Software Testing and Verification

Angelo Gargantini

versione 1.0 - febbraio 26

Contents

1 A Framework for Test and Analysis

The purpose of software test and analysis is either to assess software qualities or else to make it possible to improve the software by finding defects. Of the many kinds of software qualities, those addressed by the analysis and test techniques discussed in this book are the dependability properties of the software product.

There are no perfect test or analysis techniques, nor a single "best" technique for all circumstances. Rather, techniques exist in a complex space of trade-offs, and often have complementary strengths and weaknesses. This chapter describes the nature of those trade-offs and some of their consequences, and thereby a conceptual frame-work for understanding and better integrating material from later chapters on individual techniques.

It is unfortunate that much of the available literature treats testing and analysis as independent or even as exclusive choices, removing the opportunity to exploit their complementarities. Armed with a basic understanding of the trade-offs and of strengths and weaknesses of individual techniques, one can select from and combine an array of choices to improve the cost-effectiveness of verification.

- la (bassa) qualità del sofwtware è un problema serio soprattutto in certi domini - quando il software sbaglia può fare molti danni
- come costruire sw corretto o *sicuro/safe*?
- come assicurarsi che il software sia di qualità?
- in questo corso vediamo un approccio formale

1.1 Validation and Verification

While software products and processes may be judged on several properties ranging from time-to-market to performance to usability, the software test and analysis techniques we consider are focused more narrowly on improving or assessing dependability.

Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is called validation. Fulfilling requirements is not the same as conforming to a requirements specification. A specification is a statement about a particular proposed solution to a problem, and that proposed solution may or may not achieve its goals. Moreover, specifications are written by people, and therefore

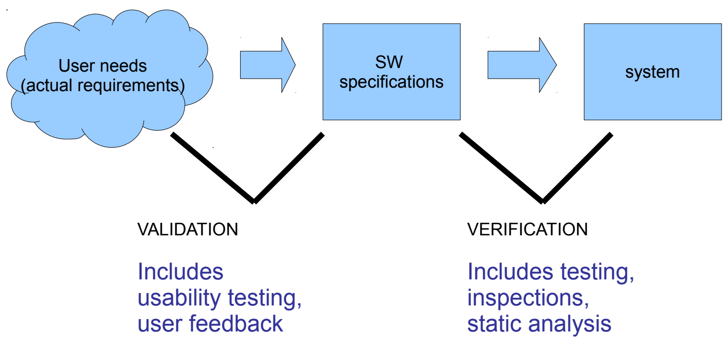

User needs (actual requirements)
SW specifications
system
VALIDATION
Includes usability testing, user feedback
VERIFICATION
Includes testing, inspections, static analysis

Figure 1.1: Validation and verification in brief

contain mistakes. A system that meets its actual goals is useful, while a system that is consistent with its specification is dependable.[1]

"Verification" is checking the consistency of an implementation with a specification. Here, "specification" and "implementation" are roles, not particular artifacts. For example, an overall design could play the role of "specification" and a more detailed design could play the role of "implementation" checking whether the detailed design is consistent with the overall design would then be verification of the detailed design. Later, the same detailed design could play the role of "specification" with respect to source code, which would be verified against the design. In every case, though, verification is a check of consistency between two descriptions, in contrast to validation which compares a description (whether a requirements specification, a design, or a running system) against actual needs.

Standard definitions

Validation & Verification - standard definitions

IEEE standard in its 4th edition defines the two terms as follows:

Validation. The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification.

Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation.

[1] A good requirements document, or set of documents, should include both a requirements analysis and a requirements specification, and should clearly distinguish between the two. The requirements analysis describes the problem. The specification describes a proposed solution. This is not a book about requirements engineering, but we note in passing that confounding requirements analysis with requirements specification will inevitably have negative impacts on both validation and verification.

ISO 9001 standard defines them this way :

Verification is the conformation that a product meets identified specifications.

Validation is the conformation that a product appropriately meets its design function or the intended use.

Validation & Verification - standard definitions
Capability Maturity Model (CMMI-SW v1.1):

Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

Boehm succinctly expressed the difference between:

Software Verification: Are we building the product right?

Software Validation: Are we building the right product?

Figure 1.2 sketches the relation of verification and validation activities with respect to artifacts produced in a software development project. The figure should not be interpreted as prescribing a sequential process, since the goal of a consistent set of artifacts and user satisfaction are the same whether the software artifacts (specifications, design, code, etc.) are developed sequentially, iteratively, or in parallel. Verification activities check consistency between descriptions (design and specifications) at adjacent levels of detail, and between these descriptions and code. [2] Validation activities attempt to gauge whether the system actually satisfies its intended purpose.

Validation activities refer primarily to the overall system specification and the final code. With respect to overall system specification, validation checks for discrepancies between actual needs and the system specification as laid out by the analysts, to ensure that the specification is an adequate guide to building a product that will fulfill its goals. With respect to final code, validation aims at checking discrepancies between actual need and the final product, to reveal possible failures of the development process and to make sure the product meets end-user expectations. Validation checks between the specification and final product are primarily checks of decisions that were left open in the specification (e.g., details of the user interface or product features). Chapter 4 provides a more thorough discussion of validation and verification activities in particular software process models.

We have omitted one important set of verification checks from Figure 2.1 to avoid clutter. In addition to checks that compare two or more artifacts, verification includes checks for self-consistency and well-formedness. For example, while we cannot judge that

[2]This part of the diagram is a variant of the well-known “V-model” of verification and validation.

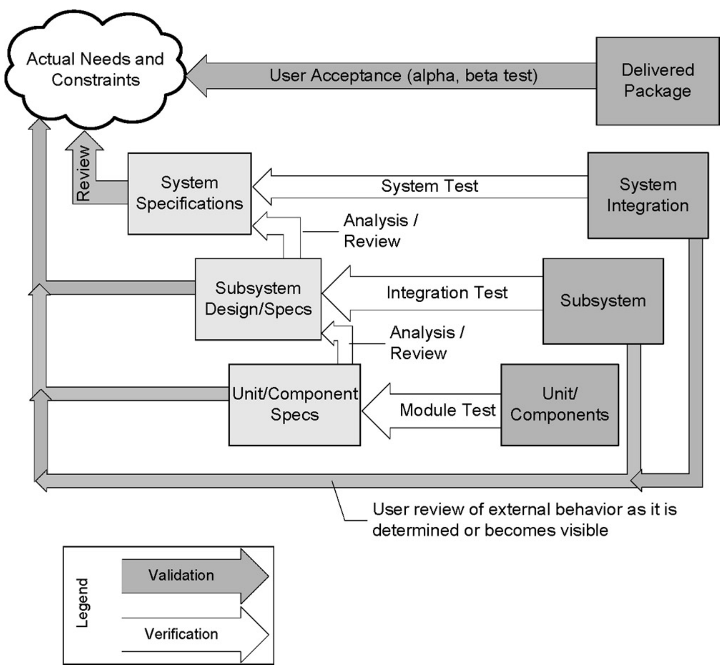


Figure 1.2: Validation activities check work products against actual user requirements, while verification activities check consistency of work products.

a program is "correct" except in reference to a specification of what it should do, we can certainly determine that some programs are "incorrect" because they are ill-formed. We may likewise determine that a specification itself is ill-formed because it is inconsistent (requires two properties that cannot both be true) or ambiguous (can be interpreted to require some property or not), or because it does not satisfy some other well-formedness constraint that we impose, such as adherence to a standard imposed by a regulatory agency.

Validation against actual requirements necessarily involves human judgment and the potential for ambiguity, misunderstanding, and disagreement. In contrast, a specification should be sufficiently precise and unambiguous that there can be no disagreement about whether a particular system behavior is acceptable. While the term testing is often used informally both for gauging usefulness and verifying the product, the activities differ in both goals and approach. Our focus here is primarily on dependability, and thus primarily on verification rather than validation, although techniques for validation and the relation between the two is discussed further in Chapter 22.

Dependability properties include correctness, reliability, robustness, and safety. Correctness is absolute consistency with a specification, always and in all circumstances. Correctness with respect to nontrivial specifications is almost never achieved. Reliability is a statistical approximation to correctness, expressed as the likelihood of correct behavior in expected use. Robustness, unlike correctness and reliability, weighs properties as more and less critical, and distinguishes which properties should be maintained even under exceptional circumstances in which full functionality cannot be maintained. Safety is a kind of robustness in which the critical property to be maintained is avoidance of particular hazardous behaviors. Dependability properties are discussed further in Chapter 4.

1.2 Degrees of Freedom

Given a precise specification and a program, it seems that one ought to be able to arrive at some logically sound argument or proof that a program satisfies the specified properties. After all, if a civil engineer can perform mathematical calculations to show that a bridge will carry a specified amount of traffic, shouldn't we be able to similarly apply mathematical logic to verification of programs?

For some properties and some very simple programs, it is in fact possible to obtain a logical correctness argument, albeit at high cost. In a few domains, logical correctness arguments may even be cost-effective for a few isolated, critical components (e.g., a safety interlock in a medical device). In general, though, one cannot produce a complete logical "proof" for the full specification of practical programs in full detail. This is not just a sign that technology for verification is immature. It is, rather, a consequence of one of the most fundamental properties of computation.

Even before programmable digital computers were in wide use, computing pioneer Alan Turing proved that some problems cannot be solved by any computer program. The universality of computers - their ability to carry out any programmed algorithm, including simulations of other computers - induces logical paradoxes regarding programs (or algorithms) for analyzing other programs. In particular, logical contradictions ensue from assuming that there is some program P that can, for some arbitrary program Q and input I, determine whether Q eventually halts. To avoid those logical contradictions, we must conclude that no such program for solving the "halting problem" can possibly exist.

Countless university students have encountered the halting problem in a course on the theory of computing, and most of those who have managed to grasp it at all have viewed it as a purely theoretical result that, whether fascinating or just weird, is irrelevant to practical matters of programming. They have been wrong. Almost every interesting property regarding the behavior of computer programs can be shown to "embed" the halting problem, that is, the existence of an infallible algorithmic check for the property of interest would imply the existence of a program that solves the halting problem, which we know to be impossible.

In theory, undecidability of a property S merely implies that for each verification technique for checking S, there is at least one "pathological" program for which that technique cannot obtain a correct answer in finite time. It does not imply that verification will always fail or even that it will usually fail, only that it will fail in at least one case. In practice, failure is not only possible but common, and we are forced to accept a significant degree of inaccuracy.

Program testing is a verification technique and is as vulnerable to undecidability as other techniques. Exhaustive testing, that is, executing and checking every possible behavior of a program, would be a "proof by cases," which is a perfectly legitimate way to construct a logical proof. How long would this take? If we ignore implementation details such as the size of the memory holding a program and its data, the answer is "forever." That is, for most programs, exhaustive testing cannot be completed in any finite amount of time.

Suppose we do make use of the fact that programs are executed on real machines with finite representations of memory values. Consider the following trivial Java class:

```
class Trivial{
        static int sum(int a, int b) {
                return a+b;
        }
}
```

The Java language definition states that the representation of an int is 32 binary digits, and thus there are only $2^{32} \times 2^{32} = 2^{64} \approx 10^{21}$ different inputs on which the method Trivial.sum() need be tested to obtain a proof of its correctness. At one nanosecond (10^{-9} seconds) per test case, this will take approximately 10^{12} seconds, or about 30,000 years.

A technique for verifying a property can be inaccurate in one of two directions (Figure 1.3). It may be pessimistic, meaning that it is not guaranteed to accept a program even if the program does possess the property being analyzed, or it can be optimistic if it may accept some programs that do not possess the property (i.e., it may not detect all violations). Testing is the classic optimistic technique, because no finite number of tests can guarantee correctness. Many automated program analysis techniques for properties of program behaviors[3] are pessimistic with respect to the properties they are designed to verify. Some analysis techniques may give a third possible answer, "don't know." We can consider these techniques to be either optimistic or pessimistic depending on how we interpret the "don't know" result. Perfection is unobtainable, but one can choose techniques that err in only a particular direction. A software verification technique that errs only in the pessimistic direction is called a conservative analysis. It might seem that a conservative analysis would always be preferable to one that could accept a faulty program. However, a conservative analysis will often produce a very large number of spurious error reports, in addition to a few accurate reports. A human may, with some effort, distinguish real faults from a few spurious reports, but cannot cope effectively with a long list of purported faults of which most are false alarms. Often only a careful choice of complementary optimistic and pessimistic techniques can help in mutually reducing the different problems of the techniques and produce acceptable results.

In addition to pessimistic and optimistic inaccuracy, a third dimension of compromise is possible: substituting a property that is more easily checked, or constraining the class of programs that can be checked. Suppose we want to verify a property S, but we are not willing to accept the optimistic inaccuracy of testing for S, and the only available static analysis techniques for S result in such huge numbers of spurious error messages that they are worthless. Suppose we know some property S´ that is a sufficient, but not necessary, condition for S (i.e., the validity of S´ implies S, but not the contrary). Maybe S´ is so much simpler than S that it can be analyzed with little or no pessimistic

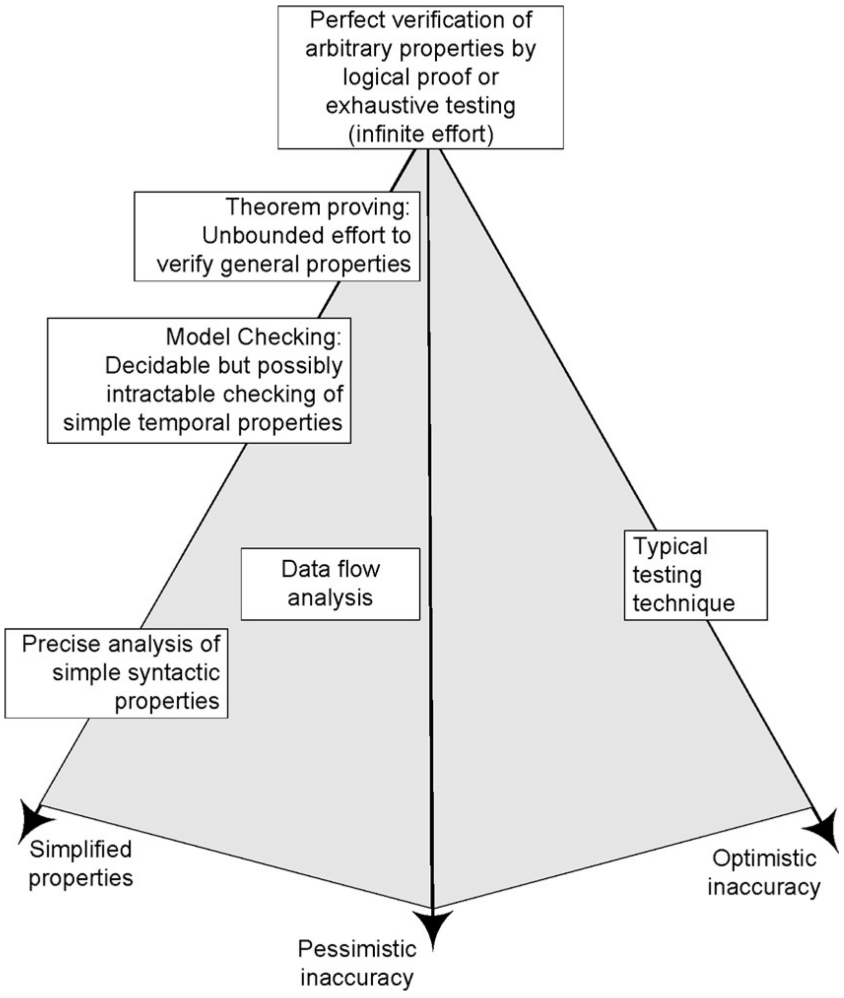

Perfect verification of arbitrary properties by logical proof or exhaustive testing (infinite effort)
Theorem proving: Unbounded effort to verify general properties
Model Checking: Decidable but possibly intractable checking of simple temporal properties
Data flow analysis
Typical testing technique
Precise analysis of simple syntactic properties
Simplified properties
Pessimistic inaccuracy
Optimistic inaccuracy

Figure 1.3: Verification trade-off dimensions

inaccuracy. If we check S´ rather than S, then we may be able to provide precise error messages that describe a real violation of S´ rather than a potential violation of S.

Many examples of substituting simple, checkable properties for actual properties of interest can be found in the design of modern programming languages. Consider, for example, the property that each variable should be initialized with a value before its value is used in an expression. In the C language, a compiler cannot provide a precise static check for this property, because of the possibility of code like the following:

```
int i, sum;
int first=1;
for (i=0; i<10; ++i) {
        if (first) {
                sum=0; first=0;
        }
        sum += i;
}
```

It is impossible in general to determine whether each control flow path can be executed, and while a human will quickly recognize that the variable sum is initialized on the first iteration of the loop, a compiler or other static analysis tool will typically not be able to rule out an execution in which the initialization is skipped on the first iteration. Java neatly solves this problem by making code like this illegal; that is, the rule is that a variable must be initialized on all program control paths, whether or not those paths can ever be executed.

A Note on Terminology

Many different terms related to pessimistic and optimistic inaccuracy appear in the literature on program analysis. We have chosen these particular terms because it is fairly easy to remember which is which. Other terms a reader is likely to encounter include:

Safe A safe analysis has no optimistic inaccuracy; that is, it accepts only correct programs. In other kinds of program analysis, safety is related to the goal of the analysis. For example, a safe analysis related to a program optimization is one that allows that optimization only when the result of the optimization will be correct.

Sound Soundness is a term to describe evaluation of formulas. An analysis of a program P with respect to a formula F is sound if the analysis returns True only when the program actually does satisfy the formula. If satisfaction of a formula F is taken as an indication of correctness, then a sound analysis is the same as a safe or conservative analysis. If the sense of F is reversed (i.e., if the truth of F indicates a fault rather than correctness) then a sound analysis is not necessarily conservative. In that case it is allowed optimistic inaccuracy but must not have pessimistic inaccuracy. (Note, however, that use of the term sound has not always been consistent in the software engineering literature. Some writers use the term unsound as we use the term optimistic.)

Complete Completeness, like soundness, is a term to describe evaluation of formulas. An analysis of a program P with respect to a formula F is complete if the analysis always returns True when the program actually does satisfy the formula. If satisfaction of a formula F is taken as an indication of correctness, then a complete analysis is one that admits only optimistic inaccuracy. An analysis that is sound but incomplete is a conservative analysis.

Software developers are seldom at liberty to design new restrictions into the programming languages and compilers they use, but the same principle can be applied through external tools, not only for programs but also for other software artifacts. Consider, for example, the following condition that we might wish to impose on requirements documents:

1. Each significant domain term shall appear with a definition in the glossary of the document.

This property is nearly impossible to check automatically, since determining whether a particular word or phrase is a “significant domain term” is a matter of human judgment. Moreover, human inspection of the requirements document to check this requirement will be extremely tedious and error-prone. What can we do? One approach is to separate the decision that requires human judgment (identifying words and phrases as “significant”) from the tedious check for presence in the glossary.

1. 1.a Each significant domain term shall be set off in the requirements document by the use of a standard style term. The default visual representation of the term style is a single underline in printed documents and purple text in on-line displays.
2. 1.b Each word or phrase in the term style shall appear with a definition in the glossary of the document.

Property (1a) still requires human judgment, but it is now in a form that is much more amenable to inspection. Property (1b) can be easily automated in a way that will be completely precise (except that the task of determining whether definitions appearing in the glossary are clear and correct must also be left to humans).

As a second example, consider a Web-based service in which user sessions need not directly interact, but they do read and modify a shared collection of data on the server. In this case a critical property is maintaining integrity of the shared data. Testing for this property is notoriously difficult, because a "race condition" (interference between writing data in one process and reading or writing related data in another process) may cause an observable failure only very rarely.

Fortunately, there is a rich body of applicable research results on concurrency control that can be exploited for this application. It would be foolish to rely primarily on direct testing for the desired integrity properties. Instead, one would choose a (well-known, formally verified) concurrency control protocol, such as the two-phase locking protocol, and rely on some combination of static analysis and program testing to check conformance to that protocol. Imposing a particular concurrency control protocol substitutes a much simpler, sufficient property (two-phase locking) for the complex property of interest (serializability), at some cost in generality; that is, there are programs that violate two-phase locking and yet, by design or dumb luck, satisfy serializability of data access.

It is a common practice to further impose a global order on lock accesses, which again simplifies testing and analysis. Testing would identify execution sequences in which data is accessed without proper locks, or in which locks are obtained and relinquished in an order that does not respect the two-phase protocol or the global lock order, even if data integrity is not violated on that particular execution, because the locking protocol failure indicates the potential for a dangerous race condition in some other execution that might occur only rarely or under extreme load.

With the adoption of coding conventions that make locking and unlocking actions easy to recognize, it may be possible to rely primarily on flow analysis to determine conformance with the locking protocol, with the role of dynamic testing reduced to a "back-up" to raise confidence in the soundness of the static analysis. Note that the critical decision to impose a particular locking protocol is not a post-hoc decision that can be made in a testing "phase" at the end of development. Rather, the plan for verification activities with a suitable balance of cost and assurance is part of system design.

[3] Why do we bother to say "properties of program behaviors" rather than "program properties?" Because simple syntactic properties of program text, such as declaring variables before they are used or indenting properly, can be decided efficiently and precisely.

1.3 Varieties of Software

The software testing and analysis techniques presented in the main parts of this book were developed primarily for procedural and object-oriented software. While these "generic" techniques are at least partly applicable to most varieties of software, particular application domains (e.g., real-time and safety-critical software) and construction methods (e.g., concurrency and physical distribution, graphical user interfaces) call for particular properties to be verified, or the relative importance of different properties, as well as imposing constraints on applicable techniques. Typically a software system does not fall neatly into one category but rather has a number of relevant characteristics that must be considered when planning verification.

As an example, consider a physically distributed (networked) system for scheduling a group of individuals. The possibility of concurrent activity introduces considerations that would not be present in a single-threaded system, such as preserving the integrity of data. The concurrency is likely to introduce nondeterminism, or else introduce an obligation to show that the system is deterministic, either of which will almost certainly need to be addressed through some formal analysis. The physical distribution may make it impossible to determine a global system state at one instant, ruling out some simplistic approaches to system test and, most likely, suggesting an approach in which dynamic testing of design conformance of individual processes is combined with static analysis of their interactions. If in addition the individuals to be coordinated are fire trucks, then the criticality of assuring prompt response will likely lead one to choose a design that is amenable to strong analysis of worst-case behavior, whereas an average- case analysis might be perfectly acceptable if the individuals are house painters.

As a second example, consider the software controlling a "soft" dashboard display in an automobile. The display may include ground speed, engine speed (rpm), oil pressure, fuel level, and so on, in addition to a map and navigation information from a global positioning system receiver. Clearly usability issues are paramount, and may even impinge on safety (e.g., if critical information can be hidden beneath or among less critical information). A disciplined approach will not only place a greater emphasis on validation of usability throughout development, but to the extent possible will also attempt to codify usability guidelines in a form that permits verification. For example, if the usability group determines that the fuel gauge should always be visible when the fuel level is below a quarter of a tank, then this becomes a specified property that is subject to verification. The graphical interface also poses a challenge in effectively checking output. This must be addressed partly in the architectural design of the system, which can make automated testing feasible or not depending on the interfaces between high-level operations (e.g., opening or closing a window, checking visibility of a window) and low-level graphical operations and representations.

Summary

Verification activities are comparisons to determine the consistency of two or more software artifacts, or self-consistency, or consistency with an externally imposed criterion.

Verification is distinct from validation, which is consideration of whether software fulfills its actual purpose. Software development always includes some validation and some verification, although different development approaches may differ greatly in their relative emphasis.

Precise answers to verification questions are sometimes difficult or impossible to obtain, in theory as well as in practice. Verification is therefore an art of compromise, accepting some degree of optimistic inaccuracy (as in testing) or pessimistic inaccuracy (as in many static analysis techniques) or choosing to check a property that is only an approximation of what we really wish to check. Often the best approach will not be exclusive reliance on one technique, but careful choice of a portfolio of test and analysis techniques selected to obtain acceptable results at acceptable cost, and addressing particular challenges posed by characteristics of the application domain or software.

Further Reading

The V-model of verification and validation (of which Figure 2.1 is a variant) appears in many software engineering textbooks, and in some form can be traced at least as far back as Myers' classic book [Mye79]. The distinction between validation and verification as given here follow's Boehm [Boe81], who has most memorably described validation as "building the right system" and verification as "building the system right."

The limits of testing have likewise been summarized in a famous aphorism, by Dijkstra [Dij72] who pronounced that "Testing can show the presence of faults, but not their absence." This phrase has sometimes been interpreted as implying that one should always prefer formal verification to testing, but the reader will have noted that we do not draw that conclusion. Howden's 1976 paper [How76] is among the earliest treatments of the implications of computability theory for program testing.

A variant of the diagram in Figure 2.2 and a discussion of pessimistic and optimistic inaccuracy were presented by Young and Taylor [YT89]. A more formal characterization of conservative abstractions in static analysis, called abstract interpretation, was introduced by Cousot and Cousot in a seminal paper that is, unfortunately, nearly unreadable [CC77]. We enthusiastically recommend Jones's lucid introduction to abstract interpretation [JN95], which is suitable for readers who have a firm general background in computer science and logic but no special preparation in programming semantics.

There are few general treatments of trade-offs and combinations of software testing and static analysis, although there are several specific examples, such as work in communication protocol conformance testing [vBDZ89, FvBK+91]. The two-phase locking protocol mentioned in Section 2.2 is described in several texts on databases; Bernstein et al. [BHG87] is particularly thorough.

Exercises

2.1 The Chipmunk marketing division is worried about the start-up time of the new version of the RodentOS operating system (an (imaginary) operating system of Chipmunk). The marketing division representative suggests a software requirement stating that the start-up time shall not be annoying to users. Explain why this simple requirement is not verifiable and try to reformulate the requirement to make it verifiable.

2.2 Consider a simple specification language SL that describes systems diagrammatically in terms of functions, which represent data transformations and correspond to nodes of the diagram, and flows, which represent data flows and correspond to arcs of the diagram.[4] Diagrams can be hierarchically refined by associating a function F (a node of the diagram) with an SL specification that details function F. Flows are labeled to indicate the type of data. Suggest some checks for self-consistency for SL.

2.3 A calendar program should provide timely reminders; for example, it should remind the user of an upcoming event early enough for the user to take action, but not too early. Unfortunately, "early enough" and "too early" are qualities that can only be validated with actual users. How might you derive verifiable dependability properties from the timeliness requirement?

2.4 It is sometimes important in multi-threaded applications to ensure that a sequence of accesses by one thread to an aggregate data structure (e.g., some kind of table) appears to other threads as an atomic transaction. When the shared data structure is maintained by a database system, the database system typically uses concurrency control protocols to ensure the atomicity of the transactions it manages. No such automatic support is typically available for data structures maintained by a program in main memory. Among the options available to programmers to ensure serializability (the illusion of atomic access) are the following:

- The programmer could maintain very coarse-grain locking, preventing any interleaving of accesses to the shared data structure, even when such interleaving would be harmless. (For example, each transaction could be encapsulated in an single synchronized Java method.) This approach can cause a great deal of unnecessary blocking between threads, hurting performance, but it is almost trivial to verify either automatically or manually.
- Automated static analysis techniques can sometimes verify serializability with finer-grain locking, even when some methods do not use locks at all. This approach can still reject some sets of methods that would ensure serializability.
- The programmer could be required to use a particular concurrency control protocol in his or her code, and we could build a static analysis tool that checks for conformance with that protocol. For example, adherence to the

common two-phase-locking protocol, with a few restrictions, can be checked in this way.

We might augment the data accesses to build a serializability graph structure representing the "happens before" relation among transactions in testing. It can be shown that the transactions executed in serializable manner if and only if the serializability graph is acyclic.

Compare the relative positions of these approaches on the three axes of verification techniques: pessimistic inaccuracy, optimistic inaccuracy, and simplified properties.

2.5 When updating a program (e.g., for removing a fault, changing or adding a functionality), programmers may introduce new faults or expose previously hidden faults. To be sure that the updated version maintains the functionality provided by the previous version, it is common practice to reexecute the test cases designed for the former versions of the program. Reexecuting test cases designed for previous versions is called regression testing. When testing large complex programs, the number of regression test cases may be large. If updated software must be expedited (e.g., to repair a security vulnerability before it is exploited), test designers may need to select a subset of regression test cases to be reexecuted. Subsets of test cases can be selected according to any of several different criteria. An interesting property of some regression test selection criteria is that they do not to exclude any test case that could possibly reveal a fault. How would you classify such a property according to the sidebar of page 21?

2 Models and Modelling

From wind-tunnels to Navier-Stokes equations to circuit diagrams to finite-element models of buildings, engineers in all fields of engineering construct and analyze models. Fundamentally, modeling addresses two problems in engineering. First, analysis and test cannot wait until the actual artifact is constructed, whether that artifact is a building or a software system. Second, it is impractical to test the actual artifact as thoroughly as we wish, whether that means subjecting it to all foreseeable hurricane and earthquake forces, or to all possible program states and inputs. Models permit us to start analysis earlier and repeat it as a design evolves, and allows us to apply analytic methods that cover a much larger class of scenarios than we can explicitly test. Importantly, many of these analyses may be automated.

This chapter presents some basic concepts in models of software and some families of models that are used in a wide variety of testing and analysis techniques. Several of the analysis and testing techniques described in subsequent chapters use and specialize these basic models. The fundamental concepts and trade-offs in the design of models is necessary for a full understanding of those test and analysis techniques, and is a foundation for devising new techniques and models to solve domain-specific problems.

2.1 Overview

A model is a representation that is simpler than the artifact it represents but preserves (or at least approximates) some important attributes of the actual artifact. Our concern in this chapter is with models of program execution, and not with models of other (equally important) attributes such as the effort required to develop the software or its usability. A good model of (or, more precisely, a good class of models) must typically be:

Compact A model must be representable and manipulable in a reasonably compact form. What is “reasonably compact” depends largely on how the model will be used. Models intended for human inspection and reasoning must be small enough to be comprehensible. Models intended solely for automated analysis may be far too large and complex for human comprehension, but must still be sufficiently small or regular for computer processing.

Predictive A model used in analysis or design must represent some salient characteristics of the modeled artifact well enough to distinguish between “good” and “bad” outcomes of analysis, with respect to those characteristics.

Typically, no single model represents all characteristics well enough to be useful for all kinds of analysis. One does not, for example, use the same model to predict airflow over an aircraft fuselage and to design internal layout for efficient passenger loading and safe emergency exit.

Semantically meaningful Beyond distinguishing between predictions of success and failure, it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure. If a finite-element model of a building predicts collapse in a category five hurricane, we want to know enough about that collapse to suggest revisions to the design. Likewise, if a model of an accounting system predicts a failure when used concurrently by several clients, we need a description of that failure sufficient to suggest possible revisions.

Sufficiently general Models intended for analysis of some important characteristic (e.g., withstanding earthquakes or concurrent operation by many clients) must be general enough for practical use in the intended domain of application.

We may sometimes tolerate limits on design imposed by limitations of our modeling and analysis techniques. For example, we may choose a conventional bridge design over a novel design because we have confidence in analysis techniques for the former but not the latter, and we may choose conventional concurrency control protocols over novel approaches for the same reason. However, if a program analysis technique for C programs is applicable only to programs without pointer variables, we are unlikely to find much use for it.

Since design models are intended partly to aid in making and evaluating design decisions, they should share these characteristics with models constructed primarily for analysis. However, some kinds of models - notably the widely used UML design notations - are designed primarily for human communication, with less attention to semantic meaning and prediction.

Models are often used indirectly in evaluating an artifact. For example, some models are not themselves analyzed, but are used to guide test case selection. In such cases, the qualities of being predictive and semantically meaningful apply to the model together with the analysis or testing technique applied to another artifact, typically the actual program or system.

Graph Representations

We often use directed graphs to represent models of programs. Usually we draw them as "box and arrow" diagrams, but to reason about them it is important to understand that they have a well-defined mathematical meaning, which we review here.

A directed graph is composed of a set of nodes N and a relation E on the set (that is, a set of ordered pairs), called the edges. It is conventional to draw the nodes as points or shapes and to draw the edges as arrows. For example: Image from book

Typically, the nodes represent entities of some kind, such as procedures or classes or regions of source code. The edges represent some relation among the entities. For example, if we represent program control flow using a directed graph model, an edge (a,b) would be interpreted as the statement “program region a can be directly followed by program region b in program execution.”

We can label nodes with the names or descriptions of the entities they represent. If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way: Image from book

Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once. For example, we might draw a single diagram in which we express both that class B extends (is a subclass of) class A and that class B has a field that is an object of type C. We can do this by drawing edges in the “extends” relation differently than edges in the “includes” relation. Image from book

Drawings of graphs can be refined in many ways, for example, depicting some relations as attributes rather than directed edges. Important as these presentation choices may be for clear communication, only the underlying sets and relations matter for reasoning about models.

Modeling maturity level

It is also useful to consider the modeling maturity level of a company that is adopting model-based testing. The UML/OCL and MDA (model-driven architecture) community has identified six levels of modeling maturity for UML development models and MDA [WK03]:

- Level 0, No Specification: The software specifications are only in the heads of the developers.
- Level 1, Textual: The software specifications are written down in informal natural-language documents.
- Level 2, Text with Diagrams: The textual specifications are augmented with some high-level diagrams.
- Level 3, Models with Text: A set of models (diagrams or text with well- defined meanings) form the backbone of the specification. Natural language is used to motivate and explain the models and to fill in many details within the models. The transition from models to code is still manual, and it can be difficult to keep models up to date after changes to the code.
- Level 4, Precise Models: This is the level where MDA becomes possible, with code being generated from the model and then modified to fulfill special requirements. The model has a precise meaning, which does not rely on natural language even though natural language is still used to explain the background of the model.
- Level 5, Models Only: At this level, the model is used like a high-level programming language, the model-to-code generation is automatic and used just like a compiler, and the generated code is used directly with- out changes. In 2003, the authors

commented that this level has not yet been reached anywhere in the world, but it is a good ultimate go

2.1.1 How to model your system

The first and most important step in modeling a system for testing is deciding on a good level of abstraction, that is, deciding which aspects of the system to include in your model and which aspects to omit. Since the model is just for verification or for test generation purposes, it does not have to specify all the behavior of the system. In the followinfg we assume that the model is used for testing. The same applies for verification. Several smaller partial models are often more useful than one huge and complex model. For example, it may be useful to write a model for each subsystem or component and to test them independently, before writing a top-level model for the whole system. So your decisions about which operations to include in the model should be driven by your top-level test objectives. Once you have decided which aspects of the SUT you wish to model, the next step in modeling a system is to think about the data that it manages, the operations that it performs, and the subsystems that it communicates with. A good notation for this is a UML class diagram, perhaps enhanced with a few textual UML use cases for the most important operations. If you already have a UML class diagram that describes the design of the SUT, you may be able to use that as a starting point for the testing model. However, a class diagram for testing purposes should be much simpler than the full class diagram that is used for design purposes. The following are some typical simplifications:

- Focus primarily on the SUT
- Show only those classes (or subsystems) associated with the SUT and whose values will be needed in the test data
- Include only those operations that you wish to test
- Include only the data fields that are useful for modeling the behavior of the operations that will be tested
- Replace a complex data field, or a class, by a simple enumeration. This allows you to limit the test data to several carefully chosen example values (one for each value of the enumeration).

For each operation that you decide to model, you should also apply the abstraction principle to its input and output parameters. If the value of an input parameter changes the behavior of an operation, and you want to test those different behaviors, then put that input parameter into your model. Otherwise, it is generally better to leave the input parameter out of the model to keep it simple—an appropriate input value can be chosen after test generation when the abstract tests are being translated into executable tests. The difficulty of test generation is usually highly dependent on the number and range of the input parameters (in addition to the state variables), so reducing this helps to control the test generation effort. Output parameters should be modeled only if their value is useful as an oracle for the test.

Key Point Design your model to meet your test objectives. When in doubt, leave it out!

Note that the operations in your model do not have to be exactly the same as the operations of the SUT. If your system has a complex operation Op, you may want to split its behavior into several cases and define one model operation Op_i for each case. On the other hand, you may want to define one model operation that corresponds to a sequence of operations in the actual system, perhaps to summarize a complex initialization sequence into a single model operation, or to reach a particular state that you want to test thoroughly.

Key Point You can have a many-to-many relationship among the operations of your model and the operations of the SUT.

The next step is to decide which notation to use for your model. This decision is often influenced by the model-based testing tools you have available and the notations they support. But in addition to this factor, it is important to consider which style of notation is most suitable for your system. In the next section, we give an overview of the different modeling notations that are available and some guidelines for choosing an appropriate notation. After you have chosen a notation and written a model of your system in that notation, the next step is to ensure that your model is accurate. You will want to validate your model (check that it does indeed specify the behavior that you want to test) and verify it (check that it is correctly typed and consistent). This is where good tool support can help. Most tool suites offer an animation tool for simulating the behavior of your model, which helps you to validate your model. They also provide tools for checking the syntax and types in your model and may offer more sophisticated tools for checking deeper properties of your model, such as an automatic prover that tries to prove that each operation of a B machine preserves the invariant or a model checker that searches for states where no operations are enabled (deadlocked states). The final step is to use your model to generate tests. This is the subject of the next few chapters. Note that your model will continue to be validated throughout the test generation process. After you generate tests from your model and execute those tests on your system, each test that fails will point either to an error in the implementation of your system or to a mistake or inadequacy in your model. The value of model-based testing comes from the automated cross-checking between these two independent works of art: the model and the system implementation.

2.1.2 Notations for Modeling

Dozens, perhaps even hundreds, of different modeling notations have been used for modeling the functional behavior of systems.

We group them into the following paradigms, adapted from van Lamsweerde [vL00].

Pre/post (or state-based) notations:

These model a system as a collection of variables, which represent a snapshot of the internal state of the sys- tem, plus some operations that modify those variables. This is similar to an object in Java or C++. Rather than the operations being defined with programming language code, each operation is usually defined by a precondition and a postcondition. Examples of these notations include B [Abr96], the UML Object Constraint Language (OCL) [WK03], the Java Modeling Language (JML) [L + 06], Spec# [Res06a], VDM [Jon90, FLM + 05] and Z [ISO02, Bow06]. Note: The traditional name for these notations is "model-based." However, this is rather confusing in our context, where all kinds of notations are being used to define models of the SUT. So, in this book, we call them pre/post notations.

Transition-based notations:

These focus on describing the transitions be- tween different states of the system. Typically, they are graphical node- and-arc notations, such as FSMs, where the nodes of the FSM represent the major states of the system and the arcs represent the actions or oper- ations of the system. Textual or tabular notations are also used to specify the transitions. In practice, transition-based notations are often made more expressive by adding data variables, hierarchies of machines, and parallelism between machines. Examples of transition-based notations include FSMs, statecharts (e.g., UML State Machines, STATEMATE statecharts, and Simulink Stateflow charts), labeled transition systems, and I/O (input/output) automata.

History-based notations:

These notations model a system by describing the allowable traces of its behavior over time. Various notions of time can be used (discrete or continuous, linear or branching, points or intervals, etc.), leading to many kinds of temporal logics. We also include message-sequence charts (MSC) and related for- malisms in this group. These are graphical and textual notations for specifying sequences of interactions among components. They are often used for modeling telecommunication protocols, particularly in combination with the System Description Language (SDL)[1]. MSCs were adopted into UML, where they are called sequence diagrams, which are one kind of interaction diagram. MSCs are good for visually showing interactions among components, but not so good at specifying the detailed behavior of each component. So, although they are sometimes used as a basis for model-based testing, our preference is to use them to describe the generated tests. That is, they are better used for visualizing the tests that result from model-based testing than for defining the model that is the input to model-based testing.

[1]See the SDL Forum Society, http://www.sdl-forum.org, for more details of SDL and MSC

Functional notations:

These describe a system as a collection of mathematical functions. The functions may be first-order only, as in the case of algebraic specifications, or higher-order, as in notations like HOL (an environment for interactive theorem proving). For example, the property push;pop = skip specifies that the pop operation undoes the effect of a push operation. Algebraic specifications tend to be more abstract and more difficult to write than other notations, so they are not widely used for model-based testing (but see [Mar95] for one test generation tool based on algebraic models).

Operational notations:

These describe a system as a collection of exe- cutable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols. Examples include process algebras such as CSP and CCS on the one hand and Petri net notations on the other hand.

Statistical notations:

These describe a system by a probabilistic model of the events and input values. They tend to be used to model environ- ments rather than SUTs. For example, Markov chains are often used to model expected usage profiles, so the generated tests exercise that usage profile. Statistical notations are good for specifying distributions of events and test inputs for the SUT but are generally weak at predicting the expected 1 .... .64 chapter 3 A Model of Your System outputs of the SUT; therefore, with only a statistical model it is not usu- ally possible to generate automated oracles as part of the tests. However, it is possible to combine a statistical model with one that models the SUT behavior. This allows the statistical model to drive the choice of test sequences and inputs, while the other model predicts the expected outputs of the SUT. Data-flow notations: These notations concentrate on the flow of data through the SUT, rather than its control flow. Some examples of this style are Lustre [MA00] and the block diagram notations that are used in Matlab Simulink[2] for the modeling of continuous systems.

2.1.3 Choosing a Notation

For model-based testing, the transition-based notations and the pre/post notations are the most used for developing behavioral models of the SUT. Which notation will be the best for modeling your SUT? In addition to practical factors, such as the availability of model-based testing tools for each notation and the degree of familiarity that you have with some notations, the answer will depend on the characteristics of your SUT. The basic guideline is to look at whether your system is more data- oriented or control-oriented. A data-oriented system typically has several state variables, with rich types

[2]See the MathWorks website, http://www.mathworks.com, for information on the Simulink product family.

such as sets, relations, maps, and sequences of values. The operations of a data-oriented system operate on that data to ac- cess and manipulate it. Data-oriented systems are most easily specified using pre/post notations, like B, which offer powerful libraries of data structures. In a control-oriented system, the set of available operations varies according to which state the system is in. For example, in the drink vending machine that we will study later in this chapter, some operations are enabled only when the machine is out of service, and others are enabled only when the machine is in service. Control-oriented systems are most easily specified using transition-based notations, such as UML state machines, because the set of transitions can be different for each state machine node.

Key Point Pre/post notations are best for data-oriented systems. Transition-based notations are best for control-oriented systems.

Of course, your system may not be so easy to classify. It may show signs of being both data-oriented and control-oriented. These classifications are not opposites, but rather two independent dimensions, so you must make a judgment call about which dimension is dominant in your system. We close by noting that it is always possible to specify control-oriented aspects of a system in a pre/post notation, and it is usually possible to specify data-oriented systems in a transition-based notation. If several "states" of your system have different behavior and could easily be modeled by nodes in a transition-system, you can still model this using a pre/post notation. You simply introduce one or more state variables to tell you which of those "nodes" your system is currently in (like state : {InService,OutOfService}), and then you add appropriate preconditions (like state = InService) or if-then-else conditions into the operations to enable the desired operations. Operations that correspond to a transition from one state to another must also update the state variables to reflect this change (like state := InService). On the other hand, a transition-based notation may have good enough support for complex data structures that it can be used for data-oriented systems. With UML state machines, OCL offers Set, OrderedSet, Bag, and Sequence data types, and associations among classes can be used to model many-to-many relations, many-to-one functions, one-to-one functions, and so on. OCL has fewer built-in operators than the B toolkit, but has a good set of quantifiers that can be used to express sophisticated queries. So choosing the "wrong" notation may make it a little more difficult to model your system. The goal is to choose the notation that suits your system best and one for which you have good tool support. Whatever notation you choose, the important thing for model-based testing is that the notation be formal. Formal means that the notation has a precise and unambiguous meaning, so that the behavior of your model can be understood and manipulated by various tools. This precise meaning makes it possible to simulate the execution of the model and to use the model as an oracle by predicting the expected output of the SUT or checking the actual output of the SUT against the model. Moreover, the precise semantics of modeling notations allow tools to perform deep consistency checks on your model, such as proving that every operation preserves the desired properties of your data structures (the invariant) or that all states are reachable.

Key Point Model-based testing requires an accurate model, written in a formal modeling notation that has precise semantics.

For example, a UML class diagram by itself does not give enough detail for test generation. One must add behavioral detail in some way, perhaps by writing OCL preconditions and postconditions for the methods of the class or perhaps by writing a state machine diagram with detailed transitions.

2.1.4 Declarative vs operational notations

Roughly speaking, there are two ways to describe a system: *operationally* (also called "imperative") and *declaratively*. An operational modeller asks "how would I make X happen?". A declarative modeller asks "how would I recognize that X has happened?". In many contexts, it is more natural and concise to use a declarative description. Often one need only to describe the rules of the game and what it means to have won, and not even think about what moves will be needed (and which must be avoided) in order to win.

Le notazioni **operazionali** e **dichiarative** sono due approcci distinti nel descrivere come funzionano i sistemi o come devono essere eseguite le operazioni, ma si differenziano principalmente nel modo in cui vengono espresse le informazioni.

Notazione Operazionali

La notazione operazionale descrive **passo per passo** il processo attraverso il quale si arriva a una soluzione. Si concentra sul *come* vengono eseguite le operazioni. In altre parole, fornisce una sequenza di azioni o comandi che devono essere eseguiti per ottenere il risultato desiderato. E' tipica di linguaggi di programmazione imperativi, come C, Java, o Python, dove si scrivono istruzioni esplicite per eseguire operazioni.

Esempio di notazione operazionale:
- Calcolare il quadrato di un numero:
- Prendi un numero ‘x‘
- Moltiplicalo per se stesso: ‘x * x‘
- Restituisci il risultato

Notazione Dichiarative

La notazione dichiarativa, invece, si concentra sul **cosa** deve essere fatto, senza entrare nei dettagli di *come* farlo. In altre parole, l'utente specifica il risultato finale che vuole ottenere, e il sistema (o il linguaggio) decide da solo il modo migliore per raggiungerlo. I linguaggi dichiarativi includono SQL, HTML, e molti linguaggi funzionali. L'utente non deve preoccuparsi della sequenza delle operazioni.

Esempio di notazione dichiarativa:
- Se vuoi trovare il quadrato di un numero:
- ‘x^2‘ (la notazione dice che vuoi il quadrato di ‘x‘, ma non specifica come calcolarlo

2.2 Finite State Machines

Finite State Machines

guarda i lucidi 2_1_model_with_FSM.pdf

1. Mealy/Moore FSM
2. Kripke Structure

2.3 Abstract State Machines

Abstract State Machines

guarda i lucidi

2.4 Logic and Temporal Logic

Guarda il prossimo capitolo

3 Formal Verification

3.1 Using logic for specifying properties

Motivation

- There is a great advantage in being able to verify the correctness of computer systems, whether they are hardware, software, or a combination. This is most obvious in the case of safety-critical systems, but also applies to those that are commercially critical, such as mass-produced chips, mission critical, etc.
  - Formal verification methods have quite recently become usable by industry and there is a growing demand for professionals able to apply them.
  - We study a fully automatic way to perform formal verification
    - not rule-based
    - called **model checking**

Formal verification by model checking

- Le tecniche di verifica formale sono generalmente viste come la somma di tre componenti:
  - Un framework in cui modellare il sistema che vogliamo analizzare
    - Un linguaggio di specifica delle proprietà da verificare
    - Un metodo per verificare che il sistema soddisfi le proprietà specificate.
  - Solitamente il Model Checking si basa sull'utilizzo di una logica temporale. Quindi, le tre componenti possono essere costituite come segue:
    - Si costruisce un modello M che descrive il comportamento del sistema
    - Si codifica la proprietà da verificare in una formula temporale ϕ
    - Si chiede al model checker di verificare che $M \models \phi$

Logiche temporali

- Esistono diverse logiche temporali che possono essere divise in due clasi fondamentali:
  - le linear-time logics (LTL) e le branching-time logics (CTL).

- LTL considera il tempo come un insieme di cammini, dove cammino é una sequenza di istanti di tempo
- CTL rappresenta il tempo come un albero, con radice l'istante corrente
- Un'altra classificazione divide tra tempo continuo e discreto. Noi studieremo solo logiche discrete e senza metrica.

3.1.1 Propositional logic

Propositional logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine. Consider the following argument:

Example 1.1
If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. Therefore, there were taxis at the station.

Intuitively, the argument is valid, since if we put the first sentence and the third sentence together, they tell us that if there are no taxis, then John will be late. The second sentence tells us that he was not late, so it must be the case that there were taxis.

Much of this book will be concerned with arguments that have this structure, namely, that consist of a number of sentences followed by the word 'therefore' and then another sentence. The argument is valid if the sentence after the 'therefore' logically follows from the sentences before it. Exactly what we mean by 'follows from' is the subject of this chapter and the next one. Consider another example:

Example 1.2 If it is raining and Jane does not have her umbrella with her, then she will get wet. Jane is not wet. It is raining. Therefore, Jane has her umbrella with her.

This is also a valid argument. Closer examination reveals that it actually has the same structure as the argument of the previous example! All we have done is substituted some sentence fragments for others:

Propositions

Example 1.1	Example 1.2
the train is late	it is raining
there are taxis at the station	Jane has her umbrella with her
John is late for his meeting	Jane gets wet.

The argument in each example could be stated without talking about trains and rain, as follows:

If p **and not** q, **then** r. **Not** r. p. **Therefore**, q.

In developing logics, we are not concerned with what the sentences really mean, but only in their logical structure. Of course, when we apply such reasoning, as done above, such meaning will be of great interest.

3.1.2 Declarative sentences

In order to make arguments rigorous, we need to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we begin with is the language of propositional logic. It is based on propositions, or declarative sentences which one can, in principle, argue as being true or false. Examples of declarative sentences are:

1. The sum of the numbers 3 and 5 equals 8.
2. Jane reacted violently to Jack's accusations.
3. Every even natural number >2 is the sum of two prime numbers.
4. All Martians like pepperoni on their pizza.
5. Albert Camus´etait un´ecrivain fran¸cais.
6. Die W¨ urde des Menschen ist unantastbar.

These sentences are all declarative, because they are in principle capable of being declared 'true', or 'false'. Sentence (1) can be tested by appealing to basic facts about arithmetic (and by tacitly assuming an Arabic, decimal representation of natural numbers). Sentence (2) is a bit more problematic. In order to give it a truth value, we need to know who Jane and Jack are and perhaps to have a reliable account from someone who witnessed the situation described. In principle, e.g., if we had been at the scene, we feel that we would have been able to detect Jane's violent reaction, provided that it indeed occurred in that way. Sentence (3), known as Goldbach's conjecture, seems straightforward on the face of it. Clearly, a fact about all even numbers >2 is either true or false. But to this day nobody knows whether sentence (3) expresses a truth or not. It is even not clear whether this could be shown by some finite means, even if it were true. However, in this text we will be content with sentences as soon as they can, in principle, attain some truth value regardless of whether this truth value reflects the actual state of affairs suggested by the sentence in question. Sentence (4) seems a bit silly, although we could say that if Martians exist and eat pizza, then all of them will either like pepperoni on it or not. (We have to introduce predicate logic in Chapter 2 to see that this sentence is also declarative if no Martians exist; it is then true.) Again, for the purposes of this text sentence (4) will do. Et alors, qu'est-ce qu'on pense des phrases (5) et (6)? Sentences (5) and (6) are fine if you happen to read French and German a bit. Thus, declarative statements can be made in any natural, or artificial, language.

NOT Declarative sentences

The kind of sentences we won't consider here are non-declarative ones, like

- Could you please pass me the salt?
- Ready, steady, go!
- May fortune come your way.

Primarily, we are interested in precise declarative sentences, or statements about the behaviour of computer systems, or programs. Not only do we want to specify such statements but we also want to check whether a given program, or system, fulfils a specification at hand. Thus, we need to develop a calculus of reasoning which allows us to draw conclusions from given assumptions, like initialised variables, which are reliable in the sense that they preserve truth: if all our assumptions are true, then our conclusion ought to be true as well. A much more difficult question is whether, given any true property of a computer program, we can find an argument in our calculus that has this property as its conclusion. The declarative sentence (3) above might illuminate the problematic aspect of such questions in the context of number theory. The logics we intend to design are symbolic in nature. We translate a certain sufficiently large subset of all English declarative sentences into strings of symbols. This gives us a compressed but still complete encoding of declarative sentences and allows us to concentrate on the mere mechanics of our argumentation. This is important since specifications of systems or software are sequences of such declarative sentences. It further opens up the possibility of automatic manipulation of such specifications, a job that computers just love to do[1].

Atomic sentences

Our strategy is to consider certain declarative sentences as being *atomic*, or *indecomposable*, like the sentence

'The number 5 is even.'

We assign certain distinct symbols p, q, r, . . ., or sometimes p1, p2, p3, . . . to each of these atomic sentences and we can then code up more complex sentences in a compositional way. For example, given the atomic sentences

p: 'I won the lottery last week.'

q: 'I purchased a lottery ticket.'

r: 'I won last week's sweepstakes.'

we can form more complex sentences according to the rules below:

[1]1 There is a certain, slightly bitter, circularity in such endeavours: in proving that a certain computer program P satisfies a given property, we might let some other computer program Q try to find a proof that P satisfies the property; but who guarantees us that Q satisfies the property of producing only correct proofs? We seem to run into an infinite regress.

$\neg$ The negation of p is denoted by ¬p and expresses 'I did not win the lottery last week,' or equivalently 'It is not true that I won the lottery last week.'

$\vee$ Given p and r we may wish to state that at least one of them is true: 'I won the lottery last week, or I won last week's sweepstakes;' we denote this declarative sentence by p ∨ r and call it the disjunction of p and r 2.

$\wedge$ Dually, the formula p ∧ r denotes the rather fortunate conjunction of p and r: 'Last week I won the lottery and the sweepstakes.'

$\rightarrow$ Last, but definitely not least, the sentence 'If I won the lottery last week, then I purchased a lottery ticket.' expresses an implication between p and q, suggesting that q is a logical consequence of p. We write p → q for that3 . We call p the assumption of p → q and q its conclusion.

$\leftrightarrow$ 'If and only if it is Sunday, I'm happy' expresses an equivalence between p and q, suggesting that q is a logical consequence of p and also p is a logical consequence of q. We write p ⟷ q for that.

Of course, we are entitled to use these rules of constructing propositions repeatedly. For example, we are now in a position to form the proposition p ∧ q → ¬r ∨ q which means that 'if p and q then not r or q'. You might have noticed a potential ambiguity in this reading. One could have argued that this sentence has the structure 'p is the case and if q then . . . ' A computer would require the insertion of brackets, as in (p ∧ q) → ((¬r) ∨ q) to disambiguate this assertion. However, we humans get annoyed by a proliferation of such brackets which is why we adopt certain conventions about the binding priorities of these symbols. Convention 1.3 ¬ binds more tightly than ∨ and ∧, and the latter two bind more tightly than →. Implication → is right-associative: expressions of the form p → q → r denote p → (q → r).

Differenza tra $\rightarrow$ e $\leftrightarrow$

$\longleftrightarrow$ 'If and only if it is Sunday, I'm happy' expresses an equivalence between p and q, suggesting that q is a logical consequence of p and also p is a logical consequence of q. We write p ⟷ q for that.

Nota che alcune volte usare ↔ è più completo che usare la semplice implicazione →.

Ad esempio se ho un metodo isPositive(x) che restituisce vero se x e positivo e false se non lo è, allora la completa carratterizzazione del metodo è $x > 0 \leftrightarrow$ isPositive(x). Se scrivessi solo $x > 0 \rightarrow isPositive(x)$ allora con x negativo il metodo sarebbe libero di ritornare quello che vuole (true o false).

3.1.3 Linear-time temporal logic

LTL sintassi

- La logica è costruita su di un insieme di formule atomiche **AP** {p, q, r, ...} che rappresentano descrizioni atomiche del sistema
  - Definiamo in maniera ricorsiva le formule LTL:
  - come la logica proposizionale (1) (| significa "oppure") - in stile come grammatica BNF
$$\phi ::= \;\top|\bot|p \in AP| \quad \neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \rightarrow \phi|$$
  - $\top, \bot$ sono vero e falso
  - $\neg, \wedge, \vee, \rightarrow$ sono connettivi logici classici

LTL sintassi - (2) operatori temporali

- Inseriamo operatori temporali (2):
$$\phi ::= \begin{array}{cc} \top|\bot|p \in AP| & \neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \rightarrow \phi| \\ X\phi|F\phi|G\phi| & \phi U\phi|\phi W\phi|\phi R\phi \end{array}$$
- X, F, G, U,W, R sono *connettivi temporali*
  - In particolare: X,F,G sono unari:
    - X means 'neXt state,'
    - F means 'some Future state,' and
    - G means 'all future states (Globally).'
  - The next three, U, R and W sono binari e sono 'Until,' 'Release' and 'Weak-until' respectively.

Alcuni esempi

- FG a -> corretto come F(G(a)): in futuro da un certo punto in poi varrà sempre a
- a U (b /\c) –> a vale fino a quando poi varrà b e c

Precedenza degli operatori

The unary connectives (consisting of $\neg$ and the temporal connectives X, F and G) bind most tightly. Next in the order come U, R and W; then come $\wedge$ and $\vee$; and after that comes $\rightarrow$.

- **Esercizio:** alcuni esempi di LTL con e senza parentesi

Semantica per LTL

- The kinds of systems we are interested in verifying using LTL may be modeled as transition systems. A transition system models a system by means of states (static structure) and transitions (dynamic structure).

  A transition system M = (S, s_0,→, L) is

  - a set of states S endowed
  - a state is the initial state s_0
  - with a transition relation → (a binary relation on S), such that every s ∈ S has some s' ∈ S with s → s', and
  - a labelling function L : S → $\mathcal{P}(AP)$

  I transition system sono i nostri *modelli*.

Labelling function

- a labelling function L : S → $\mathcal{P}(AP)$

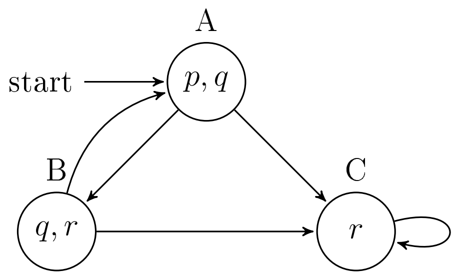
  $\mathcal{P}(AP)$ è il powerset – l'insieme delle parti – di proposizioni atomiche (**AP**)

  - L is that it is just an assignment of truth values to all the propositional atoms, as it was the case for propositional logic (we called that a valution)
  - The difference now is that we have more than one state, so this assignment depends on which state s the system is in: L(s) contains all atoms which are true in state s.

Graphical representation

- all the information about a (finite) transition system M can be expressed using directed graphs whose nodes (which we call states) contain all propositional atoms that are true in that state.

  Example: M has only three states A, B, and C. The atomic propositions AP = {p,q,r}. The only possible transitions are A → B, A → C, B → A, B → C and C → C; and if L(A) = {p, q}, L(B) = {q, r} and L(C) = {r}:



No deadlock

- The requirement in Definition that for every s ∈ S there is at least one s' ∈ S such that s → s' means that no state of the system can 'deadlock.'
  - This is a technical convenience, and in fact it does not represent any real restriction on the systems we can model. If a system did deadlock, we could always add an extra state sd representing deadlock,

    * un esempio di deadlock

Path

- A **path** in a model M = (S, →, L) is an infinite sequence of states s_1, s_2 , s_3 , . . . in S such that, for each i ≥ 1, $s_i \rightarrow s_{i+1}$.
  - We write the path as $s_1 \rightarrow s_2 \rightarrow . . . .$
  - We write π^i for the suffix starting at s_i, e.g., π^3 is $s_3 \rightarrow s_4 \rightarrow . . . .$
  - Esempio

Esempio

- A -> B ->A -> B -> C ...
  - altri esempi

Validità di una formula LTL su un path (prop)

Definition 1. Let M = (S, →, L) be a model and π = $s_1 \rightarrow . . .$ be a **path** in M. Whether π satisfies an LTL formula is defined by the satisfaction relation ⊨ as follows:

1. π⊨⊤
2. π⊭⊥
3. π⊨p iff p ∈ L(s_1)
4. π⊨¬φ iff π ⊭φ
5. π⊨φ1 ∧ φ2 iff π ⊨φ1 and π ⊨φ2
6. π⊨φ1 ∨ φ2 iff π ⊨φ1 or π ⊨φ2
7. π⊨φ1 → φ2 iff π ⊨φ2 whenever π ⊨φ1

Validità di una formula LTL su un path (time)

Definition 2. Let M = (S, →, L) be a model and π = $s_1 \to . . .$ be a path in M. Whether π satisfies an LTL formula is defined by the satisfaction relation ⊨ as follows:

8. $\pi \models X\ \varphi$ iff $\pi^2 \models \varphi$
9. $\pi \models G\ \varphi$ iff, for all i ≥ 1, $\pi^i \models \varphi$
10. $\pi \models F\ \varphi$ iff there is some i ≥ 1 such that $\pi^i \models \varphi$

Validità di una formula LTL (time 2)

11. **(Until)** $\pi \models$a U b iff there is some i ≥ 1 such that $\pi^i \models$ b and for all j = 1, . . . , i − 1 we have $\pi^j \models$ a
12. **(Weak Until)** $\pi \models$a W b iff either there is some i ≥ 1 such that $\pi^i \models$b and for all j = 1, . . . , i − 1 we have $\pi^j \models$ a; or for all k ≥ 1 we have $\pi^k \models$a

- U, which stands for 'Until,' is the most commonly encountered one of these. The formula *a U b* holds on a path if it is the case that a holds continuously until b holds. Moreover, a U b actually demands that b does hold in some future state.
  - Weak-until is just like U, except that *a W b* does not require that b is eventually satisfied along the path in question, which is required by *a U b*.

Validità di una formula LTL (time 3)

13. **(Release)** $\pi \models$a R b iff either there is some i ≥ 1 such that $\pi^i \models$ a and for all j = 1, . . . , i we have $\pi^j \models$b, or for all k ≥ 1 we have $\pi^k \models$b.

- It is called 'Release' because its definition determines that b must remain true up to and including the moment when a becomes true (if there is one); a 'releases' b.
  - Release R is the dual of U; that is, a R b is equivalent to ¬(¬a U ¬b).

Rappresentazione grafica

- Until: a is true until b become true, a U b

```
•   •   •   •   •   •   •   •   •   •
a   a   a   a   a   b
```

  - Release: a releases b: a R b

```
•   •   •   •   •   •   •   •   •   •
b   b   b   b   b   b   b
                        a
```

aggiungere grafica per weak until

Formula valida

- Quando una formula è **valida** per una macchina M (e non solo per un path) ?

  Definition 3. Suppose M =(S, → ,L) is a model, s ∈ S ,and φ an LTL formula. We write M ,s ⊨φ if, for every execution path π of M starting at s, we have $p \models f$

  Example 4. Figura 3.3 e figura 3.5, alcune formule

Formula valida con stato iniziale

- Se la macchina M ha uno stato iniziale s_0
  - Quando una formula è **valida** per una macchina M (e non solo per un path da uno stato) ?

  Definition 5. Suppose M =(S, s_0,→ ,L) is a model, s_0 ∈ S lo stato iniziale, and φ an LTL formula. We write M ⊨φ if, for every execution path π of M starting at s_0, we have $p \models f$

Formula valida con stati iniziali

- Se la macchina M ha un insieme di stati iniziali S_0
  - Quando una formula è **valida** per una macchina M (e non solo per un path) ?

  Definition 6. Suppose M =(S, S_0,→ ,L) is a model, $S_0 \subseteq S$ gli stati iniziali, and φ an LTL formula. We write M ⊨φ if per ogni $s_0 \in S_0$ vale $M, s_0 \models \varphi$

Practical Pattern of specifications

- *Safety* properties:
  - something is always true $G\phi$
    - something bad never happens $G\neg\phi$,
  - *Liveness* properties:
    - something will happen $F\phi$
    - something good keeps happening ($GF\psi$ or $G(\phi \rightarrow F\psi)$)
  - Esempi più complessi - 3.2

Important equivalences between LTL formulas

We say that two LTL formulas φ and ψ are semantically equivalent, or simply equivalent, writing φ ≡ ψ, if for all models M and all paths π in M: π ⊨φ iff π ⊨ψ.

- solite equivalenze di and, or, not ....

Until e weak until

A weak until binary operator, denoted W, with semantics similar to that of the until operator but the stop condition is not required to occur (similar to release).

- φ W $\psi \equiv (\varphi$ U $\psi) \vee$ G φ

  Both U and R can be defined in terms of the weak until:

  - Until and Weak until: φ U $\psi \equiv \varphi$ W $\psi \wedge$ F ψ

  Also R can be defined in terms of W

  - φ W $\psi \equiv (\varphi$ U $\psi) \vee$ G $\varphi \equiv \varphi$ U $(\psi \vee$ G $\varphi) \equiv \psi$ R $(\psi \vee \varphi)$ φ U $\psi \equiv$ F$\psi \wedge (\varphi$ W $\psi)$ φ R $\psi \equiv \psi$ W $(\psi \wedge \varphi)$

F and G duality

- F and G are duals:
  - $\neg$ G $\varphi \equiv$ F $\neg \varphi$ $\neg$ F $\varphi \equiv$ G $\neg \varphi$
  - X is dual of itself: $\neg$ X $\varphi \equiv$ X $\neg \varphi$
  - U and R are duals of each other:
    * $\neg$ (φ U ψ) $\equiv \neg \varphi$ R $\neg \psi$ $\neg$ (φ R ψ) $\equiv \neg \varphi$ U $\neg \psi$

Distributive

- It's also the case that F distributes over $\vee$ and G over $\wedge$, i.e.,
  - F($\varphi \vee \psi$) $\equiv$ F $\varphi \vee$ F ψ G($\varphi \wedge \psi$) $\equiv$ G $\varphi \wedge$ G ψ
  - But F does not distribute over $\wedge$ and G does not over $\vee$.
  - F and G can be written as follows using U
    * F $\varphi \equiv \top$U φ G $\varphi \equiv \bot$ R φ

Adequate sets of connectives for LTL

Non tutti i connettivi sono necessari. Basterebbero di meno, ma per facilità nelle scritture delle formule li usiamo tutti.

Pattern of LTL properties

Esistono dei pattern pratici per la specifica mediante LTL di proprietà comuni:
`http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`
Alcune volte gli operatori si indicano così: G anche [] □, F anche <>◊

Absence – P is false:	
Globally	**G** (!P)
Before R	**F** R -> (!P **U** R)
After Q	**G** (Q -> G (!P))
Between Q and R	**G** ((Q & !R & **F** R) -> (!P **U** R))

Pattern (Existence)

Existence P becomes true :	
Globally	F (P)
(*) Before R	!R W (P & !R)
After Q	G (!Q) | F (Q & F P))
(*) Between Q and R	G (Q & !R -> (!R W (P & !R)))
(*) After Q until R	G (Q & !R -> (!R U (P & !R)))

Pattern (Universality)

Universality P is true :	
Globally	G (P)
Before R	F R -> (P U R)
After Q	G (Q -> G (P))
Between Q and R	G ((Q & !R & F R) -> (P U R))
(*) After Q until R	G (Q & !R -> (P W R))

Altri Pattern

- **Precedence** S precedes P
  - **Response** S responds to P :
  - **Precedence** Chain ...

Example: mutual exclusion

When concurrent processes share a resource (such as a file on a disk or a database entry), it may be necessary to ensure that they do not have access to it at the same time. Several processes simultaneously editing the same file would not be desirable

a process to access a critical resource must be in critical section

mutual exclusion desired properties

Safety Only one process is in its critical section at any time.

Liveness: Whenever any process requests to enter its critical section, it will eventually be permitted to do so.

Non-blocking: A process can always request to enter its critical section.

No strict sequencing: Processes need not enter their critical section in strict sequence.

mutual exclusion first model

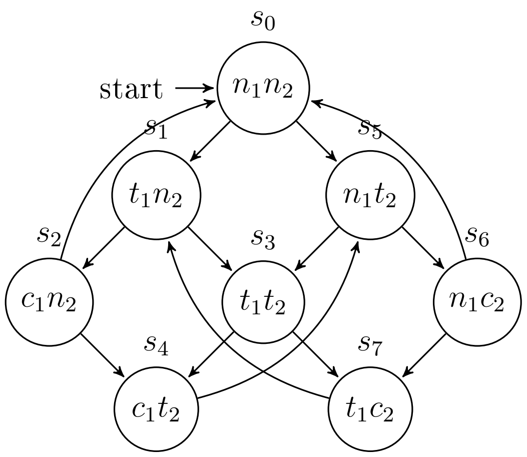


Figure 3.1: Every process can be in state: {non critical (n), trying to enter (t), critical state (c)}

.

mutual exclusion properties

Safety G ¬ (c1 ∧ c2). OK

Liveness: G (t1 → F c1). This is FALSE

Non-blocking: ... non riesco ad esprimerla in LTL

No strict sequencing: trovo un path in cui non c'è strict sequencing

Limiti LTL

Ricorda la definizione:

Definition 7. Suppose M is a model, s ∈ S ,and φ an LTL formula. We write M,s ⊨φ if, for **every** execution path π of M starting at s, we have $p \models f$

- Qundi M,s ⊨**F**a vuol dire per ogni path a partire da s a accade
  - Come faccio a dire che non sempre accade in futuro ma *potrebbe* accadere?

3.1.4 Branching-time temporal logic

CTL

COMPUTATION TREE LOGIC - CTL La CTL è una logica con connettivi che ci permette di specificare proprietà temporali.

- Essendo una logica branching-time, i suoi *modelli* sono rappresentabili mediante una struttura ad albero in cui il futuro non è deterministico: esistono differenti computazioni o paths nel futuro e uno di questi sarà il percorso realizzato.

  Cosa è un modello per una logica proposizionale ???

- Un assegnamento di un valore di verità ad ogni proposizione
    - che rende vera la formula

- $a \vee b \wedge c$: trova un modello

CTL sintassi

- La logica è costruita su di un insieme di formule atomiche AP {p, q, r, ...} che rappresentano descrizioni atomiche del sistema
    - Definiamo in maniera induttiva le formule CTL:

$$\phi ::= \top|\bot|p \in AP|\neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \to \phi|$$

    - $\top, \bot, \neg, \wedge, \vee, \to$ sono connettivi logici classici

ppp

CTL sintassi

- Operatori temporali:

$$\phi ::= \begin{array}{cc} AX\phi|EX\phi & AF\phi|EF\phi \\ |A[\phi U\phi]|E[\phi U\phi] & AG\phi|EG\phi| \end{array}$$

    - $\top, \bot, \neg, \wedge, \vee, \to$ sono connettivi logici classici
    - AX, EX, AG, EG, AU, EU, AF e EF sono *connettivi temporali*
    - In particolare: A sta per "along All paths" (inevitably) E sta per "along at least (there Exists) one path" (possibly)
    - X, F, G e U sono gli operatori della logica temporale lineare
        - Nota Bene: AU e EU sono operatori binari e i simboli X, F, G e U non possono occorrere se non preceduti da A o E e viceversa.

Priorità degli operatori

- Convenzione sull' ordinamento: gli operatori unary (AG, EG, AF, EF, AX, EX) legano con priorità più elevata, seguono gli operatori binary A, V, e dopo ancora —>, AU ed EU.
    - Esempi di formule CTL ben-formate
        - AG (q —> EG r)
        - EF E(r U q)
        - A[p U EF r]
        - EF EG p —> AF r

Attenzione

- Esempi di formule CTL non ben-formate
  - EF G r
    - A!G!p
    - F[r U q]
    - EF(r U q)
    - AEF r
    - A[(r U q) /\ (p U r)]

Semantica per CTL (brief)

Definition 8. Let M = (S, →, L) be a model for CTL, s in S, φ a CTL formula. The relation $M, s \models \varphi$ is defined by structural induction on φ.

- If φ is atomic, satisfaction is determined by L.
  - If the top-level connective of φ is a boolean connective (∧ , ∨ , ¬ , etc.) then the satisfaction question is answered by the usual truth-table definition and further recursion down φ.
  - If the top level connective is an operator beginning A, then satisfaction holds if all paths from s satisfy the 'LTL formula' resulting from removing the A symbol.
  - Similarly, if the top level connective begins with E, then satisfaction holds if some path from s satisfy the 'LTL formula' resulting from removing the E.

Semantic of CTL

Non temporal formula are treated as usual

1. $M, s \models \top$
2. $M, s \not\models \bot$
3. $M, s \models$p iff p ∈ L(s)
4. $M, s \models \neg\varphi$ iff π $M, s \not\models \varphi$
5. $M, s \models \varphi 1 \wedge \varphi 2$ iff $M, s \models \varphi 1$ and $M, s \models \varphi 2$
6. $M, s \models \varphi 1 \vee \varphi 2$ iff $M, s \models \varphi 1$ or $M, s \models \varphi 2$
7. $M, s \models \varphi 1 \rightarrow \varphi 2$ iff $M, s \models \varphi 2$ whenever $M, s \models \varphi 1$

Validità di una formula CTL (time)

8. $M, s \models$AX φ iff forall $s_1$such that $s \to s_1$ we have $M, s_1 \models\varphi$
9. $M, s \models$EX φ iff some $s_1$such that $s \to s_1$ we have $M, s_1 \models\varphi$
10. $M, s \models$AG φ iff, for all paths $s \to s_1 \to s_2 \dots$ and all s_i along the path, we have $M, s \models\varphi$
11. $M, s \models$EG φ iff, there is a path $s \to s_1 \to s_2 \dots$ and all s_i along the path, we have $M, s \models\varphi$

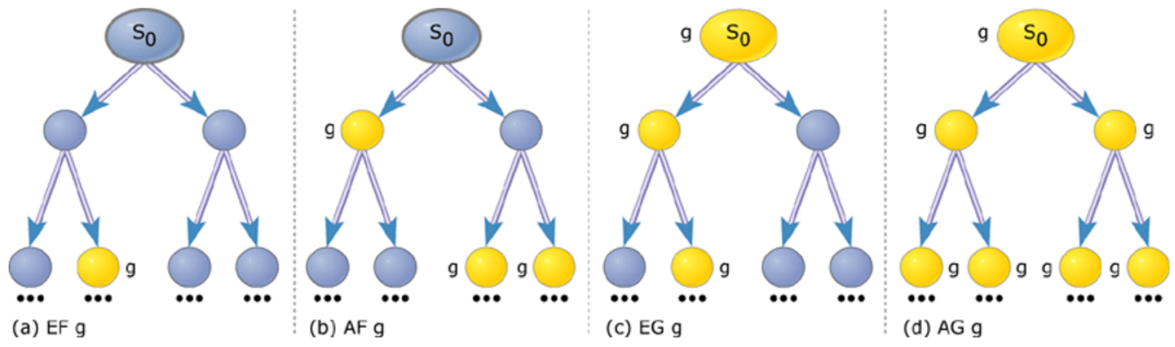
- AX: 'in every next state.'
  - EX: 'in some next state.'
  - AG: for All computation paths beginning in s the property φ holds Globally
  - EG: there Exists a path beginning in s such that φ holds Globally along the path.

Validità di una formula CTL (time 2)

12. $M, s \models$AF φ iff, for all paths $s \to s_1 \to s_2 \dots$ there exists some s_i along the path, we have $M, s \models\varphi$
13. $M, s \models$EF φ iff, there is a path $s \to s_1 \to s_2 \dots$ and for some s_i along the path, we have $M, s \models\varphi$

- AF: for All computation paths beginning in s there will be some Future state where φ holds.
  - EF: there Exists a computation path beginning in s such that φ holds in some Future state;

Validità di una formula CTL



Validità di una formula CTL (time 3)

11. $M, s \models A[\phi_1 U \phi_2]$ iff, for all paths $s \to s_1 \to s_2 \dots$, that path satisfies $\phi_1 U \phi_2$i.e., there is some s_i along the path, such that $M, s \models \phi_2$, and, for each j < i, we have $M, s \models \phi_1$.

12. $M, s \models E[\phi_1 U \phi_2]$ iff, there exists a path $s \to s_1 \to s_2 \dots$, that path satisfies $\phi_1 U \phi_2$.

- A U All computation paths beginning in s satisfy that φ1 Until φ2 holds on it.
  - E U there Exists a computation path beginning in s such that φ1 Until φ2 holds on it.

Esempio

Figura 3.3 e computation tree 3.5

Formula valida con stato iniziale

- Se la macchina M ha un insieme di stati iniziali S_0
  - Quando una formula è **valida** per una macchina M (e non solo per un path) ?

Definition 9. Suppose M =(S, S_0,→ ,L) is a model, $S_0 \subseteq S$ gli stati iniziali, and φ an CTL formula. We write $M \models \varphi$ if per ogni $s_0 \in S_0$ vale $M, s_0 \models \varphi$

Pattern of CTL properties

Esistono dei pattern pratici per la specifica mediante CTL di proprietà comuni:
`http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml`

Absence – P is false:	
Globally	AG(!P)
Before R	A[(!P | AG(!R)) W R]
After Q	AG(Q -> AG(!P))

Many of the mappings use the weak until operator (W) which is related to the strong until operator (U) by the following equivalences:
A[x W y] = !E[!y U (!x & !y)]
E[x U y] = !A[!y W (!x & !y)]

Pattern (Existence)

Existence P becomes true :	
Globally	AF(P)
(*) Before R	A[!R W (P & !R)]
After Q	A[!Q W (Q & AF(P))]
(*) Between Q and R	AG(Q & !R -> A[!R W (P & !R)])
(*) After Q until R	AG(Q & !R -> A[!R U (P & !R)])

Pattern (Universality)

Universality P is true :	
Globally	AG(P)
(*) Before R	A[(P | AG(!R)) W R]
After Q	AG(Q -> AG(P))
(*) Between Q and R	AG(Q & !R -> A[(P | AG(!R)) W R])
(*) After Q until R	AG(Q & !R -> A[P W R])

Practical patterns of specifications

- It is possible to get to a state where **started** holds, but **ready** doesn't: EF (**started** $\wedge$ $\neg$**ready**). To express impossibility, we simply negate the formula.
  - For any state, if a request (of some resource) occurs, then it will eventually be acknowledged: AG (**requested** $\rightarrow$ AF **acknowledged**).
  - A certain process is enabled infinitely often on every computation path: AG (AF enabled).
  - From any state it is possible to get to a restart state: AG (EF restart).
  - Altri esempi

Important equivalences between CTL formulas

- We have already noticed that A is a universal quantifier on paths and E is the corresponding existential quantifier. Moreover, G and F are also universal and existential quantifiers, ranging over the states along a particular path.
- We can derive the following equivalences:
  - $\neg$ AF $\varphi \equiv$ EG $\neg\varphi$ and EG $\varphi \equiv \neg$ AF $\neg\varphi$
  - $\neg$ EF $\varphi \equiv$ AG $\neg\varphi$ and AG $\varphi \equiv \neg$ EF $\neg\varphi$
  - $\neg$ AX $\varphi \equiv$ EX $\neg\varphi$.
  - We also have the equivalences AF $\varphi \equiv$ A[$\top$U φ] and
    - EF $\varphi \equiv$ E[$\top$U φ] which are similar to the corresponding equivalences in LTL.
  - Adequate sets of CTL connectives: not all the connectives are necessary.
  - We could (and will) use only AF, EU, EX

CTL* and the expressive powers of LTL and CTL

- CTL allows explicit quantification over paths, and in this respect it is more expressive than LTL, as we have seen.

  - However, it does not allow one to select a range of paths by describing them with a formula, as LTL does. In that respect, LTL is more expressive. For example, in LTL we can say 'all paths which have a p along them also have a q along them,' by writing F p → F q . It is not possible to write this in CTL because of the constraint that every F has an associated A or E.
  - CTL* is a logic which combines the expressive powers of LTL and CTL, by dropping the CTL constraint that every temporal operator (X, U, F, G) has to be associated with a unique path quantifier (A, E).
  - Past operators in LTL can be added.

Outline

- model checking algorithm for CTL logics

TESTO DI RIFERIMENTO: M.R.A. Ruth, M.D. Ryan Logic in Computer Science Modelling and Reasoning about systems - Capitolo 3 - allegato a questi appunti

3.2 Model-checking algorithms

Model checking algorithm

- We want to solve this problem $M, s \overset{???}{\models} \phi$
- There are everal approaches to formal verification
  - (automated) theorem prover
  - **model checking** (several types) ... Spin, PRISM, JavaPathFinder, ...

Model checking algorithm - CTL

- We want to solve this problem $M, s \overset{???}{\models} \phi$
  - model checking
  - by a labelling algorithm:
    * INPUT: a model M = (S, →, L) and a CTL formula φ .
    * OUTPUT: the set of states of M which satisfy φ.

1. First, change φ in terms of the connectives AF, EU, EX, ∧ , ¬ and ⊥ using the equivalences given earlier.
2. Next, label the states of M in which φ holds

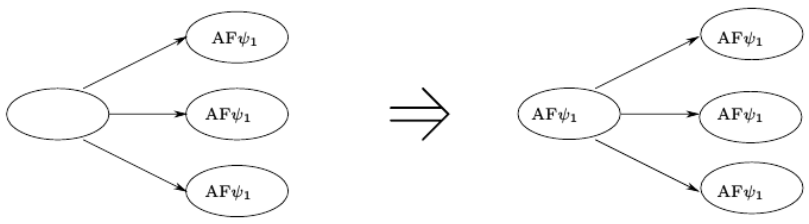
Labeling algorithm

Case analysis over ψ. If ψ is

- ⊥: then no states are labelled with ⊥
- p: then label every s such that p $\in L(s)$
- ψ1 ∧ ψ2:
  - do labelling with ψ1 and with ψ2
  - label s with ψ1 ∧ ψ2 if s is already labelled both with ψ1 and with ψ2
- ¬ψ :
  - do labelling with ψ
  - label s with ¬ψ1 if s is not labelled with ψ.

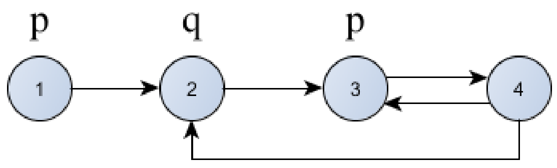
Labeling algorithm - AF

- AF ψ :
  - do labeling with ψ
    - If any state s is labelled with ψ , label it with AF ψ.
    - Repeat: label any state with AF ψ if all successor states are labelled with AF ψ , until there is no change. See picture

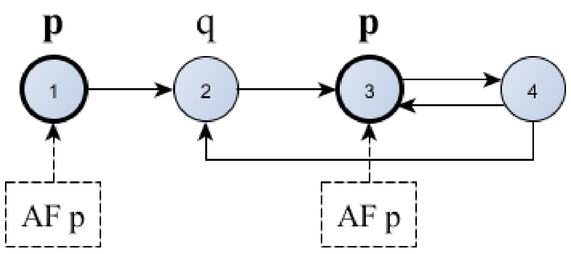


Esempio: *labeling AF*

Sia il seguente schema dove è già stato fatto il labeling per p e q:



Si farà il labeling per $AF\,p$: dove vale p si mette anche $AF\,p$.

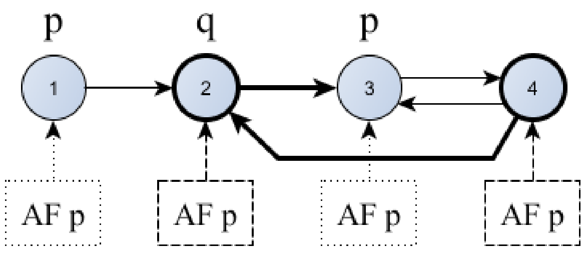


Ora si guarda di stato in stato per vedere se etichettarli come $AF\,p$ (1) Già etichettato con $AF\,p$

(2) Ho la transizione (2) $\rightarrow$ (3) e quindi avendo nello stato (3) $AF\,p$ allora anche (2) è etichettato.

(3) Già etichettato con $AF\,p$

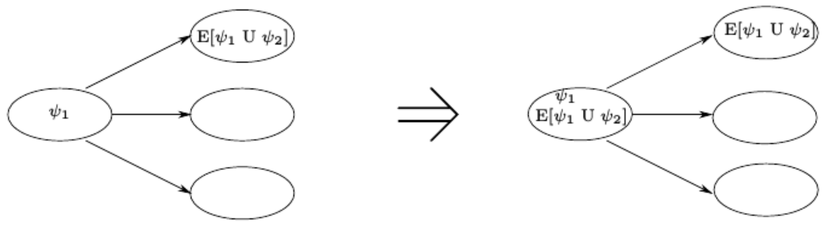
(4) Come prima, ho (4) $\rightarrow$ (2) con (2) etichettato $AF\,p$, quindi anche (4)è etichettato. Si è ottenuto il seguente grafico:



Labeling algorithm - EU

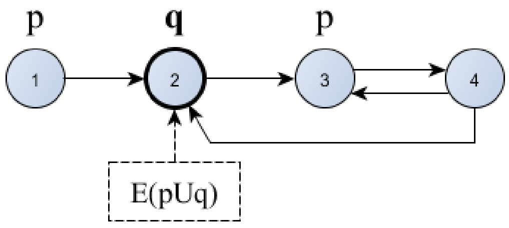
- E[ψ1 U ψ2]

- do labeling for ψ1 and ψ2
  - If any state s is labelled with ψ2, label it with E[ψ1 U ψ2]
  - Repeat: label any state with E[ψ1 U ψ2] if it is labelled with ψ1 and at least one of its successors is labelled with E[ψ1 U ψ2], until there is no change.

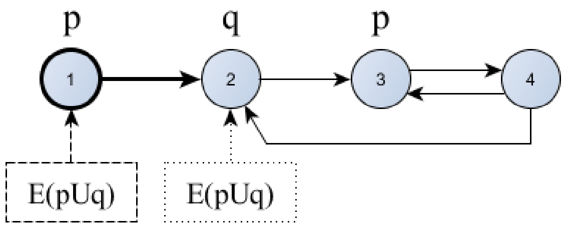


Esempio: *labeling EU*

Riprendiamo lo stesso schema dell'esercizio precedente ed etichettiamo per $E(p\,U\,q)$: si inizia mettendo l'etichetta $E(p\,U\,q)$ a tutti gli stati con q.



Cerco negli stati etichettati con p se esiste uno stato successivo etichettato con $E(p\,U\,q)$: in caso affermativo, anche lo stato con p è etichettato. (1) Ho (1) $\rightarrow$ (2) con (2) etichettato, quindi si etichetta anche (1)
(2) Non ha p
(3) Non esiste uno stato successivo etichettato con $E(p\,U\,q)$
(4) Non ha p Quindi si è ottenuto il seguente grafico:



Labeling algorithm - EX

- EX ψ :
  - do labeling for ψ
    - * label any state with EXψ if one of its successors is labelled with ψ

Complessità

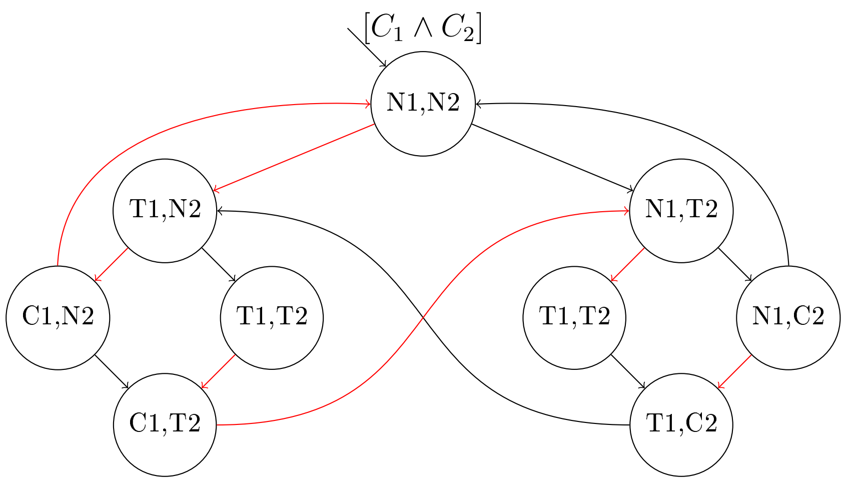
- The complexity of this algorithm is O(f · V · (V + E)), where f is the number of connectives in the formula, V is the number of states and E is the number of transitions; the algorithm is linear in the size of the formula and quadratic in the size of the model.

  - Some improvements
    * Handling EG directly
  - LTL is treated differently (skip)

Altri casi

Come faccio in tutti gli altri casi????? TODO

Esempio



State Explosion problem

The 'state explosion' problem Although the labelling algorithm (with the clever way of handling EG) is linear in the size of the model, unfortunately the size of the model is itself more often than not exponential in the number of variables and the number of components of the system which execute in parallel. This means that, for example, adding a boolean variable to your program will double the complexity of verifying a property of it. The tendency of state spaces to become very large is known as the state explosion problem. A lot of research has gone into finding ways of overcoming it, including the use of:

- efficient data structure BDDs
  - ....

L'algoritmo è lineare rispetto alle dimensioni sia di φ che di $\mathcal{M}$, tuttavia la dimensione di quest'ultimo è molto spesso esponenziale rispetto al numero di variabili e al munero di componenti del sistema che eseguono in parallelo; questo problema è noto come il **problema di esplosione degli stati**: si sono creati così tanti stati da aver occupato tutta la memoria disponibile.

Una possibile soluzione è utilizzare le strutture dati BDD (Binary Decision Diagram) che utilizzano una versione grafica alternativa all'albero (utilizzato nella logica proposizionale) per rappresentare le proposizioni.

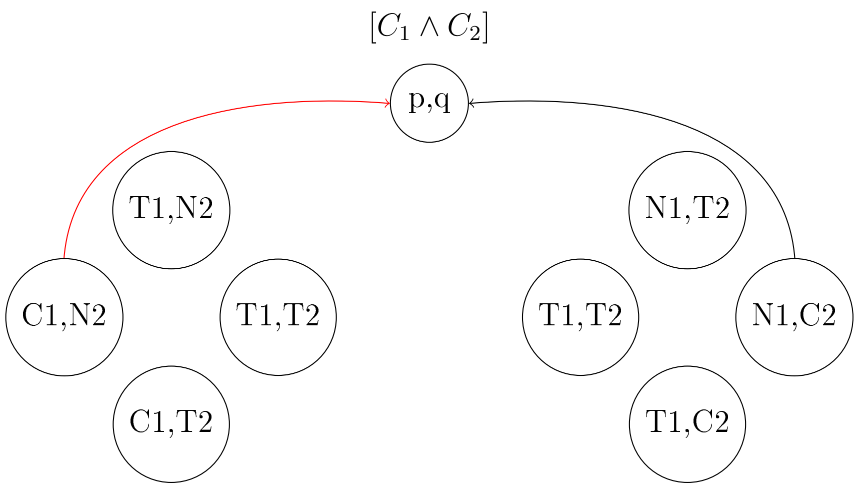
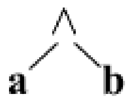
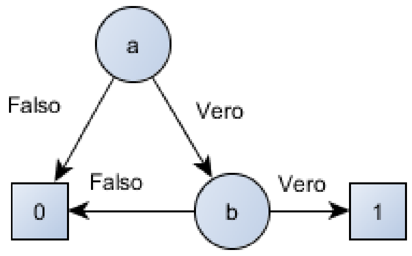


Figura 3.2: A machine

Espressione: $a \wedge b$



Albero della logica proposizionale



Binary Decision Diagram

Exercises

3.3 Model Checking

3.4 AsmetaSMV - formal verification of ASMs