances, as in ambulances, where malfunctioning of the software can have disastrous consequences.
- *Information processing* plays a significant role in process control systems, as in nuclear power plants or in chemical industry where, errors in software are highly undesirable.

System Validation (2/12)

- Radiation machines in hospitals
- Storm surge barriers

These are examples where **Reliability** is a key issue and where the **Correctness** of a system is of vital importance

System Validation (3/12)

- “Non-information processing” systems, such as *electric razors, audio equipment, and TV-sets* the amount of software and hardware is increasing rapidly.
- In transport systems (i.e. cars, trains and airplanes) information technology is coming to play a dominant role; expectations are that about 20 percent of the total development costs of those vehicles will be needed for information technology.

System Validation (4/12)

- This increasing integration of information processing into various kinds of (critical) applications it is fair to state that:

The reliability of information processing is a key issue in the system design process

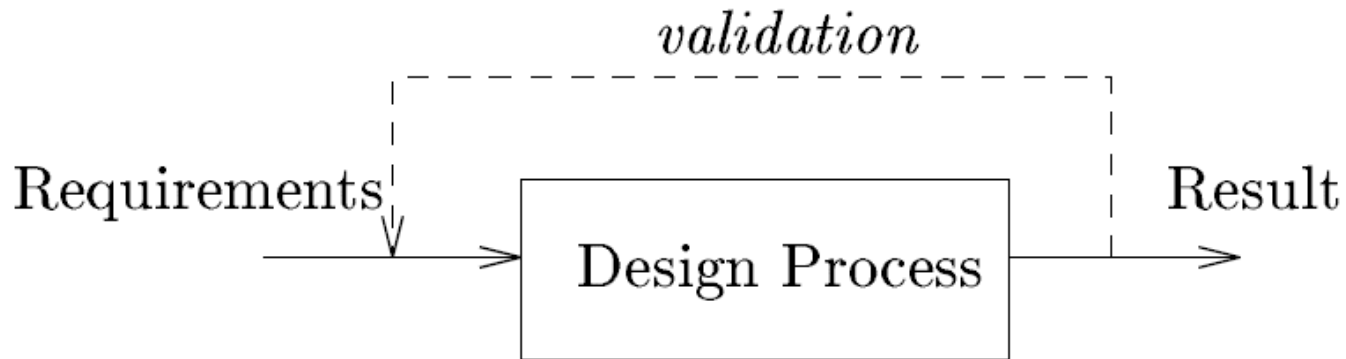
System Validation (5/12)

What is common practice on maintaining the reliability of software systems

- The design is started with a **requirements analysis** where the desires of the client's are extensively analyzed and defined.
- After traversing several **distinct design phases**, at the end of the design process a (prototype) design is obtained.

The process of establishing that a design fulfills its requirements is referred to as *System Validation*

System Validation (6/12)

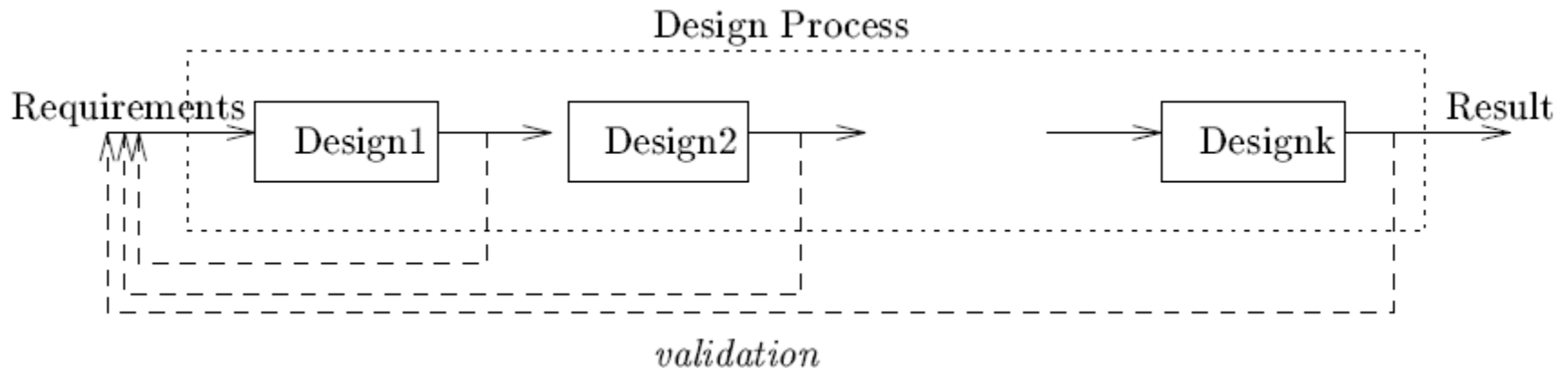


Checking for faults only at the end of the design trajectory is not acceptable:

If an error is found it takes too much effort to repair it since the whole design trajectory needs to be re-traversed in order to see where and how the error could arise.

This is costly

System Validation (7/12)



Schematic view of on-the-fly system validation

It is therefore more common practice to check on-the-fly i.e. while the system is being designed.

If an error is determined in this setting ideally only the design phase up to the previous validation step needs to be re-examined.

This reduces the costs significantly

System Validation (8/12)

- Two techniques are widely applied in practice to ensure that the final result does what it originally is supposed to do:
 - *Peer reviewing* &
 - *Testing*

System Validation (9/12)

Investigations in the software engineering show that 80 % of all projects use Peer reviewing

Peer reviewing is a completely manual activity in which (part of) a design is reviewed by a team of developers (that in order to increase the effect of reviewing) have not been involved in the design process of that particular part

System Validation (10/12)

Testing is an **operational** way to check whether a given system realization conforms to the original abstract specification

Testing is an essential validation technique for checking the correctness of real implementations, it is usually applied in an ad-hoc manner - tests are, for instance, generated by hand.

System Validation (11/12)

In most designs more time and effort is spent on validation than on construction.

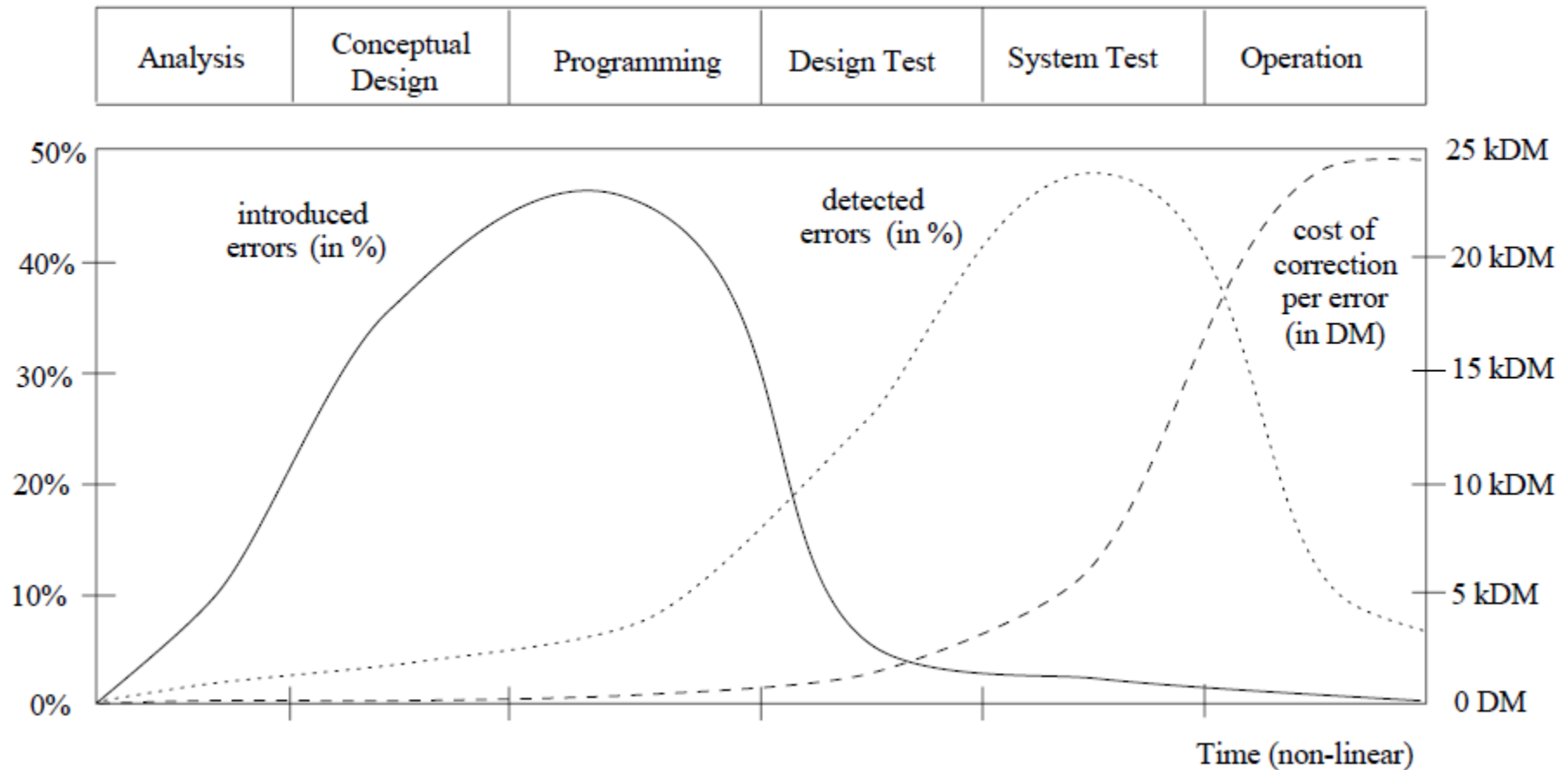
System Validation (12/12)

- *Errors can be very expensive.*
- The error in **Intel's Pentium floating-point division unit** is estimated to have caused a loss of about 500 million US dollars.
- A mistake in the **missile Ariane-5** (costs of about 500 million).
- The mistake in **Denver's baggage handling system** that postponed the opening of the new airport for 9 months (at 1.1 million dollar per day).

System validation as part of the design process

- In Germany interviews with several **software engineering companies** have resulted in some interesting figures for the costs of error-repair and statements about the stages in the design process where errors are introduced and detected
- These investigations show once more that *integration of system validation into the design process should take place at an early stage* particularly from an **economical point of view**.

System life cycle and error introduction, detection and costs of repair



Validation by Formal methods (1/5)

There is a strong need for an early integration of system validation in the design process

Validation by Formal methods (2/5)

- Computer systems are composed of several subsystems such as parallel and distributed systems
 - Computer/telephony networks
 - Chips that consist of many communicating components
 - High-speed parallel computers connected by shared memory

Validation by Formal methods (3/5)

- Due to the increase in the magnitude and complexity of information processing systems, errors can easily and unexpectedly occur in their design. Reasoning about such systems becomes more and more complicated.
- There is a strong need for the use of advanced systematic techniques and tools that support the system validation process

Validation by Formal methods (4/5)

- Using formal methods system designs can be defined in terms of **precise** and **unambiguous** specifications that provide the basis for a **systematic analysis**.
- Important validation techniques based on formal methods are:
 - Testing
 - Simulation
 - Formal verification
 - Model checking

Validation by Formal methods (5/5)

- System validation based on formal methods has a great potential and is well-accepted in fields like:
 - Hardware verification
 - Verification of communication protocols
 - Aeronautical software
 - Space-craft software
- Formal methods use in software engineering is still limited

Validation of Reactive Systems (1/7)

- **Reactive systems** are characterized by a *continuous interaction* with their environment.
 - They continuously receive inputs from their environment and usually within quite a short delay react on these inputs

Validation of Reactive Systems (2/7)

- Examples: (Reactive systems)
 - Operating systems
 - Aircraft control systems
 - Communication protocols
 - Process control software.

Validation of Reactive Systems (3/7)

Example:

A **control program** of a chemical process receives control signals regularly like **temperature and pressure**, at several points in the process, Based on this information, the program can decide to turn on the heating elements, to switch on a pump, and so forth.

As soon as a dangerous situation is anticipated. for example the pressure in the tank exceeds certain thresholds, the control software needs to take appropriate action.

Validation of Reactive Systems (4/7)

Usually reactive systems are rather complex, the nature of their interaction with the environment can be intricate and they typically have a distributed and concurrent nature.

Validation of Reactive Systems (5/7)

- **Correctness** and **well-functioning** of reactive systems is crucial. To obtain correctly functioning and dependable reactive systems a coherent and **well-defined methodology** is needed in which different phases can be distinguished.
 1. In the **first phase**, a thorough investigation of requirements is needed and as a result a requirement specification is obtained.

Validation of Reactive Systems (6/7)

2. Secondly, a conceptual design phase results in an abstract design specification. This specification needs to be validated for internal consistency and it needs to be checked against the requirement specification. This validation process can be supported by formal verification, simulation and model checking techniques, where a model (like a finite state automaton) of the abstract design specification can be extensively checked.

Validation of Reactive Systems (7/7)

3. Once a trustworthy specification has been obtained, *the third phase* consists of building a system that implements the abstract specification,
 - *Testing is a useful technique to support the validation of the realization versus the original requirement specification.*

Simulation (1/2)

- *Simulation is based on a model that describes the possible behavior of the system design at hand.* This model is executable by software tool called **Simulator**.
 - A **Simulator** can determine the system's behavior on the basis of some scenarios. In this way the user gets some insight on the reactions of the system on certain stimuli.

Simulation (2/2)

- The scenarios can be either provided by the user or by a tool, like a *random generator* that provides random scenarios.
 - Simulation is typically useful for a *quick, first assessment of the quality of a design*. It is less suited to find subtle errors
 - It is infeasible, and mostly even impossible, to simulate all representative scenarios.

Testing (1/11)

- A widely applied and traditional way of validating the correctness of a design is by means of testing.
 - In testing one takes the implementation of the system as realized (*a piece of software, hardware, or a combination*), stimulates it with certain (*well-chosen*) inputs, called tests, and observes the reaction of the system.
 - *Finally, it is checked whether the reaction of the system conforms to the required output*

Testing (2/11)

- *The principle of testing is almost the same as that of simulation, the important distinction being that*
 - *testing is performed on the real, executing, system implementation, whereas*
 - *simulation is based on a model of the system*

Testing (3/11)

- Testing is a validation technique that is most used in practice in software engineering, for instance, in all projects some form of testing is used but almost exclusively *on the basis of informal and heuristic methods*.
- *Since testing is based on observing only a small subset of all possible instances of system behavior it can never be complete*

Testing (4/11)

- Testing can only show the presence of errors never their absence
- Testing is complementary to formal verification and model checking.
- Formal verification and Model checking are based on a mathematical model of the system rather than on a real executing system.

Testing (5/11)

- Since testing can be applied to the real implementation it is useful in most cases:
 - where a **valid and reliable model of the system is difficult to build due to complexity,**
 - where system **parts cannot be formally modeled** (as physical devices) or
 - where the model is **proprietary** (e.g. in case of third-party testing)

Testing (6/11)

- *Testing is the dominating technique for system validation and consists of applying a number of tests that have been obtained in an **ad-hoc** or **heuristic** manner to an implementation*
- *Recently, the interest and possibilities of applying formal methods to testing are increasing.*
- For instance, in the area of communication protocols: International Standard on "*formal methods in conformance testing*" (ISO, 1996)

Testing (7/11)

Formal methods in conformance testing (ISO, 1996) the process of testing is partitioned into several phases:

1. In the *test generation phase*, abstract descriptions of tests, so-called test cases, are systematically generated starting from a *precise and unambiguous specification of the required properties in the specification*. The test cases must be guaranteed to test these properties.

Testing (8/11)

2. In the *test selection phase*, a representative set of abstract test cases is selected.
3. In the *test implementation phase*, the abstract test cases are transformed into executable test cases by compiling them or implementing them.

Testing (9/11)

4. In the *test execution phase*, the executable test cases are applied to the implementation under test by executing them on a test execution system. The results of the test execution are observed and logged
5. In the *test analysis phase*, the logged results are analyzed to decide whether they comply with the expected results.

Testing (10/11)

- Testing is a technique that can be applied both to prototypes, in the form of *systematic simulation*, and to *final products*.
- *Transparent box testing* where the internal structure of an implementation can be observed and sometimes partially controlled (i.e. stimulated)

Testing (11/11)

- Black box testing where only the communication between the system under test and its environment can be tested and where the internal structure is completely hidden to the tester.
- Grey box testing (in practical circumstances) is mostly somewhere in between these two extremes (i.e. transparent and black).

Formal verification (1/5)

- A complementary technique to simulation and testing is to prove that a system *operates correctly*, The term for this mathematical demonstration of the correctness of a system is formal verification
- The basic idea is to construct a *formal (i.e. mathematical)* model of the system under investigation which represents the possible behavior of the system.

Formal verification (2/5)

- In addition, the correctness requirements are written in a formal *requirement specification* that represents the desirable behavior of the system.

Based on these two specifications one checks by formal proof whether the possible behavior *“agrees with”* the desired behavior.

Formal verification (3/5)

Since the verification is treated in a mathematical fashion, the notion of “agree with” can be made *precise*, and *verification amounts to proving or disproving the correctness with respect to this formal correctness notion.*

Formal verification (4/5)

Formal verification requires:

1. A **model of the system** consisting of
 - a set of *states*, incorporating information about values of variables, program counters and
 - a *transition relation* that describes how the system can change from one state to another.

Formal verification (5/5)

2. A **specification method** for expressing requirements in a formal way.
3. A **set of proof rules** to determine whether the model satisfies the stated requirements.

Verifying Sequential Programs (1/13)

- This approach can be used to prove the correctness of sequential algorithms such as quick-sort or the computation of the greatest common divisor of two integers.

One starts by formalizing the desired behavior by using pre- and post-conditions, formulas in predicate logic. The syntax of such formulas is, for instance, defined by

$$\phi ::= p \mid \neg \phi \mid \phi \wedge \phi$$

Verifying Sequential Programs (2/13)

$$\phi ::= p \mid \neg \phi \mid \phi \wedge \phi$$

where p is a basic statement (like “ x equals 2”),

\neg denotes negation, \vee denotes disjunction.

$$\phi \vee \psi = \neg(\neg \phi \wedge \neg \psi),$$

$$\text{true} = \phi \vee \neg \phi,$$

$$\text{false} = \neg \text{true}$$

$$\phi \Rightarrow \psi = \neg \phi \vee \psi.$$

For simplicity we omit universal and existential quantification.

Verifying Sequential Programs (3/13)

- A *pre-condition* describes the set of interesting start states (i.e. the allowed input(s)), and the *post-condition* describes the set of desired final states (i.e. the required output(s)).
- Once the *pre-* and *post-condition* are formalized, the algorithm is coded in some *abstract pseudo-code language* and it is proven in a step-by-step fashion that the program satisfies its specification.

Verifying Sequential Programs (4/13)

To construct the proofs, a proof system, that is, a set of proof rules, is used. These proof rules usually correspond to program constructs. They are written:

$$\{ \phi \} S \{ \psi \}$$

where ϕ is a precondition, S a program statement, and ψ a postcondition. The triple $\{ \phi \} S \{ \psi \}$ is known as the Hoare triple (Hoare 1969), christened to one of the pioneers in the field of formal verification of computer programs. There are two possible interpretations of Hoare triples, depending on whether one considers partial or total correctness.

Verifying Sequential Programs (5/13)

- The formula $\{ \phi \} S \{ \psi \}$ is called *partially correct* if any terminating computation of S that starts in a state satisfying ϕ , terminates in a state satisfying ψ .
- $\{ \phi \} S \{ \psi \}$ is called *totally correct* if any computation of S that starts in a state satisfying ϕ , terminates and finishes in a state satisfying ψ .

So, in the case of partial correctness no statements are made about computations of S that diverge, that is not terminates.

Verifying Sequential Programs (6/13)

The basic idea of the approach by Hoare (and others) is to prove the correctness of programs at a syntactical level, only using triples of the above form. To illustrate his approach we will briefly treat a proof system for a simple set of deterministic sequential programs. A program is called deterministic if it always provides the same result when provided a given input. These sequential programs are constructed according to the following grammar:

$$S ::= \text{skip} \mid x := E \mid S ; S \mid \text{if } B \text{ then } S \text{ else } S \text{ fi} \mid \text{while } B \text{ do } S \text{ od}$$

Verifying Sequential Programs (7/13)

$$S ::= \text{skip} \mid x := E \mid S ; S \mid \text{if } B \text{ then } S \text{ else } S \text{ fi} \mid \text{while } B \text{ do } S \text{ od}$$

where **skip** stands for no operation, $x := E$ for the assignment of the value of expression E to variable x (where x and E are assumed to be equally typed), $S ; S$ for the sequential composition of statements, and the latter two for alternative composition and iteration (where B denotes a boolean expression), respectively.

The proof rules should be read as follows: if all conditions indicated above the straight line are valid, then the conclusion below the line is valid. For rules with a condition true only the conclusion is indicated; these proof rules are called axioms. A proof system for sequential deterministic programs is given in Table

Proof system for partial correctness of sequential programs

Axiom for **skip** $\{ \phi \} \text{ skip } \{ \phi \}$

Axiom for assignment $\{ \phi[x := k] \} x := k \{ \phi \}$

Sequential composition
$$\frac{\{ \phi \} S_1 \{ \chi \} \wedge \{ \chi \} S_2 \{ \psi \}}{\{ \phi \} S_1 ; S_2 \{ \psi \}}$$

Alternative
$$\frac{\{ \phi \wedge B \} S_1 \{ \psi \} \wedge \{ \phi \wedge \neg B \} S_2 \{ \psi \}}{\{ \phi \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{ \psi \}}$$

Iteration
$$\frac{\{ \phi \wedge B \} S \{ \phi \}}{\{ \phi \} \text{ while } B \text{ do } S \text{ od } \{ \phi \wedge \neg B \}}$$

Consequence
$$\frac{\phi \Rightarrow \phi', \{ \phi' \} S \{ \psi' \}, \psi' \Rightarrow \psi}{\{ \phi \} S \{ \psi \}}$$

Verifying Sequential Programs (8/13)

Axiom for **skip**

$$\{ \phi \} \text{ skip } \{ \phi \}$$

Axiom for assignment

$$\{ \phi[x := k] \} x := k \{ \phi \}$$

The proof rule for the skip-statement, the statement that does nothing, is what one would expect: under any condition, if ϕ is valid before the statement, then it is valid afterwards. According to the axiom for assignment, one starts with the postcondition ϕ and determines by substitution the precondition $\phi[x := k]$. $\phi[x := k]$ roughly means ϕ where all occurrences of x are replaced by k , e.g.

$$\{ k^2 \text{ is even and } k = y \} x := k \{ x^2 \text{ is even and } x = y \}.$$

The procedure of starting the proof from a postcondition is usually applied successively to parts of the program in a way such that finally the precondition of the entire program can be proved.

Verifying Sequential Programs (9/13)

Sequential composition	$\frac{\{\phi\} S_1 \{\chi\} \wedge \{\chi\} S_2 \{\psi\}}{\{\phi\} S_1 ; S_2 \{\psi\}}$
Alternative	$\frac{\{\phi \wedge B\} S_1 \{\psi\} \wedge \{\phi \wedge \neg B\} S_2 \{\psi\}}{\{\phi\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{\psi\}}$

The rule for sequential composition uses an intermediate predicate χ that characterizes the final state of S_1 and the starting state of S_2 . The rule for alternative composition uses the boolean B whose value determines whether S_1 or S_2 is executed.

Greek Symbol: χ (Kai – ki)

Verifying Sequential Programs (10/13)

Iteration

$$\frac{\{ \phi \wedge B \} S \{ \phi \}}{\{ \phi \} \textbf{while } B \textbf{ do } S \textbf{ od } \{ \phi \wedge \neg B \}}$$

Stated in words, this rule states that predicate ϕ holds after the termination of **while** B **do** S **od** if the validity of ϕ can be maintained during each execution of the iteration-body S . This explains why ϕ is called an invariant. One of the main difficulties in the proof of programs in this approach is to find appropriate invariants.

Verifying Sequential Programs (11/13)

We must stress that the above proof system allows the establishment of the relation between inputs and outputs of composed programs by only considering such relations for program parts. Proof rules and proof systems that exhibit this property are called *compositional*. For instance, the proof rule for sequential composition allows the correctness of the composed program $S_1 ; S_2$ to be established by considering the pre- and postconditions of its components S_1 and S_2 .

Verifying Sequential Programs (12/13)

Let us briefly consider total correctness. The proof system in Table 1. is insufficient to prove termination of sequential programs. The only syntactical construct that can possibly lead to divergent (i.e. non-terminating) computations is iteration. In order to prove termination, the idea is, therefore, to strengthen the proof rule for iteration by:

$$\frac{\{\phi \wedge B\} S \{\phi\}, \{\phi \wedge B \wedge n = N\} S \{n < N\}, \phi \Rightarrow n \geq 0}{\{\phi\} \text{ while } B \text{ do } S \text{ od } \{\phi \wedge \neg B\}}$$

Verifying Sequential Programs (13/13)

$$\frac{\{\phi \wedge B\} S \{\phi\}, \{\phi \wedge B \wedge n = N\} S \{n < N\}, \phi \Rightarrow n \geq 0}{\{\phi\} \textbf{while } B \textbf{ do } S \textbf{ od } \{\phi \wedge \neg B\}}$$

Here, the auxiliary variable N does not occur in ϕ , B , n , or S . The idea is that N is the initial value of n , and that at each iteration the value of n decreases (but remains positive). This construction precisely avoids infinite computations, since n cannot be decreased infinitely often without violating the condition $n \geq 0$. The variable n is known as the variant function.

Formal verification of Parallel Systems (1/5)

For statements S_1 and S_2 let the construct

$$S_1 \parallel S_2$$

denote the parallel composition of these statements. The major aim of applying formal verification to parallel programs is to obtain a proof rule such as:

$$\frac{\{ \phi \} S_1 \{ \psi \}, \{ \phi' \} S_2 \{ \psi' \}}{\{ \phi \wedge \phi' \} S_1 \parallel S_2 \{ \psi \wedge \psi' \}}$$

Formal verification of Parallel Systems (2/5)

The introduction of parallelism inherently leads to the introduction of *non-determinism*. This results in the fact that for parallel programs which interact using shared variables the input-output behavior strongly depends on the order in which these common variables are accessed. For instance, if S_1 is $x := x+2$, S'_1 is $x := x+1 ; x := x+1$, and S_2 is $x := 0$, the value of x at termination of $S_1 \parallel S_2$ can be either 0 or 2, and the value of x at termination of $S'_1 \parallel S_2$ can be 0, 1 or 2. The different outcomes for x depend on the order of execution of the statements in S_1 and S_2 , or S'_1 and S_2 . Moreover, although the input-output behavior of S_1 and S'_1 is obviously the same (increasing x by 2), there is no guarantee that this is true in an identical parallel context.

Formal verification of Parallel Systems (3/5)

Concurrent processes can potentially interact at any point of their execution, not only at the beginning or end of their computation. In order to infer how parallel programs interact, it is not sufficient to know properties of their initial and final states. Instead it is important to be able to make, in addition, statements about what happens *during* computation. So properties should not only refer to start and end-states, but the expression of properties about the executions themselves is needed.

Formal verification of Parallel Systems (4/5)

The main problem of the classical approach for the verification of parallel and reactive systems as explained before is that it is completely focused on the idea that a program (system) computes a function from inputs to outputs. That is, given certain allowed input(s), certain desired output(s) are produced. For parallel systems the computation usually does not terminate, and correctness refers to the behavior of the system in time, not only to the final result of a computation (if a computation ends at all). Typically, the global property of a parallel program can often not be stated in terms of an input-output relationship.

Formal verification of Parallel Systems (5/5)

The use of so-called *proof assistants*, software tools that assist the user in obtaining mathematical proofs, and *theorem provers*, tools that generate the proof of a certain theorem, may overcome these problems to some extent; their use in formal verification is a research topic that currently attracts a strong interest.

Temporal Logic (1/7)

The correctness of a reactive system refers to the behavior of the system over time, it does not refer only to the input-output relationship of a computation (as pre- and postconditions do), since usually computations of reactive systems do not terminate.

Temporal Logic (2/7)

Consider, for instance, a communication protocol between two entities, a sender P and a receiver, which are connected via some bidirectional communication means. A property like

“if process P sends a message, then it will not send the next message until it receives an acknowledgement”

cannot be formulated in a pre- and postcondition way.

Temporal Logic (3/7)

To facilitate the formal specification of these type of properties, propositional logic has been extended by some operators that refer to the behavior of a system over time. \mathbf{U} (until) and \mathbf{G} (globally) are examples of operators that refer to sequences of states (as executions). $\phi \mathbf{U} \psi$ means that property ϕ holds in all states until a state is reached in which ψ holds, and $\mathbf{G} \phi$ means that always, that is in all future states, ϕ holds. Using these operators we can formalize the above statement for the protocol by, for instance,

$$\mathbf{G} [\text{snd}_P(m) \Rightarrow \neg (\text{snd}_P(\text{nxt}(m)) \mathbf{U} \text{rcv}_P(\text{ack}))].$$

Temporal Logic (4/7)

$$\mathbf{G}[\text{snd}_P(m) \Rightarrow \neg(\text{snd}_P(\text{nxt}(m)) \mathbf{U} \text{rcv}_P(\text{ack}))].$$

Stated in words, if a message m is sent by process P , then there will be no transmission by P of some next message ($\text{nxt}(m)$) until an acknowledgement has been received by P .

Temporal Logic (5/7)

Logics extended by operators that allow the expression of properties about executions, in particular those that can express properties about the relative order between events, are called *temporal logics*.

Such logics are rather well-established and have their origins in other fields many decades ago. The introduction of these logics into computer science is due to Pnueli (1977).

Temporal logic is a well-accepted and commonly used specification technique for expressing properties of computations (of reactive systems) at a rather high level of abstraction.

Temporal Logic (6/7)

In a similar way as in verifying sequential programs one can construct proof rules for temporal logic for reactive systems and prove the correctness of these systems with the same approach as we have seen for sequential programs using predicate logic.

The disadvantages of the proof verification method, for checking parallel systems:

- they are tedious,
- labour-intensive, and
- require a high degree of user guidance.

Temporal Logic (7/7)

An interesting approach for reactive systems that we would like to mention, for which tool support is available, is TLA (Temporal Logic of Actions) of Lamport (1994). This approach allows one to specify requirements and system behavior in the same notation.

Model Checking (1/4)

The basic idea of what is known as *model checking* is to use algorithms, executed by computer tools, to verify the correctness of systems. The user inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The counter-example consists of a scenario in which the model behaves in an undesired way.

Model Checking (2/4)

Thus the counter-example provides evidence that the model is faulty and needs to be revised. This allows the user to locate the error and to repair the model specification before continuing. If no errors are found, the user can refine its model description (e.g. by taking more design decisions into account, so that the model becomes more concrete/realistic) and can restart the verification process.

Model Checking (3/4)

The algorithms for model checking are typically based on an *exhaustive state space search* of the model of the system: for each state of the model it is checked whether it behaves “correctly”, that is, whether the state satisfies the desired property. In its most simple form, this technique is known as *reachability analysis*. E.g., in order to determine whether a system can reach a state in which no further progress is possible (a so-called deadlock), it suffices to determine all reachable states and to determine whether there exists a reachable state from which no further progress can be made. Reachability analysis is only applicable to proving freedom from deadlock and proving invariant properties, properties that hold during an entire computation.

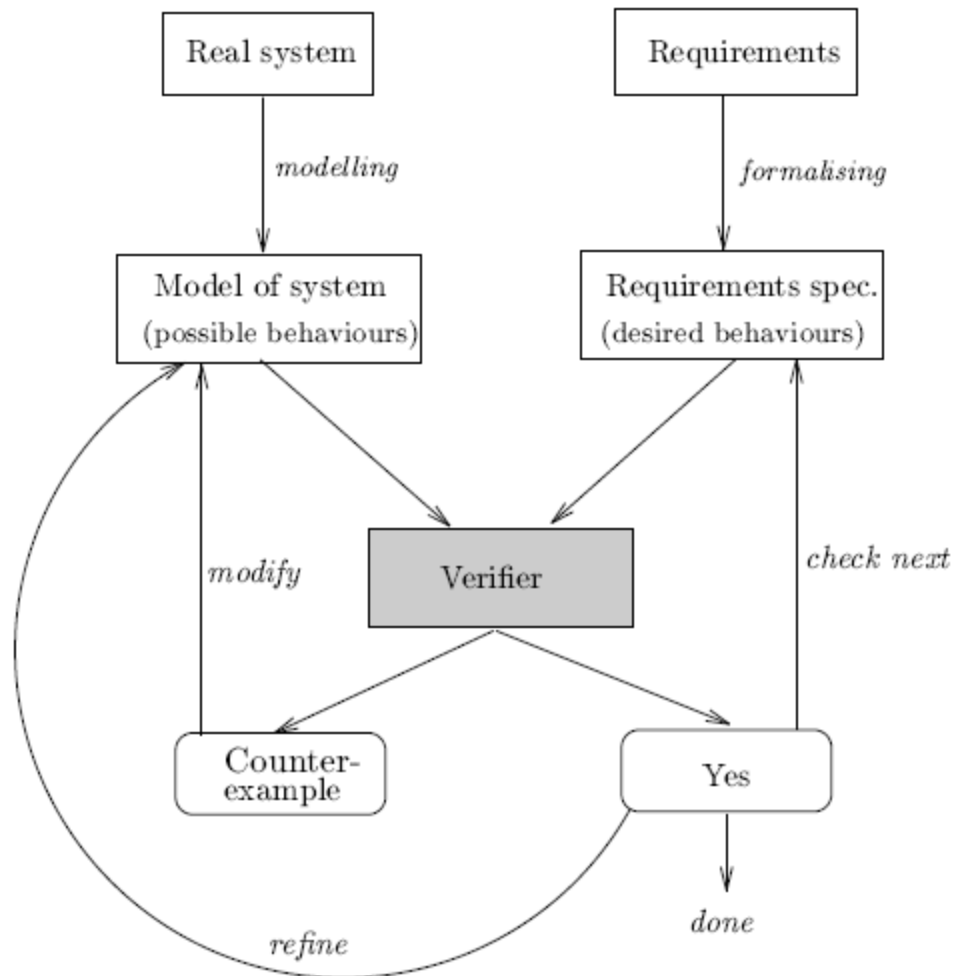


Figure : Verification methodology of model checking

Model Checking (4/4)

There has been quite some work done on automated reachability analysis for communication protocols in the late seventies and early eighties (West, 1982). Here, protocols were modeled as a set of finite-state automata that communicate via bounded buffered, asynchronous message passing. Starting from the initial system state which is expressed in terms of the states of the interacting automata and message buffers, all system states are determined that can be reached by exchanging messages. Protocols like X.21, X.25, and IBM SNA protocols have been analyzed automatically using this techniques. Model checking can in fact be considered as a successor of these early state-space exploration techniques for protocols. It allows a wider class of properties to be examined, and handles state spaces much more efficiently than these early techniques.

Methods of Model Checking

- There are basically two approaches in model checking that differ in the way the desired behavior i.e. the requirement specification is described
 1. Logic based or heterogeneous approach
 2. Behavior based or homogeneous approach

Logic based or heterogeneous approach

in this approach the desired system behavior is captured by stating a set of properties in some appropriate logic, usually some temporal or modal logic.

A system — usually modeled as a finite-state automaton, where states represent the values of variables and control locations, and transitions indicate how a system can change from one state to another — is considered to be correct with respect to these requirements if it satisfies these properties for a given set of initial states.

Behavior based or homogeneous approach (1/2)

in this approach both the desired and the possible behavior are given in the same notation (e.g. an automaton), and equivalence relations (or pre-orders) are used as a correctness criterion. The equivalence relations usually capture a notion like “behaves the same as”, whereas the pre-order relation represents a notion like “behaves at least as”. Since there are different perspectives and intuitions about what it means for two processes to “behave the same” (or “behave at least as”), various equivalence (and pre-order) notions have been defined.

Behavior based or homogeneous approach (2/2)

One of the most well-known notions of equivalence is bisimulation. In a nutshell, two automata are bisimilar if one automaton can simulate every step of the other automaton, and vice versa. A frequently encountered notion of pre-order is (language) inclusion. An automaton A is included in automaton B , if all words accepted by A are accepted by B . A system is considered to be correct if the desired and the possible behavior are equivalent (or ordered) with respect to the equivalence (or pre-order) attribute under investigation.

Although these two techniques are conceptually different, connections between the two approaches can be established in the following way.

the two approaches can be established in the following way. In particular a logic induces an equivalence relation on systems as follows: two systems are equivalent if (and only if) they satisfy the same formulas. Using this concept relationships between different logics and equivalence relations have been established. For instance, it is known that two models that are bisimilar satisfy the same formulas of the logic CTL, a logic that is commonly used for model checking purposes

The connection between the two different approaches is now clear: if two models possess the same properties (checked using the logic approach), then they are behaviorally equivalent (as could be checked in the behavioral approach).

Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model.

model checking is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite.

The benefits of model checking (1/3)

- *General approach* with applications to hardware verification, software engineering, multi-agent systems, communication protocols, embedded systems, and so forth.
- Supports *partial* verification: a design can be verified against a partial specification, by considering only a subset of all requirements. This can result in improved efficiency, since one can restrict the validation to checking only the most relevant requirements while ignoring the checking of less important, though possibly computationally expensive requirements.

The benefits of model checking (2/3)

- Case studies have shown that the incorporation of model checking in the design process does not delay this process more than using simulation and testing. For several case studies the use of model checking has led to shorter development times. In addition, due to several advanced techniques, model checkers are able to deal with rather large state spaces (an example with 10^{130} states has been reported in the literature).
- Model checkers can potentially be *routinely used* by system designers with as much ease as they use, for instance, compilers; this is basically due to the fact that model checking does not require a high degree of interaction with the user.

The benefits of model checking (3/3)

- Rapidly increasing *interest of the industry*: jobs are offered where skills in applying model checkers are required; industry is building their own model checkers (e.g. Siemens and Lucent Technologies), have initiated their own research groups on model checking (e.g. Intel and Cadence) or is using existing tools (Fujitsu, Dutch Railways, to mention a few); finally, the first model checkers are becoming commercially available.
- *Sound and interesting mathematical foundations*: modeling, semantics, concurrency theory, logic and automata theory, data structures, graph algorithms, etc. constitute the basis of model checking.

The limitation of model checking (1/5)

The main limitations of model checking are:

- Appropriate mainly to *control-intensive applications* with communications between components. It is less suited to data-intensive applications, since the treatment of data usually introduces infinite state spaces.
- The applicability of model checking is subject to *decidability issues*: for particular cases — like most classes of infinite-state systems — model checking is not effectively computable. Formal verification, though, is in principle applicable to such systems.

The limitation of model checking (2/5)

- Using model checking a *model* of the system is verified, rather than the real system itself. The fact that a model possesses certain properties does not guarantee that the final realization possesses the same properties. (For that purpose complementary techniques such as systematic testing are needed.)
In short,

*Any validation using model checking is only
as good as the model of the system.*

The limitation of model checking (3/5)

- *Only stated requirements are checked:* there is no guarantee of the completeness of desired properties.
- Finding appropriate abstraction (such as the system model and appropriate properties in temporal logic) requires some expertise.
- As any tool, model checking software might be unreliable. Since (as we will see in the sequel of these notes) standard and well-known algorithms constitute the basis for model checking, the reliability does not in principle cause severe problems. (For some cases, more advanced parts of model checking software have been proven correct using theorem provers.)

The limitation of model checking (4/5)

- It is impossible (in general) to check generalizations with model checking (Apt and Kozen, 1986). If, for instance, a protocol is verified to be correct for 1, 2 and 3 processes using model checking, it cannot provide an answer for the result of n processes, for arbitrary n . (Only for particular cases this is feasible.) Model checking can, however, suggest theorems for arbitrary parameters that subsequently can be verified using formal verification.

The limitation of model checking (5/5)

Since in model checking it is quite common to model the possible behavior of the system as (finite-state) automata, model checking is inherently vulnerable to the rather practical problem that the number of states may exceed the amount of computer memory available. This is in particular a problem for parallel and distributed systems that have many possible system states — the size of the state space of such systems is in the worst case proportional to the product of the size of the state spaces of their individual components.

The problem that the number of states can become too large is known as the *state-space explosion problem*. Several effective methods have been developed to combat this problem;

*Model checking can provide a significant increase
in the level of confidence of a system.*

Automated Theorem Proving (1/10)

Automated *theorem proving* is an approach to automate the proof of mathematical theorems. This technique can effectively be used in fields where mathematical abstractions of problems are available.

In case of system validation, the system specification and its realization are both considered as formulas, ϕ and ψ say, in some appropriate logic. Checking whether the implementation conforms to the specification now boils down to check whether $\psi \Rightarrow \phi$. This represents that each possible behavior of the implementation (i.e. satisfies ψ) is a possible behavior of the system specification (and thus satisfies ϕ).

Automated Theorem Proving (2/10)

Note that the system specification can allow other behavior that is, however, not realized.

For the proof of $\psi \Rightarrow \phi$, theorem provers can be used. Most theorem provers have algorithmic and search components. The general demand to prove theorems of a rather general type avoids to follow a solely algorithmic approach. Therefore, search components are incorporated.

Different variants exist: highly automated, general-purpose theorem provers, and interactive programs with special-purpose capabilities.

Automated Theorem Proving (3/10)

Proof checking is a field that is closely related to theorem proving. A user can give a proof of a theorem to a proof checker. The checker answers whether the proof is valid.

Usually the logics used in proof checking enable the proofs to be expressed more efficiently than those that are used in theorem provers. These differences in logic reflect the fact that proof checkers have an easier task than theorem provers, therefore checkers can deal with more complex proofs.

Automated Theorem Proving (4/10)

The verification process using theorem provers is therefore usually slow, error-prone and labour-intensive to apply. Besides this, the logic used by the tool requires a rather high degree of expertise of the user.

Automated Theorem Proving (5/10)

Logics that are used by theorem provers and proof checkers are usually variants of first-order predicate logic. In this logic we have an infinite set of variables and a set of function symbols and predicate symbols of given arities.

The arity specifies the number of arguments of a function or predicate symbol. A term is either a variable of the form $f(t_1, \dots, t_n)$ where f is a function symbol of arity n and t_i is a term. Constants can be viewed as functions of arity 0. A predicate is of the form $P(t_1, \dots, t_n)$ where P is a predicate symbol of arity n and t_i a term.

Sentences in first-order predicate logic are either predicates, logical combinations of sentences, or existential or universal quantifications over sentences.

Automated Theorem Proving (6/10)

In *typed* logics there is, in addition, a set of types and each variable has a type (like a program variable `x` has type `int`), each function symbol has a set of argument types and a result type, and each predicate symbol has a set of argument types (but no result type). In these typed logics, quantifications are over types, since the variables are typed.

Many theorem provers use *higher-order* logics: typed first-order predicate logic where variables can range over function-types or predicate-types. This enables to quantify over these types. In this way the logic becomes more expressive than first-order predicate logic.

Automated Theorem Proving (7/10)

Most theorem provers have *algorithmic* and *search* components.

The algorithmic components are techniques to apply proof rules and to obtain conclusions from this.

Important techniques that are used by theorem provers to support this are natural deduction (e.g. from the validity of ϕ_1 and the validity of ϕ_2 we may conclude the validity of $\phi_1 \wedge \phi_2$),

Automated Theorem Proving (8/10)

unification (a procedure which is used to match two terms with each other by providing all substitutions of variables under which two terms are equal),
rewriting (where equalities are considered to be directed; in case a system of equations satisfies certain conditions the application of these rules is guaranteed to yield a normal form).

Automated Theorem Proving (9/10)

Completely automated theorem provers are not very useful in practice: the problem of theorem-proving in general is exponentially difficult, i.e. the length of a proof of a sentence of length n may be of size exponential in n . (To find such proof a time that is exponential in the length of the proof may be needed; hence in general theorem proving is double exponential in the size of the sentence to be proven.) For user-interactive theorem provers this complexity is reduced to a significant extent.

Automated Theorem Proving (10/10)

Some well-known and often applied theorem provers are Coq, Isabelle, PVS, NQTHM (Boyer-Moore), nuPRL, and HOL.

Theorem Proving Vs Model Checking (1/4)

The following differences between theorem proving and model checking can be listed:

- Model checking is completely automatic and fast.
- Model checking can be applied to partial designs, and so can provide useful information about a system's correctness even if the system is not completely specified.

Theorem Proving Vs Model Checking (2/4)

- Model checkers are rather user-friendly and easy to use; the use of theorem provers requires considerable expertise to guide and assist the verification process. In particular, it is difficult to familiarize oneself with the logical language (usually some powerful higher-order logic) of the theorem prover.
- Model checking is more applicable to control-intensive applications (like hardware, communication protocols, process control systems and, more general, reactive systems). Theorem proving can deal with infinite state spaces; it is therefore also suitable for data-intensive applications.

Theorem Proving Vs Model Checking (3/4)

- When successful, theorem proving gives an (almost) maximal level of precision and reliability of the proof.
- Model checking can generate counter-examples, which can be used to aid in debugging.
- Using model checking a system design is checked for a fixed (and finite) set of parameters; using theorem provers a proof for arbitrary values of parameters can be given.

Theorem Proving Vs Model Checking (4/4)

Model checking is not considered to be “better” than theorem proving; these techniques are to a large extent complementary and both have their benefits. The emerging effort to integrate these two techniques such that one can benefit from the advantages of both approaches is interesting. In this way, one could verify a system model for a small, finite set of parameters using model checking. From these verifications one constructs a general result and subsequently the system model can be proven to be correct, for instance using inductive arguments, for an arbitrary set of parameters with the help of a theorem prover.