ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Rigorous Software Engineering

Roman Böhringer

June 30, 2020

Department of Computer Science, ETH Zürich

Contents

0.1 Introduction

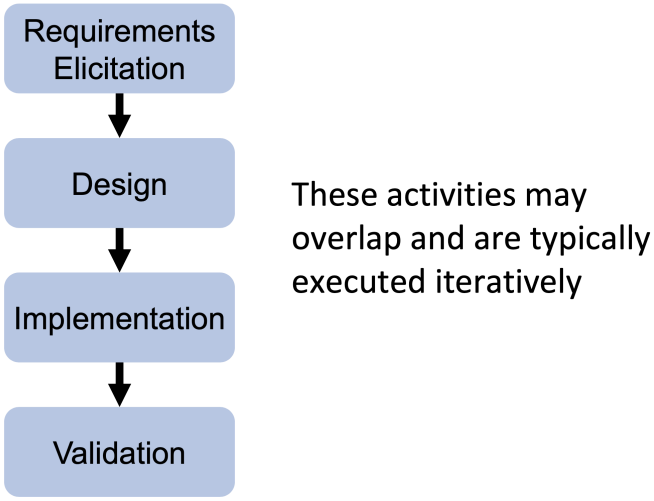
84% of all software projects are unsuccessful and the average unsuccessful project is 222% longer than planned, 189% over budget and has 61% of the originally specified features. Software is so difficult to get right for four main reasons:

1. **Complexity**: Modern software systems are huge and have a high number of discrete states (infinite if memory is unbounded) and execution paths (infinite if the system may not terminate). Complexity grows worse than linearly with size.
2. **Change**: Software systems often deviate from their initial design (with new features, interfaces or bug fixes and performance tuning) because software is perceived as being easy to change. These changes often erode the system structure.
3. **Competing Objectives**: There are various design goals that are often times competing / conflicting. Typical tradeoffs are functionality / usability, performance / portability, cost / robustness, cost / reusability and backward compatibility / understandability.
4. **Constraints**: Software development is constrained by budget, time and staff.

According to Brügge, software engineering is a collection of techniques / methodologies and tools that help producing a high quality software system with a given budget, before a given deadline, while change occurs.

Dynamic analyses (with testing being a special case of it) focuses on a subset of program behaviors and prove they are correct whereas static analyses capture all possible program behaviors in a mathematical model and prove properties of this model. Dynamic analysis is (usually) an under-approximation, static analysis (usually) an over-approximation of the possible program behaviors.

The main activities of software development are:



Design can be further divided into system design and detailed design. System design determines the software architecture as a composition of subsystems. It defines the components (computational units with specified interface, i.e. filters, databases and layers) and the connectors (interactions between components, i.e. method calls, pipes, events). Detailed design chooses among different ways to implement the system design and provides the basis for the implementation. Topics are data structures, algorithms and subclass hierarchies.

Part I

Software Design

Chapter 1

Requirements Elicitation

1.1 Requirements

A requirement is a feature that the system must have or a constraint it must satisfy to be accepted by the client. Requirements engineering defines the requirements of the system under construction and the result of it is a requirements specification (document). Requirements describe the user's view of the system and identify the what of the system, not the how (e.g. functionality, user interaction, error handling, environmental conditions / interfaces but not system structure, implementation technology, system design or development methodology).

We distinguish between:

- **Functional Requirements**:
  - Functionality (what is the software supposed to do?): This can be the relationship of outputs to inputs, the response to abnormal situations, the exact sequence of operations, validity checks on the inputs and the effect of parameters.
  - External Interfaces (interaction with people, hardware, other software): A detailed description of all inputs / outputs, consisting of purpose, source / destination, valid ranges / accuracy / tolerances, units of measure, relationships to other inputs / outputs, screen and window formats and date and command formats.
- **Nonfunctional Requirements**:
  - Performance (speed, availability, response time, recovery time): We distinguish between static numerical requirements (e.g. number of terminals or simultaneous users supported) and dynamic numerical requirements (e.g. number of transactions processed

within certain time periods for the average / peak workload, for instance 95% of the transactions shall be processed in less than 1 second).

- Attributes / Quality Requirements (portability, correctness, maintainability, security)

- Design Constraints (required standards, operating environment, etc...): Standard compliance (e.g. report format), implementation requirements (e.g. tools, programming languages, etc...), operations requirements (administration and management of the system) and possibly legal requirements (licensing, regulation, certification)

We have the following quality criteria for requirements:

- **Correctness**: They represent the client's view.
- **Completeness**: All possible scenarios are described, including exceptional behavior.
- **Consistency**: Requirements do not contradict each other.
- **Clarity (Un-ambiguity)**: Requirements can be interpreted in only one way.
- **Realism**: Requirements can be implemented and delivered.
- **Verifiability**: Repeatable tests can be designed to show that the system fulfills the requirements.
- **Traceability**: Each feature can be traced to a set of functional requirements.

A requirement like "System shall be usable by elderly people" is not verifiable and unclear, whereas "Text shall appear in letters at least 1cm high" meets the criteria.

1.1.1 Requirements Validation

Requirements validation is a quality assurance step that is usually done after requirements elicitation / analysis. It consists of reviews by clients and developers and all quality criteria are checked. Furthermore, future validations (testing) can be defined and prototypes can be built to study feasibility and give clients an impression of the future system.

1.2 Activities

1.2.1 Identifying Actors

Actors represent roles, e.g. a kind of user, external system or physical environment. Typical questions to ask are:

- Which user groups are supported by the system?
- Which user groups execute the system's main functions?
- Which user groups perform secondary functions (maintenance, administration)?
- With what external hardware / software will the system interact?

1.2.2 Identifying Scenarios

Scenarios and use cases document the behavior of the system from the user's point of view. They can be understood by customer and users. Scenarios describe common cases and focuses on understandability, whereas use cases generalize scenarios to describe all possible cases and focuses on completeness. A scenario is therefore an instance of a use case (a use case specifies all possible scenarios for a given piece of functionality).

The definition of a scenario is "A narrative description of what people do and experience as they try to make use of computer systems and applications". It is a concrete, focused, informal description of a single feature of the system. It is used in requirements elicitation, client acceptance tests and system deployment.

Questions to ask when identifying scenarios are:

- What are the tasks the actor wants the system to perform?
- What information does the actor access?
- Which external changes does the actor need to inform the system about?
- Which events does the system need to inform the actor about?

Sources of information can be the client, existing documentation, task observation and users.

1.2.3 Identifying Use Cases

A use case is a list of steps describing the interaction between an actor and the system, to achieve a goal. It consists of:

- Unique name

- Initiating / participating actors
- Flow of events
- Entry conditions
- Exit conditions
- Exceptions
- Special requirements

1.2.4 Identifying Nonfunctional Requirements

Nonfunctional requirements are defined together with functional requirements because of dependencies. Elicitation is typically done with check lists and the resulting set of nonfunctional requirements typically contains conflicts.

Chapter 2

Modeling

Source code provides very limited support for leaving design choices unspecified (because code is executable). Some relevant design information is not represented in the program or difficult to extract. Design specifications are models of the software system that provide suitable abstractions (simplifications that ignore irrelevant details, where the relevance depends on the system view / purpose of the model). Modeling is a means for dealing with complexity.

2.1 Code Documentation

Design decisions determine how code should be written and must be communicated among many different developers. Source code is insufficient because developers require information that is difficult to extract from source code (possible result values and when they occur, possible side effects, consistency conditions of data structures, how data structures evolve over time, whether objects are shared among data structures). Details in the source code may be overwhelming. Furthermore, source code does not express which properties are stable during software evolution (which details are essential, which are incidental?).

2.1.1 What to Document

For clients, one should document how to use the code (i.e. the interface) whereas for implementors, how the code works (i.e. the implementation). The documentation should focus on what the essential properties are, not how they are achieved.

For interface documentation, especially method documentation (constructors are analogous, fields can be viewed as getter / setter methods), we usually want to document:

- **Call**: How to call a method correctly (assumption on parameter values and the input state)
- **Results**: What is returned by a method (on error and success)
- **Effects**: How a method affects the state (heap effects, other effects like exceptions or blocking / non-blocking)

Some implementations also have properties that affect all methods (guarantees that are maintained by all methods together). These can be properties of states (consistency; e.g. a list is sorted), sequences (evolution; e.g. a list is immutable) or requirements / guarantees for all methods (abbreviations; e.g. a list is not thread-safe). For data structures, we want to document value and structural invariants as well as one-state and temporal invariants.

For the implementation documentation, method documentation is similar to interfaces but often includes more details. Data structure documentation is more prominent and includes implementation invariants, properties of fields, internal sharing, etc... Furthermore, there should be documentation of the algorithms inside the code (justification for assumptions, behavior of code snippets, explanation of control flow, etc...).

2.1.2 How to Document

Comments are a simple, flexible way of documenting interfaces / implementations. Tool support is limited and they aren't present in executable code.

Types document typically syntactic aspects of inputs, results and invariants. Modifiers (such as `final`) can express some specific semantic properties. There is tool support (static checking, run-time checking, auto completion) for such documentation.

Effect systems are extensions of type systems that describe computational effects (read / write effects, allocation / de-allocation, locking, exceptions). There's also tool support (static checking) for them. An example of a simple effect system are Java's checked exceptions.

Annotations (such as `@NonNull`) allow one to attach additional syntactic / semantic information to declarations. There is tool support (type checking of annotations, static processing through plugins, dynamic processing) for them.

Assertions specify semantic properties of implementations and there's also tool support (run-time checking, static checking, test case generation) for them.

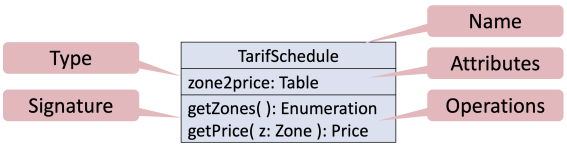
Contracts are stylized assertions for the documentation of interfaces and implementations. They state method pre- / postconditions and invariants and there's tool support for them.

2.2 Informal Models

The Unified Modeling Language (UML) is a modeling language using text and graphical notation for documenting specification, analysis, design and implementation. There are different types of diagrams for various aspects, e.g. class diagrams for the structure of a system or interaction diagrams for message passing.

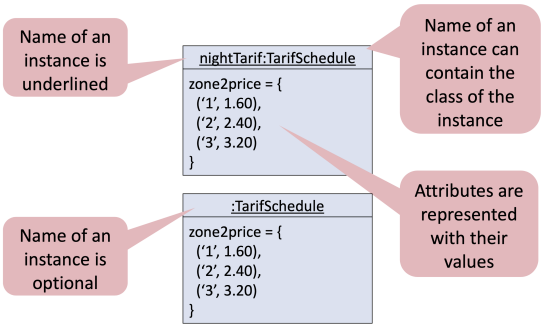
2.2.1 Static Models

A class includes state (attributes) and behavior (operations). In UML, classes are documented like this:



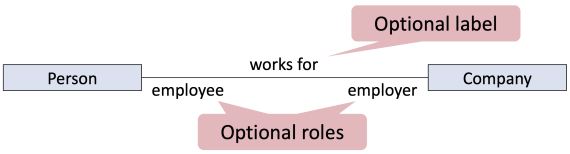
Only the class name is mandatory.

Instances (objects) are represented like this:

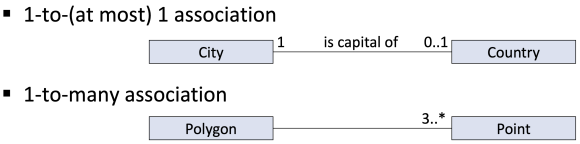


A link (an association between objects) represents a connection between two objects. This can be the ability of an object to send a message to another object, the association via attributes, creation or when an object receives a message with another object as argument.

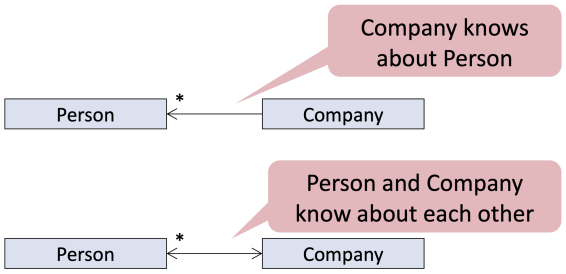
Associations between classes denote relationships:



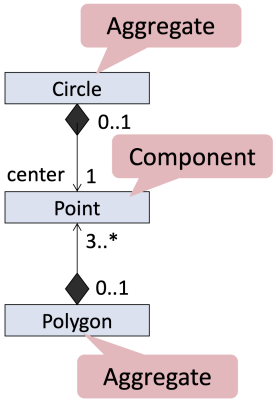
The multiplicity of an association end denotes how many objects the source object can reference and can be an exact number (where 1 is the default if nothing else is specified), an arbitrary number (zero or more) denoted with $*$ or a range, e.g. `1..3` or `1..*`.



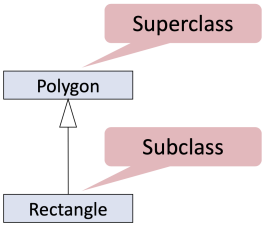
Associations can also be directed:



Composition expresses an exclusive part-of (”has-a”) relationship in which the childs (in the following example the point) cannot exist without the parents. Furthermore, there is no sharing:



Generalization expresses a kind-of (”is-a”) relationship:

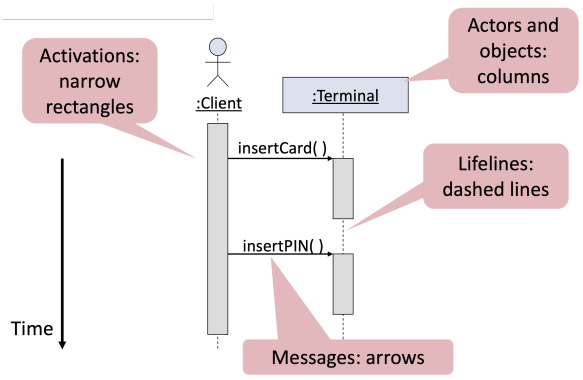


It is implemented by inheritance and simplifies the model by eliminating redundancy. Specialization is the reverse process of generalization, i.e. creating new sub-classes from an existing class (instead of combining existing classes).

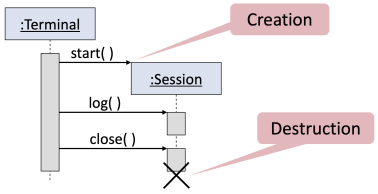
2.2.2 Dynamic Models

In contrast to static models (that describe the structure of a system), dynamic models describe its behavior. Sequence diagrams describe collaboration between objects, state diagrams the lifetime of a single object. Dynamic models should only be used for classes with significant dynamic behavior and only relevant attributes should be considered.

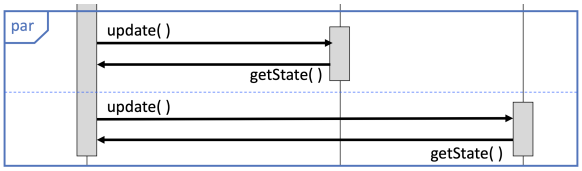
Sequence Diagrams



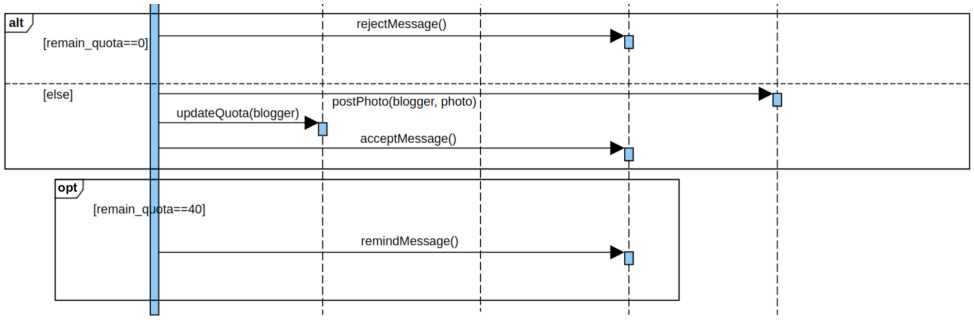
Messages can be nested (i.e. cause new messages), an activation is as long as all nested activations. Creation is denoted by a message arrow pointing to the object:



There exists also different notations for specifying certain views / properties, e.g. that some part may be executed in parallel:

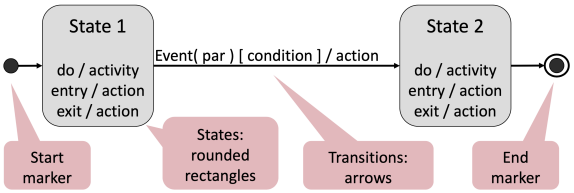


Alternatives (if / else statements) and optional segments (if statements) are depicted like this, where the condition is denoted in square brackets []:

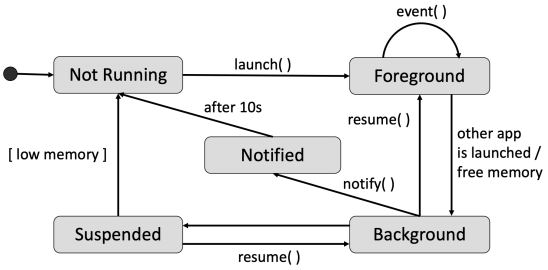


State Diagrams

States are abstractions of the attribute values of an object, a state is an equivalence class of all those attribute values and links that do not need to be distinguished for the control structure of the class. Objects with state-dependent behavior are often modeled as state diagrams / state charts:



An example of a state diagram looks like this:

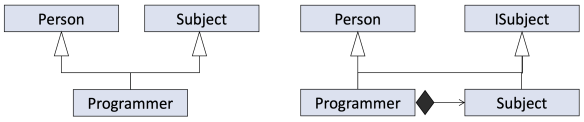


Where an event is something that happens at a point in time (e.g. receipt of a message, change event for a condition, time event, ...), an action an operation in response to an event (e.g. performing a computation upon receipt of a message) and an activity an operation performed as long as an object is in some state (e.g. performing a computation without external trigger).

2.2.3 Mapping Models to Code

The idea of model-driven development is to generate code automatically. While this has some advantages (support of many platforms, uniform code, frees programmer from recurring activities, can be used to enforce coding conventions, models aren't only documentation), there are also many problems: UML models may use different abstractions that can't be directly mapped (e.g. multiple inheritance), the specifications are incomplete and may be informal (meaning the generated code has to be extended manually) which requires complicated synchronization between code and models.

For mapping classes / inheritance, classes may be split into interfaces / implementation classes and inheritance can be mapped to subtyping and aggregation, e.g.:



Associations are typically mapped to fields or separate objects (collections).

For sequence diagrams, synchronous messages are implemented by method calls.

For state diagrams, one can introduce a state variable for the current state and check the transition conditions. If they are met, the state is changed and the action performed.

2.2.4 Contracts

Diagrams are often not detailed enough to express certain properties (e.g. restrictions on associations or values). The Object Constraint Language (OCL) is the contract language for UML and used to specify invariants of objects, pre- and postconditions of operations and conditions (e.g. in state diagrams).

To specify that a savings account has a non-negative balance, one would use:

```
context SavingsAccount inv:
self.balance >= 0
```

An example pre- / postcondition looks like this:

```
context Account::Withdraw(a: int):
pre: a >= 0
post: GetBalance() = GetBalance()@pre() - a
```

2.3 Formal Models

In formal modeling, the notations and tools are based on mathematics and therefore precise. They are typically used to describe some aspect of a system and enable automatic analysis (finding ill-formed examples or checking properties).

Alloy is a formal modeling language based on set theory. An Alloy model specifies a collection of constraints that describe a set of structures. The Alloy analyzer is a solver that takes the constraints of a model and finds structures that satisfy them (generates sample structures and counterexamples for invalid properties).

2.3.1 Logic

In Alloy, everything (including sets, scalars, tuples, structures in space and time) is a relation. Alloy checks all cases within a small bound and therefore relies on the "small scope hypothesis" that most flaws have small counterexamples. The analysis is done by SAT solvers.

Atoms are Alloy's primitive entities. They are indivisible, immutable and uninterpreted. Relations associate atoms with one another.

Sets are unary (1 column) relations, e.g.:

```
Name = {(N0),   Addr = {(A0),   Book = {(B0),
        (N1),           (A1),           (B1)}
        (N2)}           (A2)}
```

Scalars are singleton sets:

```
myName = {(N1)}
myBook = {(B0)}
```

`names` is a binary relation, `addrs` a ternary one:

```
names = {(B0, N0),  addrs = {(B0, N0, A0),
         (B0, N1),           (B0, N1, A1),
         (B1, N2)}           (B1, N2, A2)}
```

Columns are ordered but unnamed and the number of columns denotes the arity. Rows are unordered and the number of rows denotes the size of a relation. All relations are first-order, meaning relations cannot contain relations.

Alloy has the following constants:

- **none**: The empty set (`none = {}`)
- **univ**: The universal set, containing every scalar (`univ = {(N0), (N1), (N2), (A0), (A1)}`)
- **iden**: The identity relation, containing the identity relation for every atom of the universe (`iden = {(N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1)}`)

It supports the basic set operators:

- `+` union
- `&` intersection
- `-` difference
- `in` subset
- `=` equality
- `->` cross product

Where the cross product `A -> B` returns a set consisting of all possible concatenated combinations of elements from `A` and `B`.

One of the most important operator is the dot join (composition). Given two tuples s_1 `->` ... `->` s_m and t_1 `->` ... `->` t_n, one checks if the last atom of the first tuple (s_m) and the first of the second (t_1) match. If not, the result is empty. Otherwise, the concatenated tuple without the matching atom is returned, i.e. s_1 `->` ... `->` s_{m-1} `->` t_2 `->` ... `->` t_n. The dot join `p.q` of relations `p` and `q` takes every combination of a tuple in `p` and `q` and includes their join (if it exists). When `s` is a set and `r` is a binary relation, `s.r` corresponds to the image of the set `s` under the relation `r`. When `x` is a scalar and `r` a binary relation, `x.r` is the set of atoms that `x` maps to (i.e. for a function a scalar).

The box operator `[]` is semantically identical to join, but takes the arguments in a different order. `e1[e2]` has the same meaning as `e2.e1`, but dot binds more tightly than box. So `a.b.c[d]` is short for `d.(a.b.c)`. For instance, if we have a ternary relation `addr`, associating books, names, and addresses, the expression `b.addr[n]` (equivalent to `n.(b.addr)`) denotes the set of addresses associated with name `n` in book `b`.

The transpose `r` of a binary relation `~r` takes its mirror image, forming a new relation by reversing the order of atoms in each tuple. A binary relation `r` is therefore symmetric if `~r in r` holds. The symmetric closure of `r` is the smallest relation that contains `r` and is symmetric, it is equal to `r + ~r`. `r.~r in iden` says that `r` is injective, whereas `~r.r in iden` says that `r` is functional.

The transitive closure ^r of a binary relation r is the smallest relation that contains r and is transitive (i.e. whenever it contains a -> b and b -> c, it also contains a -> c). It is obtained by repeatedly taking the relation, adding the join of the relation with itself, adding the join with that, and so on: ^r=r + r.r + r.r.r + ...

The reflexive-transitive closure *r is the smallest relation that contains r and is both transitive and reflexive, it is obtained by adding the identity relation to the transitive closure: *r = ^r + iden.

The domain / range restriction operators are used to filter relations to a given domain / range. The expression s <: r (with a set s and a relation r) contains those tuples of r that start with an element in s. Similarly, r :> s contains the tuples of r that end with an element in s. For instance, if we have:

```
address = {(G0, A0), (G0, G1), (A0, D0), (G1, D0), (G1, A1)}
Alias = {(A0), (A1), (A2)}
Group = {(G0), (G1)}
Addr = {(D0), (D1), (D2)}
```

address :> Addr = {(A0, D0), (G1, D0)} (entries that map names to addresses).

Group <: address = {(G0, A0), (G0, G1), (G1, D0), (G1, A1)} (entries that map groups).

The override p ++ q of a relation p by a relation q is like the union, but tuples of q can replace the tuples of p. Any tuple in p that matches a tuple in q by starting with the same element is dropped. For instance:

```
homeAddress = {(A0, D1), (A1, D2), (A2, D3)}
workAddress = {(A0, D0), (A1, D2)}
homeAddress ++ workAddress = {(A0, D0), (A1, D2), (A2, D3)}
```

In this example, we express by the override that the preferred address for an alias is the work address if it exists, otherwise the home address. We have:

```
p ++ q = p - (domain[q] <: p) + q
```

Override can for instance be used to model insertions into maps, with:

```
m' = m ++ (k -> v)
```

For logical operators, there are two forms (a shorthand and a verbose form):

- **not** `!`
- **and** `&&`
- **or** `||`
- **implies** `=>`
- **iff** `<=>`

For quantifiers, we have:

- `all x: e | F` (`F` holds for every `x` in `e`)
- `some x: e | F` (`F` holds for some `x` in `e`)
- `no x: e | F` (`F` holds for no `x` in `e`)
- `lone x: e | F` (`F` holds for at most one `x` in `e`)
- `one x: e | F` (`F` holds for exactly one `x` in `e`)

Several variables can be bound in the same quantifier: `one x: e, y: e | F` Variables with the same bounding expression can share a declaration, the constraint can therefore also be written as: `one x, y: e | F`. With `disj`, the bindings can be restricted to include only ones in which the bound variables are disjoint from one another, e.g. `all disj x, y: e | F`.

Quantifiers can also be applied to expressions:

- `some e` (`e` has some tuples)
- `no e` (`e` has no tuples)
- `lone e` (`e` has at most one tuples)
- `one e` (`e` has exactly one tuple)

Comprehensions make relations from properties: {`x1: e1, x2: e2, ..., xn: en | F`} makes a relation with all the tuples of the form `x1 -> x2 -> ...-> xn` for which `F` holds. For instance, {`n: Name, a: Address | n -> a in ^address`} is the binary relation mapping names to reachable addresses.

Constraints of the form `relation-name: expression` are called declarations and introduces a relation name. A declaration can include multiplicity constraints:

- `set` any number
- `one` exactly one
- `lone` zero or one
- `some` one or more

The default is `one`, meaning `senderAddress: Addr` and `senderAddress: one Addr` both say that `senderAddress` is a scalar in the set `Addr`. Multiplicity constraints can also be applied to relations, `r: A -> one B` denotes a function whose domain is `A`, `r: A one -> B` an injective relation whose range is `B` and `r: A one -> one B` a bijection from `A` to `B`. The default for relation declarations is `set`.

For cardinalities / integers, we have:

- `#r` (number of tuples in `r`)
- `0, 1, ...` (integer literals)
- `plus, minus` (addition, subtraction)
- `=, <, >, =<, >=` (comparison)

`sum x: e | ie` denotes the integer obtained by summing the values of the integer expression `ie` for all values of the scalar `x` drawn from the set `e`, e.g. `sum b: Bag | #b.marbles`.

Alloy includes two logics. The expression "everybody loves a winner" can be expressed as:

- Predicate logic: `all p: Person, w: Winner | p -> w in loves`
- Relational calculus: `Person -> Winner in loves`
- Combination: `all p: Person | Winner in p.loves`

Some useful properties of binary relations can be expressed like this in relational calculus style:

```
some r: univ -> univ {
  some r  -- nonempty
  r.r in r  -- transitive
  no iden & r -- irreflexive
  ~r in r -- symmetric
  ~r.r in iden -- functional
  r.~r in iden -- injective
  univ in r.univ -- total
  univ in univ.r -- onto
}
```

However, we can also express them in predicate calculus style like this:

```
some r iff (some x, y: univ | x -> y in r)
r.r in r iff (all x, y, z: univ | (x -> y in r) and (y -> z in r)
  implies (x -> z in r))
```

```
no iden & r iff (all x: univ | x -> x not in r)
~r in r iff (all x, y: univ | x -> y in r iff y -> x in r)
~r.r in iden iff (all x, y: univ | x -> y in r implies
  (no z: univ | z != y and x -> z in r))
r.~r in iden iff (all x, y, z: univ | x -> y in r and z -> y in r
  implies x = z)
univ in r.univ iff (all x: univ | some y: univ | x -> y in r)
univ in univ.r iff (all x: univ | some y: univ | y -> x in r)
```

We can use `let` to bind expressions to a name in the following formula / expression (i.e. `let x = e | formula` or `let x = e | expression`), which can simplify the formula or expression.

2.3.2 Language / Static Models

A signature declares a set of atoms, different signatures declare disjoint sets. Extends-clauses declare subset relations, e.g. for `sig File extends FSObject{}` and `sig Dir extends FSObject{}`, `File` and `Dir` are disjoint subsets of `FSObject`. Marking a signature as abstract says that is has no elements of its own that do not belong to its extensions. Besides extends-clauses, one can also use `sig A3 in A {}` to denote that `A3` is a subset of `A`. Those signatures aren't disjoint in contrast to extends-clauses. We can also use multiplicities in the declaration of a signature (`one` for a singleton set, `lone` for a singleton or empty set, `some` for a non-empty set). Multiple inheritance can't be expressed entirely by declarations, but one can use additional facts to express it.

A signature declaration may introduce some fields, which represent a relation. For instance, `sig A {f: e}` expresses `f in A -> e`. All relations must be declared as fields (there are no top-level relation declarations), but one can use a singleton signature to declare some relations that don't belong naturally to any existing signatures, e.g.

```
one sig Globals {
    lookup: Name -> Object }
```

The `disj` keyword indicates that a group of fields are mutually disjoint, for instance `sig Cat {disj daily, peculiar, ineffable: Name}` introduces three different names for every cat. We could also use dependent declarations (expressions that reference previously declared fields) for the same declaration:

```
sig Cat {
    daily: Name,
```

```
    peculiar: Name - daily,
    ineffable: Name - (daily + peculiar) }
```

We can use multiplicities for fields as well (`one`, which is the default, for a singleton set, `lone` / `some` / `set` for a singleton or empty / non-empty / any set.

A fact records a constraint that is assumed always to hold. Facts can optionally be named and their order doesn't matter. One fact can include one or multiple constraints. For instance one would express the constraint that no directory cycles exists like this:

```
fact noCycles {
  no d: Directory | d in d.^contents }
```

Because many facts are constraints that apply to each element of a signature's set, these can also be written as signature facts (where the field references are implicitly dereferenced). Therefore, these two declarations are equivalent:

```
sig Link {from, to: Host}
fact {all x: Link | x.from != x.to}
sig Link {from, to: Host} {from != to}
```

On the other hand, an assertion introduces a constraint that is intended to follow from the facts of the model. The command `check` tells the analyzer to find a counterexample to the assertion.

Functions define reusable, parameterized expression and therefore can make the models easier to read / work with, e.g.:

```
fun leaves[ f: FSObject ]: set FSObject {
  { x: f.*contents | isLeaf[ x ] }}
```

Predicates define reusable, parameterized constraints, e.g.:

```
pred isLeaf[ f: FSObject ] {
  f in File || no f.contents }
```

The command `run <constraint> for <number>` instructs the analyzer to attempt to find a solution to the constraint (or predicate), where `for <number>` is a scope specification that limits the search to a universe in which each top-level set (i.e. signatures that don't extend another signature) contains

at most <number> elements. One can also use explicit bounds, e.g. `check A for 4 Directory, 3 File` or `check A for 5 but 3 Directory`, which limits top-level sets to 5 elements and additionally places a bound of three on `Directory`. To prescribe the exact size (not only a bound), `exactly` is used, e.g. `check A for exactly 3 Directory`.

Missing or weak facts under-constrain the model (permit undesired structures), whereas unnecessary facts over-constrain the model. Inconsistencies (which preclude the existence of any structure) are an extreme case of over-constraining and all assertion checks (even things like `0 = 1`) will succeed.

2.3.3 Dynamic Models

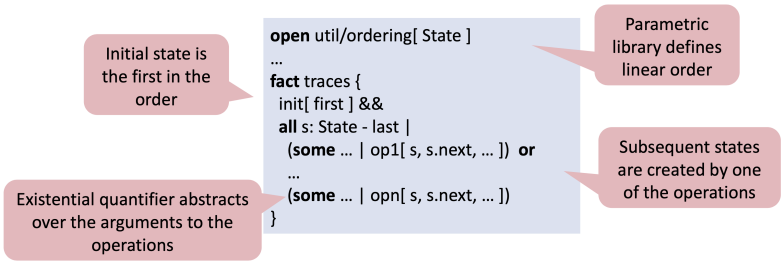
Alloy has no built-in model of execution (no notion of time or mutable state), state or time have to be modeled explicitly. Operations are described declaratively by relating the atoms before and after the operation, e.g.:

```
pred update[ a, a': Array, i: Int, e: E ] {
  a'.length = a.length &&
  a'.data = a.data ++ i -> e }
```

Because modeling mutations via different atoms can be cumbersome if atoms occur in several relations, it's common to move all relations and operations to a global state which is modified by operations. In contrast to static models, where invariants are expressed as facts, invariants can be asserted as properties maintained by the operations. For instance, we can have:

```
pred inv[ s: FileSystem ] {
  s.contents = ~(s.parent)
  s.live in s.root.*(s.contents) }
pred init[ s': FileSystem ] {
  #s'.live = 1 &&
  s'.contents[ s'.root ] = none }
assert initEstablishes {
  all s': FileSystem |
  init[ s' ] => inv[ s' ] }
check initEstablishes
```

These are one-state invariants and we often need to explore or check properties of sequences of states such as temporal invariants. To do this, sequences of execution steps of an abstract machine (execution traces) are modeled. Alloy has a library `util/ordering` for defining a linear order on all states:



Using this library, one-state invariants can be asserted more conveniently:

```
assert invHolds {
  all s: State | inv[s] }
```

Furthermore, temporal invariants can be expressed where `s.next`, `lt[s, s']` or `lte[s, s']` is used to relate states. The following assertion for instance expresses that the root of a file system never changes:

```
assert invtemp {
  all s, s': FileSystem | s.root = s'.root }
```

2.3.4 Analyzing Models

An Alloy model specifies a collection of constraints C that describe a set of structures. A formula F is consistent (satisfiable) if it evaluates to true in at least one of these structures, i.e.:

$$\exists s\,(C(s) \wedge F(s))$$

A formula F is valid if it evaluates to true in all of these structures:

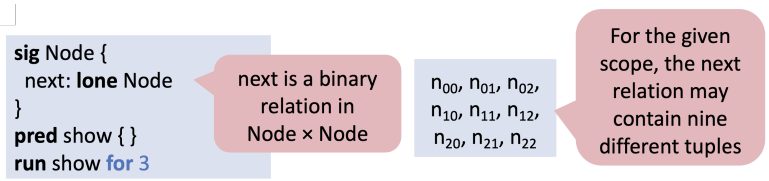
$$\forall s\,(C(s) \Rightarrow F(s))$$

Validity and consistency checking for Alloy is undecidable, the Alloy analyzer only checks validity / consistency within a given scope.

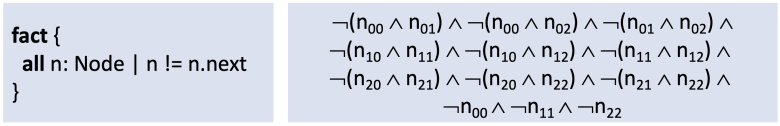
For consistency checking, the constraints and formula are translated into formula over boolean variables and it's checked whether the formula has a satisfying assignment. If not, the formula is inconsistent within the given scope, otherwise the formula is consistent and the satisfying assignment is translated back to the model.

Constraints and formulas in the model are internally represented as formulas over relations. For instance, `all n: Node | n != n.next` is translated to $\forall n\,((n,n) \notin next)$. A relation is translated into boolean variables (one

boolean variable is introduced for each tuple that is potentially contained in the relation:



Combining the model constraint and the previously described fact, we get the following formula (which is satisfiable):



Where the first 9 terms are because of the `lone` and the last three because of the `n != n.next`. A satisfying assignment can then be translated back to relations and get visualized.

For validity checking, the alloy analyzer checks for invalidity (i.e. looks for counterexamples) by making use of the following equality:

$$\neg(\forall s\,(C(s) \Rightarrow F(s))) \equiv (\exists s\,(C(s) \wedge \neg F(s)))$$

To do that, the constraints and negated formula is translated into formula over boolean vars and it is checked whether this formula has a satisfying assignment. If yes, the formula is invalid and the satisfying assignment (counterexample) is translated back to the model. If no, the formula is valid within the given scope.

Therefore, for consistency checking (performed with `run`), positive answers (structures) are definite whereas for validity checking (performed with `check`), negative answers (counterexamples) are definite.

Chapter 3

Modularity

Decomposition has many benefits:

- Partitions the overall development effort
- Supports independent testing / analysis
- Decouples parts of a system so that changes to one part do not affect other parts
- Permits the system to be understood as a composition of mind-sized chunks with one issue at a time.
- Enables reuse of components

Coupling measures interdependence between different modules, tightly-coupled modules cannot be developed, tested, changed, understood, or reused in isolation.

Generally, low coupling is a design goal. However, there are trade-offs:

- Cohesion: Each module has a clear responsibility
- Performance / convenience
- Adaptability
- Code duplication

Coupling to stable classes is less critical (e.g. from library classes).

3.1 Coupling

3.1.1 Data Coupling

Modules that expose their internal data representation become tightly coupled to their clients. Because of this, data representation is difficult to change

during maintenance, modules cannot maintain strong invariants and concurrency requires complex synchronization. Furthermore, exposing sub-objects may lead to unexpected side effects.

Different modules get coupled by operating on shared data structures, including databases and files. Problems can be caused by changes in the data structure, unexpected side effects and concurrency.

There are different approaches for this problem:

Restricting Access to Data

Clients are forced to access the data representation through a narrow interface, implementation details are hidden behind the interface (information hiding). The interface is used for the necessary checks. No sub-objects should be exposed, which means no leaking (no returning of references to sub-objects) and no capturing (which happens when arguments are stored as sub-objects). Objects need to be cloned when necessary.

This is implemented in the facade pattern. Facade objects provide a single, simplified interface to the more general facilities of a module without hiding the details completely. For instance, for a complex video conversion library, a facade object could provide a `convertVideo` function as a simple interface to the library (that hides many implementation details).

Making Shared Data Immutable

Some drawbacks of shared data (maintaining invariants, thread synchronization and unexpected side effects) only apply to mutable shared data. However, changing the data representation remains a problem and copies can lead to run-time and memory overhead.

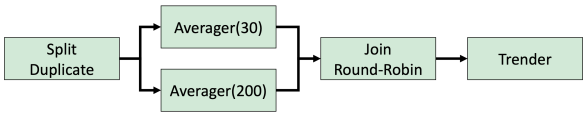
The flyweight pattern maximizes sharing of immutable objects. An immutable object that only stores the intrinsic state (constant data of an object that can only be read) is called a flyweight. In the flyweight pattern, there is a flyweight factory that looks for an existing flyweight object matching the desired state, and returns it if it was found. If not, it creates a new flyweight and adds it to the pool. Java uses the flyweight pattern for constant strings.

Avoiding Shared Data

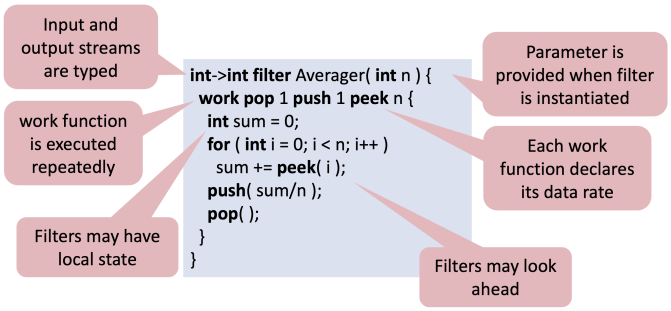
One architectural style that completely avoids shared data is the pipe-and-filter style. Data flow is the only form of communication between components. Components (filters) read data from input ports, compute and write data to output ports. Connectors (pipes) connect filters and are usually streams (asynchronous first-in-first-out buffers) or split-join connectors.

Data is processed incrementally as it arrives and output usually begins before input is consumed. Filters must be independent (with no shared state) and don't know upstream / downstream filters. Examples are unix pipes or stream processing.

For instance, if we would want to compute the 30-days / 200-days simple moving average and determine a trend for a stream of stock quotes, the architecture would look like this:



The averager in the StreamIt language would be implemented like this:



The dual averager (that splits, averages, and joins the stream) like this:

```
int -> int splitjoin dualAverager( int n, int m ) {
  split duplicate;
  add Averager(n);
  add Averager(m);
  join roundrobin;
}
```

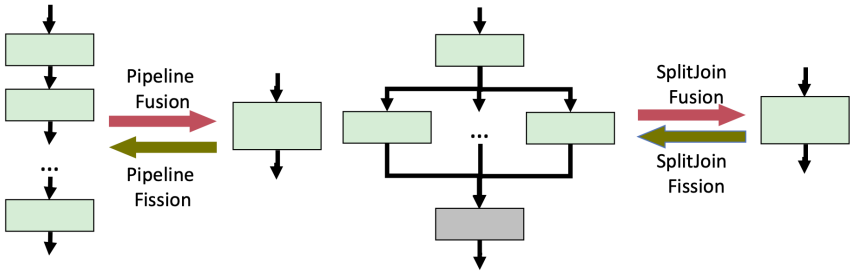
The trender like this:

```
int -> int filter Trender {
  work pop 2 push 1 {
    int a = pop();
    int b = pop();
    if(a > b) {
      push(1);
```

```
    } else {
      push(0);
    }
  }
}
```

And the whole system therefore like this:

```
int -> int pipeline System {
  add dualAverager(30, 200);
  add Trender;
}
```



Fusion / fission combines / splits up filters. Fusion reduces communication cost at the expense of parallelism. On the other hand, fission is profitable if the benefits of parallelization outweigh the overhead introduced by fission.

Strengths of the pipe-and-filter style are:

- Reuse: Any two filters can be connected if they agree on the data format that is transmitted.
- Ease of maintenance: Filters can be added or replaced.
- Potential for parallelism: Filters implemented as separate tasks, consuming and producing data incrementally.

Weaknesses are:

- Sharing global data is expensive / limiting.
- It can be difficult to design incremental filters.
- It's not appropriate for interactive applications.
- Error handling is very difficult (e.g. if some intermediate filter crashes)
- Often the smallest common denominator on the data transmission is used, e.g. ASCII for Unix pipes.

3.1.2 Procedural Coupling

Modules are coupled to other modules whose methods they call which means that callers cannot be reused without callee modules. Furthermore, any change in the callees may require changes in the caller.

There are multiple approaches for the problem:

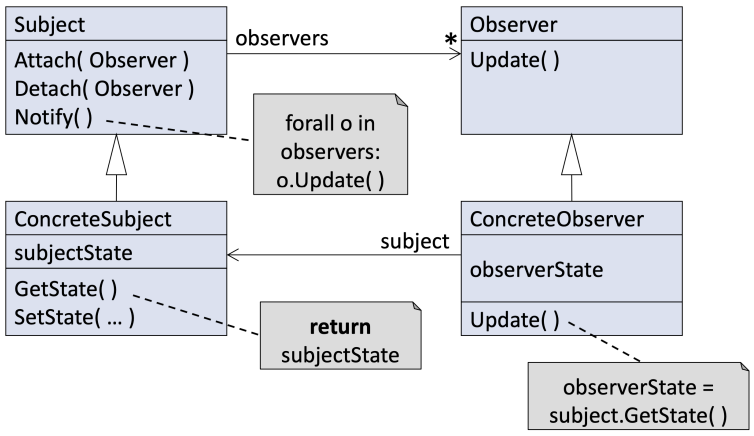
Moving Code

Moving code to other classes may reduce procedural coupling, at the possible expense of functionality duplication.

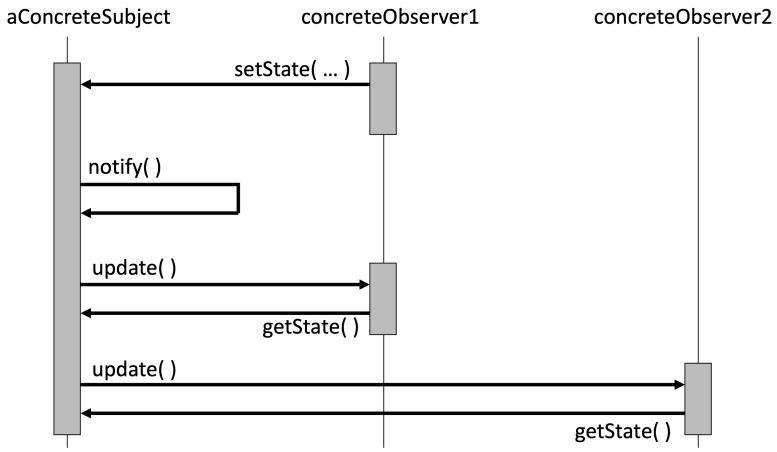
Event-Based Style

In this style, components may generate events and register for events of other components with a callback. Generators of events do not know which components will be affected by their events.

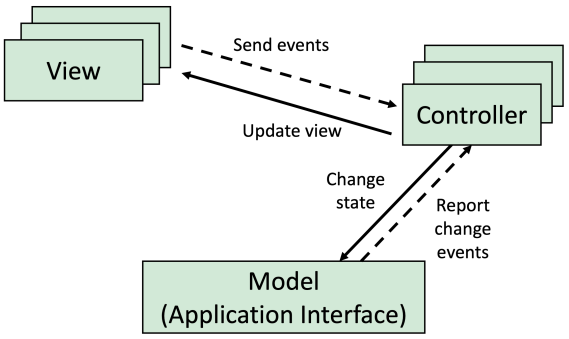
This style is implemented by the observer pattern. In this pattern, there exists subjects and observer objects (that are usually interfaces which are implemented by concrete subjects / observers). A subject has an `attach` / `detach` method which can be used to add / remove observers that should be notified on an important event. It also has a `notify` method to notify all the attached observers, which is done with a common method (e.g. `update`). After observers are notified, they can call `getState` on the subject to get the newest state:



A typical communication sequence therefore looks like this:



In the Model-View-Controller architecture style, the model contains the core functionality and data. One or more views display information to the user and one or more controllers handle user input. A change-propagation mechanism via events ensures consistency between user interface and the model. If the user changes the model through the controller of one view, the other views will be updated automatically.



Advantages of the event-based style are:

- Strong support for reuse: New components can be plugged in by registering them for events.
- Adaptation: Components can be added, removed and replaced with minimum effect on other components in the system.

Weaknesses are:

- There's a loss of control (what components, in which order will respond to an event? Are they finished?)

- Ensuring correctness is difficult because it depends on the invocation context.

Restricting Calls

One can enforce a policy that restricts which other modules a module may call. For instance with a layered architectures, where a layer only depends on lower layers and has no knowledge of higher layers. Individual layers can be exchanged. An example of this is a three-tier architecture with a presentation tier (user interface; e.g. front-end web server), logic tier (business functionality; e.g. back-end application server) and data tier (persistent storage; e.g. database).

Strengths are:

- Increasing levels of abstraction as we move up through layers, therefore partitions complex problems.
- Maintenance: A layer only interacts with the layer below (low coupling).
- Reuse: Different implementations of the same level can be interchanged.

A weakness is the performance. Communication is done down through layers and back up, bypassing may therefore occur for efficiency reasons.

3.1.3 Class Coupling

Inheritance couples the subclass to the superclass. Therefore, changes in the superclass may break the subclass.

Solution approaches for class coupling are:

Replacing Inheritance with Aggregation

As described in section 2.2.3, inheritance can be replaced by subtyping, aggregation and delegation. For instance, instead of having a `SymbolTable` that extends a `TreeMap`, it can have a field with a `TreeMap`. However, using class names in declarations of methods, fields and local variables couples the client to the used classes and data structures are difficult to change during maintenance.

Using Interfaces

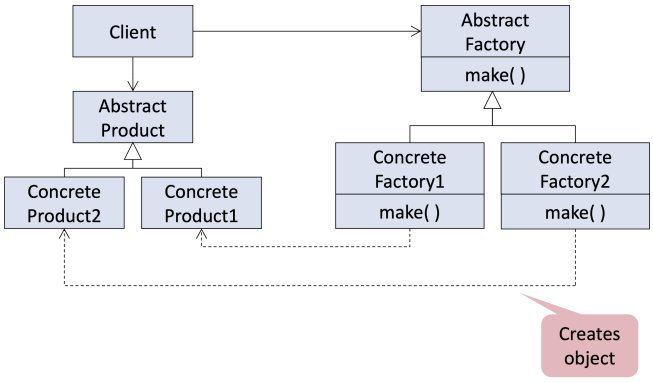
Occurences of class names are replaced by supertypes (the most general supertype that offers all required operations). For instance, a `TreeMap` field can be replaced with a `Map` field. This allows the change of data structures without affecting the code.

However, object allocations still couple clients to the instantiated class. The problem can be shifted to the class clients by making them responsible for the allocation (and having an additional parameter for the object), but then it's difficult to create objects for testing.

Delegating Allocations

In dependency injection, dependencies between classes are defined in a separate configuration file (e.g. XML). Frameworks use this information to initialize fields (e.g. via reflection).

With factories, allocations are delegated to a dedicated class called an abstract factory. Different concrete factory classes make objects of different classes. The concrete factory to be used is chosen by the client. The abstract factory pattern looks like this:



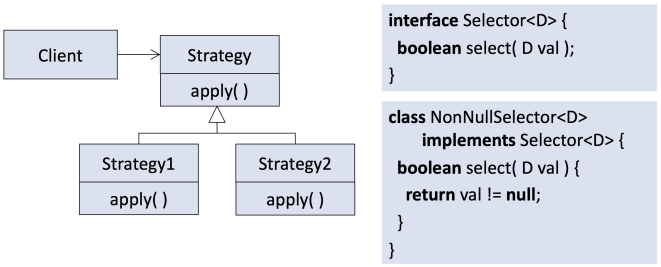
3.2 Adaptation

3.2.1 Parameterization

Modules can be prepared for change by allowing clients to influence their behavior (parametrization). Modules can be made parametric in:

- The values they manipulate: This is done by using variable values instead of constant values, e.g. an array of streams instead hard-coding two streams.
- The data structures they operate on: This can be done using interfaces / factories instead of concrete classes.
- The types they operate on: Generic types allow type parametrization.
- The algorithms they apply: Function objects (delegates in C#, function pointers in C++, lambda expressions in Java) allow the parametrization of algorithms.

The strategy pattern (for algorithm parametrization) is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

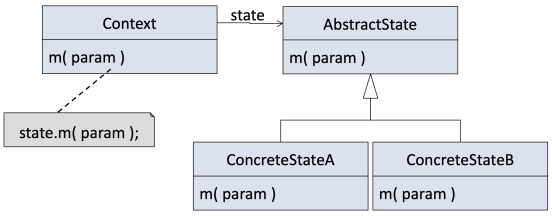


3.2.2 Specialization

Overriding and dynamic method binding allows specialized behavior. Dynamic method binding is a case distinction on the dynamic type of the receiver object. Adding or removing cases (method overrides) does not require changes in the caller. However, dynamic method binding has drawbacks:

- Reasoning: Subclasses share responsibility for maintaining invariants.
- Testing: Dynamic binding increases the number of possible behaviors that need to be tested.
- Versioning: Dynamic binding makes it harder to evolve code without breaking subclasses.
- Performance: Overhead of method look-up at run-time.

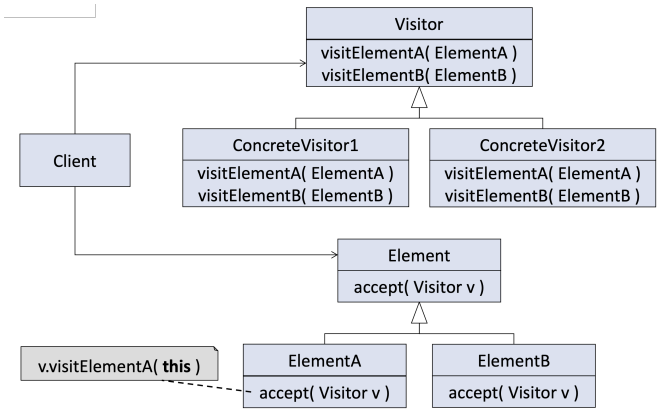
The state pattern is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



For instance, we could have a `Movie` class as context with a `getCharge()` method that calls `_price.getCharge()`, which behaves differently for `RegularPrice`

and `ChildPrice` (both implementing the interface `Price`, which corresponds to the abstract state).

In some cases, it is useful to select an operation based on the dynamic type of the receiver object and of the arguments. This can be implemented with the visitor pattern:



The visitors define (possibly) different behaviors based on the object that is passed to them. The clients don't longer select a proper version of the method to call, but the objects themselves. This is called double dispatch.

Part II

Static Analysis

Chapter 4

Foundations

We cannot build an automatic analyzer that takes as input an arbitrary program and an arbitrary property and when it returns "yes", it is certain that the property holds (soundness) and when it returns "no", then it is certain that it doesn't hold (precision). However, we can build one that when it returns "no", it is unknown if the property holds or not (and when it returns "yes", it's still certain and the analyzer should return "yes" for as many programs as possible).

In contrast to dynamic analysis (e.g. testing), static analysis is an over-approximation of all possible program behaviors. Testing and dynamic (symbolic) analysis report true bugs, but can miss bugs. Automated verification can report false alarms, but doesn't miss bugs.

Static program analysis can automatically prove interesting properties (e.g. absence of null pointer dereferences, absence of data races, termination, etc...) and automatically find bugs in large scale programs / detect bad patterns. It can run the program without giving a concrete input and doesn't need manual annotations such as loop invariants.

Abstract interpretation is a general theory of how to do approximation systematically and a very useful thinking framework that relates the concrete with the abstract, the infinite with the finite. It consists of three steps:

1. Select / define an abstract domain (based on the type of properties you want to prove)
2. Define abstract semantics for the language w.r.t. to the domain (involves defining abstract transformers, i.e. the effect of statements / expressions on the abstract domain) and prove soundness w.r.t. concrete semantics.
3. Iterate abstract transformers over the abstract domain until we reach a fixed point. The fixed point is the over-approximation of the program.

It is desirable to be both sound (whenever a property can be proven, it holds) and precise (whenever a program has a property, we can proof it).

Chapter 5

Abstract Interpretation

5.1 Mathematical Concepts

5.1.1 Structures

A partial order is a binary relation $\sqsubseteq \subseteq L \times L$ on a set L with these properties:

- Reflexive: $\forall p \in L : p \sqsubseteq p$
- Transitive: $\forall p, q, r \in L : (p \sqsubseteq q \wedge q \sqsubseteq r) \implies p \sqsubseteq r$
- Anti-symmetric: $\forall p, q \in L : (p \sqsubseteq q \wedge q \sqsubseteq p) \implies p = q$

Intuitively, $p \sqsubseteq q$ means $p \implies q$ (p is "more precise" than q, that is p represents fewer concrete states than q).

A poset $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$, e.g. $(\mathcal{P}(L), \subseteq)$ is a poset, where $\mathcal{P}$ denotes the powerset. Posets can be visualized by a Hasse diagram where two nodes a, b are connected iff $a \sqsubset b$ and there is no c s.t. $a \sqsubset c \sqsubset b$.

Given a poset $(L, \sqsubseteq)$, an element $\bot \in L$ is called the least element if it is smaller than all other elements of the poset: $\forall p \in L : \bot \sqsubseteq p$. The greatest element is an element $\top$ if $\forall p \in L : p \sqsubseteq \top$. The least / greatest elements may not exist, but if they do they are unique.

Given a poset $(L, \sqsubseteq)$ and $Y \subseteq L$: $u \in L$ is an upper bound of Y if $\forall p \in Y : p \sqsubseteq u$ and $l \in L$ is a lower bound of Y if $\forall p \in Y : l \sqsubseteq p$. The bounds for Y may not exist. $\sqcup Y \in L$ is a least upper bound of Y if $\sqcup Y$ is an upper bound of Y and $\sqcup Y \sqsubseteq u$ whenever u is another upper bound of Y. $\sqcap Y \in L$ is a greatest lower bound of Y if $\sqcap Y$ is a lower bound of Y and $l \sqsubseteq \sqcap Y$ whenever l is another lower bound of Y. A common notation for $\sqcup\{p, q\}$ / $\sqcap\{p, q\}$ is $p \sqcup q$ / $p \sqcap q$. Whenever we have two abstract elements A and B, we can join them to produce their (least) upper bound, for which we also use the notation $A \sqcup B$. This is e.g. used for two program states at the same label.

We can also meet them, denoted $A \sqcap B$, to produce their (greatest) lower bound.

A complete lattice $(L, \sqsubseteq, \sqcup)$ is a poset where $\sqcup Y$ and $\sqcap Y$ exist for any $Y \subseteq L$. For example, for a set L, $(\mathcal{P}(L), \subseteq, \cup, \cap)$ is a complete lattice. The interval or sign domain is also a complete lattice.

Domains

The interval domain is an example of a non-relational domain that doesn't explicitly keep the relationship between variables. On the other hand, octagon and polyhedra are relational domains where the relationship is kept. In the octagon domain, constraints are of the following form: $\pm x \pm y \leq c$ and an abstract state is a map from labels to conjunction of constraints.

In the polyhedra domain, constraints are of the form $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \leq c$ and an abstract state is again a map from labels to conjunctions.

5.1.2 Functions

A function $f : A \to B$ between two posets $(A, \sqsubseteq)$ and $(B, \leq)$ is increasing (monotone) iff:

$$\forall a, b \in A : a \sqsubseteq b \implies f(a) \leq f(b)$$

If the function is between elements in the same poset ($f : A \to A$), this becomes: $\forall a, b \in A : a \sqsubseteq b \implies f(a) \sqsubseteq f(b)$.

For a poset $(L, \sqsubseteq)$, function $f : L \to L$ and element $x \in L$:

- x is a fixed point iff $f(x) = x$
- x is a post-fixedpoint iff $f(x) \sqsubseteq x$

$\mathrm{Fix}(f)$ denotes the set of all fixed points and $\mathrm{Red}(f)$ the set of all post-fixedpoints. We say that $\mathrm{lfp}^{\sqsubseteq} f \in L$ is a least fixed point of f if:

- $\mathrm{lfp}^{\sqsubseteq} f \in L$ is a fixed point
- It is the least fixed point: $\forall a \in L : a = f(a) \implies \mathrm{lfp}^{\sqsubseteq} f \sqsubseteq a$

The least fixed point may not exist.

Tarski's fixed point theorem states: If $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice and $f : L \to L$ is a monotone function, then $\mathrm{lfp}^{\sqsubseteq} f$ exists and $\mathrm{lfp}^{\sqsubseteq} f = \sqcap \mathrm{Red}(f) \in \mathrm{Fix}(f)$.

For a poset $(L, \sqsubseteq)$, a function $f : L \to L$, an element $a \in L$, the iterates of the function from a are $f^0(a), f^1(a), f^2(a), \ldots$ where $f^{n+1}(a) = f(f^n(a))$. $f^0(a) = a$ and in program analysis, a is usually $\bot$. Given a poset of finite height, a least element $\bot$ and a monotone f, the iterates $f^0(\bot), f^1(\bot), f^2(\bot), \ldots$ form an increasing sequence which eventually stabilizes for some $n \in \mathbb{N}$,

that is: $f^n(\bot) = f^{n+1}(\bot)$ and it holds that: $\text{lfp}^{\sqsubseteq} f = f^n(\bot)$. This gives us therefore an easy algorithm for computing the least fixed point.

5.1.3 Approximating Functions

We denote with $[\![P]\!]$ the set of reachable states of a program P and let the function F (where I is an initial set of states and $\rightarrow$ is the transition relation between states) be defined as:

$$F(S) = I \cup \{c' | c \in S \wedge c \rightarrow c'\}$$

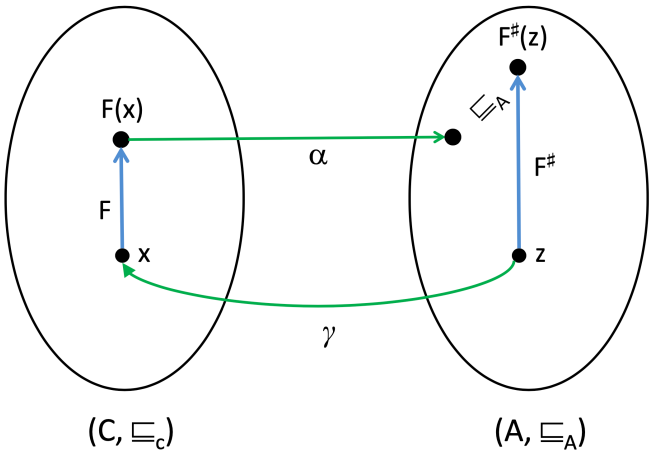
$[\![P]\!]$ is a fixed point of F, $F([\![P]\!]) = [\![P]\!]$ and is in fact the least fixed point of F.

In static program analysis, we want to define a function $F^\#$ such that $F^\#$ approximates F. This is typically done manually and can be tricky, but is only done once and for a programming language. We can then use existing theorems that state that the least fixed point of $F^\#$, e.g. some V, approximates the least fixed point of F, e.g. $[\![P]\!]$. Finally, we can automatically compute a fixed point of $F^\#$, i.e. a V with $F^\#(V) = V$.

Given two functions $F : C \rightarrow C$ and $F^\# : C \rightarrow C$, $F^\#$ approximates F iff: $\forall x \in C : F(x) \sqsubseteq_C F^\#(x)$.

If we have $F : C \rightarrow C$ and $F^\# : A \rightarrow A$, we need to connect the concrete C and the abstract A. This is done with two functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$, where α is the abstraction function and γ is the concretization function. If we know that α and γ form a Galois connection, we can use the following definition of approximation:

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$$



This equation says that if we have some function in the abstract that approximates the concrete function, for any abstract element, concretizing it, applying the concrete function and abstracting back is less than applying the function in the abstract directly.

The least precise approximation (which is always sound but too imprecise) is $F^{\#}(z) = \top$. The most precise approximation (and therefore the best abstract function) is $F^{\#}(z) = \alpha(F(\gamma(z)))$. However, we often cannot implement such a function algorithmically and want to come up with a $F^{\#}$ that has the same behavior but with a different implementation. Any such $F^{\#}$ is referred to as the best transformer.

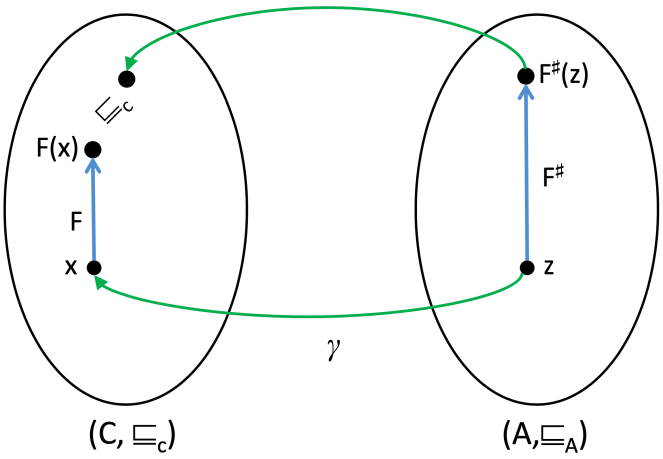
The least fixed point approximation theorem states that if we have:

1. Monotone functions $F : C \to C$ and $F^{\#} : A \to A$
2. $\alpha : C \to A$ and $\gamma : A \to C$ forming a Galois connection
3. $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^{\#}(z)$ ($F^{\#}$ approximates F)

Then $\alpha(\text{lfp}(F)) \sqsubseteq_A \text{lfp}(F^{\#})$. Therefore, once we have proven the 3 premises (which is usually done manually), we can automatically compute a least fixed point in the abstract and be sure that our result is sound.

If α and γ do not form a Galois connection, we can use the following approximation definition:

$$\forall z \in A : F(\gamma(z)) \sqsubseteq_C \gamma(F^{\#}(z))$$



Then the three premises become:

1. Monotone functions $F : C \to C$ and $F^{\#} : A \to A$

2. $\gamma : A \to C$ is monotone
3. $\forall z \in A : F(\gamma(z)) \sqsubseteq_C \gamma(F^\#(z))$ ($F^\#$ approximates F)

And we have if those hold: $\text{lfp}(F) \sqsubseteq_C \gamma(\text{lfp}(F^\#))$

$F^\#$ is to be defined for the particular abstract domain A that we work with (e.g. sign, parity, interval, octagon, polyhedra, etc...). Commonly, we keep a map from every label (program counter) in the program to an abstract element in A and $F^\#$ simply updates the mapping from labels to abstract elements. Or more formally, $F^\# : (\text{Lab} \to A) \to (\text{Lab} \to A)$ with:

$$F^\# = \begin{cases} \top, & \text{if } l \text{ is initial label} \\ \sqcup_{(l',\text{action},l)} [\![\text{action}]\!](m(l')), & \text{otherwise} \end{cases}$$

Where $[\![\text{action}]\!] : A \to A$ captures the effect of a language statement on the abstract domain A (often called abstract transformer). It is therefore sufficient to define $[\![\text{action}]\!]$. (l', action, l) is the set of source labels / actions that reach a destination label (i.e. an edge in the control flow graph). Multiple actions can reach a destination label:

```
foo (int i) {

1: int x := 5;
2: int y := 7;

3: if (0 ≤ i) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }
7:}
```

```
Actions:

(1, x := 5, 2)
(2, y := 7, 3)
(3, 0 ≤ i, 4)
(3, 0 > i, 7)
(4, y = y + 1, 5)
(5, i := i - 1, 6)
(6, goto 3, 3)
```

Multiple (two) actions reach label 3

action can be assignment, skip but also a boolean expression (in a conditional) that is fully evaluated.

If we have a complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ and a monotone function F, if the height is infinite or the computation of the iterates of F takes too long, we need to approximate the least fixed point of F, i.e. we need to find a way to compute an element A such that $\text{lfp}^\sqsubseteq F \sqsubseteq A$. An operator $\triangledown : L \times L \to L$ is called a widening operator if:

- $\forall a, b \in L : a \sqcup b \sqsubseteq a \triangledown b$ (widening approximates the join)
- If $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \ldots \sqsubseteq x^n$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \ldots \sqsubseteq y^n$ stabilizes after a finite number of steps where $y^0 = x^0$ and $\forall i \geq 0 : y^{i+1} = y^i \triangledown x^{i+1}$

Widening is completely independent of the function F, it is an operator defined for the particular domain. If L is a complete lattice, $\triangledown : L \times L \to L$

and $F : L \to L$ is monotone, then the sequence $y^0 = \bot$, $y^1 = y^0 \triangledown F(y^0)$, $y^2 = y^1 \triangledown F(y^1)$, $\ldots$, $y^n = y^{n-1} \triangledown F(y^{n-1})$ will stabilize after a finite number of steps with y^n being a post-fixedpoint of F. By Tarski's theorem, we know that a post-fixedpoint is above the least fixed point and it follows that $\text{lfp}^{\sqsubseteq} F \sqsubseteq y^n$.

Instead of the sequential iteration to compute a fixed point, there's also chaotic (asynchronous) iteration:

```
x1 := ⊥;  x2 = ⊥; ...; xn = ⊥;
W := {1,...,n};

while (W ≠ {}) do {
  ℓ := removeLabel(W);
  prevℓ := xℓ;
  xℓ := prevℓ ▽ fℓ(x1,...,xn) ;
  if (xℓ ≠ prevℓ)
     W := W ∪ influence(ℓ);
}
```

- W is the worklist , a set of labels left to be processed
- influence(ℓ) returns the set of labels where the value at those labels is influenced by the result at ℓ
- Re-compute only when necessary, thanks to influence (ℓ)
- Asynchronous computation can be parallelized

5.2 Interval Analysis

Our starting point is a domain where each element of the domain is a set of states. The domain of states is a complete lattice with $(\mathcal{P}(\sum), \subseteq, \cup, \cap, \emptyset, \sum)$ where $\sum = \text{Lab} \times \text{Store}$.

The interval domain abstracts the set of states into a map which captures the range of values that a variable can take and forms a complete lattice of infinite height (with the top element $[-\infty, \infty]$). Formally, let the interval domain be $(L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, [-\infty, \infty])$ with $L^i = \{[x, y] | x, y \in \mathbb{Z}^\infty, x \leq y\} \cup \{\bot_i\}$ and $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, \infty\}$ We have:

- $[a, b] \sqsubseteq [c, d]$ if $c \leq a$ and $b \leq d$
- $[a, b] \sqcup_i [c, d] = [\min(\{a, c\}), \max(\{b, d\})]$
- $[a, b] \sqcap_i [c, d] = \text{meet}(\max(\{a, c\}), \min(\{b, d\}))$

Where $meet(a, b)$ returns $[a, b]$ if $a \leq b$, $\bot_i$ otherwise.

For programs, we use the domain Lab $\to$ (Var $\to L^i$) (which is also a complete lattice). The operators of L^i, i.e. $\sqsubseteq_i$, $\sqcup_i$ and $\sqcap_i$ are lifted directly to the domains Var $\to L^i$ and Lab $\to$ (Var $\to L^i$) (e.g. for $\sqsubseteq_i$ on Var $\to L^i$, $\forall x_i : a_i \sqsubseteq_i a'_i$ has to hold and on Lab $\to$ (Var $\to L^i$), the lifted definition has to hold for all labels).

For the abstract transformers, we have:

- $[\![x := a]\!]_i(m) = m[x \to v]$, where $\langle a, m \rangle \Downarrow_i v$ (which says that given a map m, the expression a evaluates to a value $v \in L^i$). The operational

semantics rules for expression evaluation are as before, but any constant is abstracted to an element in L^i and the operators $+$, $-$ and $*$ are redefined for the interval domain. For instance, adding $\bot_i$ to anything produces $\bot_i$ and when both operands aren't $\bot_i$, $[x, y] + [z, q] = [x + z, y + q]$.

- $[\![b]\!]_i(m)$ for a boolean expression b. For instance, if b is of the form $a_1 \leq b$, we have: $[l_1, u_1] \leq [l_2, u_2] = ([l_1, u_1] \sqcap_i [-\infty, u_2], [l_1, \infty] \sqcap_i [l_2, u_2])$. Furthermore, $[\![b_1 \vee b_2]\!]_i = [\![b_1]\!] \sqcup_i [\![b_2]\!]$ and $[\![b_1 \wedge b_2]\!]_i = [\![b_1]\!] \sqcap_i [\![b_2]\!]$.

We can define the widening operator $\triangledown_i : L^i \times L^i \to L^i$ for the intervals with $[a, b] \triangledown_i [c, d] = [e, f]$ where:

- If $c < a$, then $e = -\infty$, else $e = a$
- If $d > b$, then $f = \infty$, else $f = b$

If one operand is $\bot$, the result is the other operand.

5.3 Alias Analysis

Pointer and alias analysis is fundamental for reasoning about heap manipulating programs. We have:

- Objs: The set of all possible objects
- PtrVal $=$ Objs $\cup \{\text{null}\}$
- $\rho \in$ PrimEnv : Var $\to \mathbb{Z}$
- $r \in$ PtrEnv : PtrVar $\to$ PtrVal
- $h \in$ Heap : Objs $\to$ (Field $\to \{\text{PtrVal} \cup \mathbb{Z}\}$)

A store is now $\sigma = \langle \rho, r, h \rangle \in$ Store $=$ PrimEnv $\times$ PtrEnv $\times$ Heap (before, it was only ρ) and as before, $\sum =$ Lab $\times$ Store.

Two pointers p and q are aliases if they point to the same object. (p, A) means p holds the address of object A. If (p, A) and (r, A), p and r are aliases.

Because a program can create an unbounded number of objects, we need to use abstraction. One abstraction are allocation sites: The heap is divided into a fixed partition based on allocation site (the statement label). All objects allocated at the same program point (label) get represented by a single "abstract object". Instead of using only the allocation site, the calling context can also be included (useful for library frameworks where we need to know where the library was called from). For allocation sites, the abstract objects are defined as

$$\text{AbsObj} = \{l \mid \text{statement is p:=alloc}^l\}$$

Pointer analysis can be flow sensitive which respects the program control flow. There is a separate set of points-to-pairs for every program point and the set at a program point represent possible may-aliases on some path from entry to the program point. On the other hand, with flow insensitive analysis, we assume all execution orders are possible and abstract away the order between statements. This can be good for concurrency.

Once we have performed the pointer analysis, it's trivial to compute the alias analysis. A function points-to(p) returns the set of all abstract objects that a pointer p can point to and two pointers p and q may alias if points-to$(p) \cap$ points-to$(q) \neq \emptyset$.

5.3.1 Flow Sensitive Analysis

The abstract domain is:

$$\text{Labs} \to ((\text{PtrVar} \to \mathcal{P}(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \to \mathcal{P}(\text{AbsObj})))$$

Because the lattice is of finite height, no widening is needed. One example element in the domain could be: $1 \to (p \to \{a_5, a_{10}\}, a_5.f \to \{a_6, a_9\})$. $\sqsubseteq, \sqcup, \sqcap, \bot, \top$ are based on $\subseteq, \cup, \cap$, lifted appropriately.

The abstract transformer for pointer heap store ($p.f := q^l$) has to apply union on the content of $A.f$ (if $p \to \{A\}$) and q, which is called weak update.

5.3.2 Flow Insensitive Analysis

The abstract domain is:

$$(\text{PtrVar} \to \mathcal{P}(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \to \mathcal{P}(\text{AbsObj}))$$

No information per label is kept. Therefore, if we have a program of the following form:

```
p := alloc // A1
q := alloc // A2
if p = q then
  z := p
else
  z := q
```

We remove the if / else and the analysis outputs $p \to \{A1\}, q \to \{A2\}, z \to \{A1, A2\}$. In contrast to flow sensitive analysis, we therefore can't conclude that z and p don't alias.

5.4 Proving Determinism

Proving arbitrary programs deterministic is hard, but we can prove conflict-freedom, which is a stronger property that implies determinism. To do that, we compute all reachable abstract states and check if they are conflict-free.

If we have a parallel / concurrent program, we first compute the abstract states by analyzing the program sequentially. Then, for every pair of abstract states σ_i, σ_k we check if st_i in σ_i and st_k in σ_k may happen in parallel. If so, we combine (meet) σ_i, σ_k with the constraint that $\text{tid}_i \neq \text{tid}_k$. If $\text{ALoc}(\text{st}_i)$ and $\text{ALoc}(\text{st}_k)$ may equal in A, we output that the program may be non-deterministic. If this isn't the case for any pair, the program is deterministic.

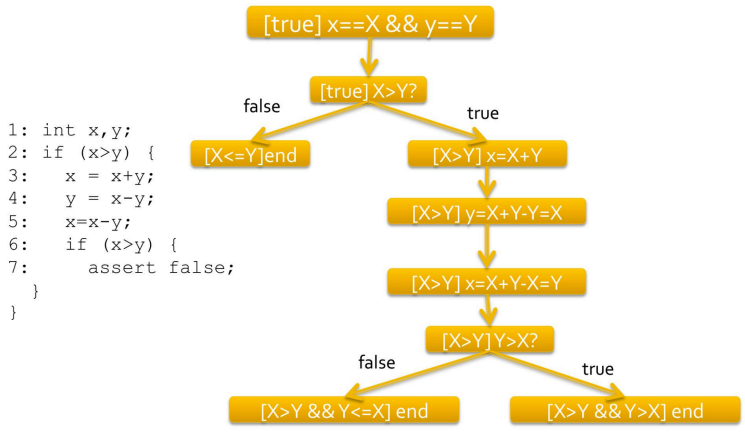
However, the heap abstraction can be imprecise as aliasing information inside reference arrays is lost (e.g. for arrays `double G[][]` or `Object A[]`. If we assume (with our domain-specific knowledge), that references inside array are distinct, the abstraction can be more precise.

Chapter 6

Symbolic Execution

Symbolic execution is an over- and under-approximation of the program behaviors. It sits between testing and static analysis and is completely automatic. The goal is to explore as many program executions as possible, but it has false negatives (may miss program executions / errors).

In classic symbolic execution, a symbolic value is associated with each variable instead of a concrete value. The program is then run with the symbolic values, obtaining a big constraint formula. At any program point, a constraint (SMT) solver can be invoked to find satisfying assignments, which can be used to get real concrete inputs for which the program reaches a program point.



Not all formulas are decidable, therefore the SMT solver can't always find satisfying assignments (even if they exist).

At any point during program execution, symbolic execution keeps a sym-

bolic store and a path constraint and the symbolic state is the conjunction of these two formulas. The values of variables is given by a function $\sigma_s \in \text{SymStore} = \text{Var} \rightarrow \text{Sym}$. Arithmetic expression evaluation simply manipulates the symbolic values, e.g. for $\sigma_s : x \rightarrow x_0, y \rightarrow y_0$, $z = x + y$ will produce the symbolic store $\sigma_s : x \rightarrow x_0, y \rightarrow y_0, z \rightarrow x_0 + y_0$. The path constraint records the history of all branches taken so far and is simply a formula. Evaluation of conditional only affects the path constraint. We can avoid further exploring a path if we already know that the path constraint is unsatisfiable.

Handling unbounded loops is a limitation of symbolic execution as it will keep running forever. In practice, we can provide some loop bound (which is an under-approximation).

One key optimization for symbolic execution is to keep a map / cache of formulas to a satisfying assignment for the formula and accessing the cache before calling the SMT solver (as the analysis will invoke the SMT solver often with similar formulas).

6.1 Concolic Execution

Concolic execution tries to solve the problem that SMT solvers may not always be able to find satisfying assignments. It combines both symbolic and concrete (normal) execution and is ultimately a search-based testing technique with the goal of improving branch coverage in testing. The basic idea is to have the concrete execution drive the symbolic execution. The program runs as usual (with some input), but also maintains the usual symbolic information in addition. Whenever the concrete execution needs to obtain new inputs (in order to explore some paths that wouldn't be explored with the current concrete values), it uses the symbolic information to obtain new inputs. Furthermore, concolic execution can simplify unsolvable constraints by plugging in concrete values. For instance, if we have a constraint like `x0 = y0 * y0` and assume our solver is unable to find an assignment, concolic execution can plug in the current concrete value of y to get the value x. But as soon as we plug in concrete values, we lose symbolic information (and therefore may not be able to explore all following paths).

Concolic execution can also be used when we are calling OS code / native libraries that we don't want to execute symbolically. We simply can call this function with the concrete values and continue execution. But because of bounding loops / recursion, it can be potentially unsound.

Part III

Testing

Chapter 7

Foundations

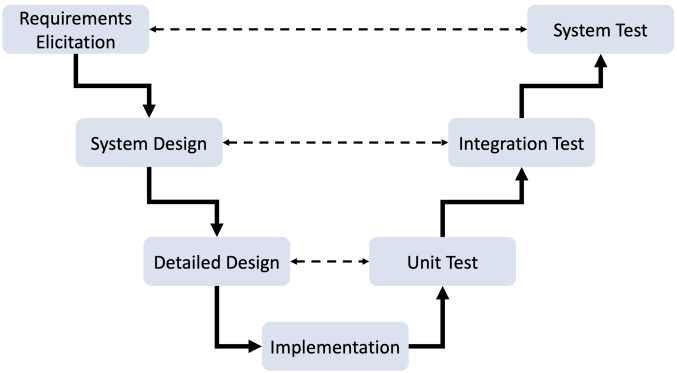
To increase software reliability, there are three main techniques:

- **Fault Avoidance**: Detect faults statically without actually executing the program. This includes development methodologies, reviews and program verification.
- **Fault Detection**: Detect faults by executing the program, which includes testing.
- **Fault Tolerance**: Recover from faults at runtime (e.g. transactions), which includes adding redundancy (e.g. n-version programming).

An error is a deviation of the observed behavior from the required (desired) behavior (which can be functional requirements, e.g. user-acceptance testing or nonfunctional requirements, e.g. performance testing). Testing is a process of executing a program with the intent of finding an error. A successful test is a test that finds errors. But testing can only show the presence of bugs, not their absence because it's impossible to completely test any nontrivial module / system.

7.1 Test Stages

There are different test stages, corresponding to a respective design phase:



When creating tests, we have a test driver that applies test cases to the unit under test (UUT), including setup and cleanup. The UUT uses tests stubs, which are partial, temporary implementations of a component used by it. They simulate the activity of a missing component by answering to the calling sequence of the UUT and returning fake data. A test oracle (or just oracle) is a mechanism for determining whether a test has passed or failed, e.g. an assertion that a method returns a specific value.

The differences between the individual test stages are:

- **Unit Testing**: Individual subsystems (collection of classes) are tested, the goal is to confirm that the subsystem is correctly coded and carries out the intended functionality. For reasonable test coverage, the method has to be tested with several inputs (valid / invalid input, different paths through the method). Parametrized test methods take arguments for the test data and therefore decouple the test driver (logic) from the test data and avoid boiler-plate code. The data can be specified as values, ranges or random values. However, this requires generic test oracles.
- **Integration Testing**: Groups of subsystems and eventually the entire system are tested, the goal is to test interfaces between subsystems. The integration testing strategy determines the order in which the subsystems are selected for testing and integration. There's big-bang integration (non-incremental; all components or modules are integrated simultaneously, after which everything is tested as a whole), bottom-up integration (the lowest modules in the dependency graph / call hierarchy are tested / integrated first, until the whole system is tested; doesn't require stubs but testing of the whole system is done only in the end) or top-down integration (testing starts from the top of the call hierarchy, stubs are used for the underlying modules; allows early integration tests of the whole system, but developing stubs can be time-consuming).

- **System Testing**: The whole system is tested with the goal to determine if it meets the (functional and non-functional) requirements. System testing is further divided into the following stages:
  - Functional Test: Testing functional requirements. The system is treated as a black box and test cases are designed from requirements specification (based on use cases). Test cases describe input data, flow of events, and results to check.
  - Performance Test: Testing non-functional requirements
  - Acceptance Test: Testing the client's understanding of the requirements. It is performed by the client, not by the developer. In an alpha test, a client uses the software at the developer's site in a controlled setting, with the developer ready to fix bugs. In a beta test at the client's site, the software gets a realistic workout in the target environment.
  - Installation Test: Testing the user environment

Test cases should be re-executed after every change (when possible automatic). Regression testing is testing that everything that used to work still works after changes are made to the system. The "eight rules of testing" are:

1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that reduces the time it takes to find bugs
3. Run your tests frequently–every test at least once a day
4. When you get a bug report, start by writing a unit test that exposes the bug
5. Better to write and run incomplete tests than not run complete tests
6. Concentrate your tests on boundary conditions
7. Do not forget to test exceptions raised when things are expected to go wrong
8. Do not let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

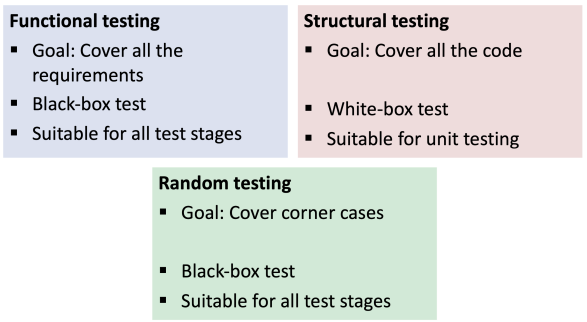
Because programmers have a hard time believing they made a mistake and an interest in not finding mistakes, they often stick to the data that makes the program work. Therefore, testing is done best by independent testers. Usually, system tests (with the exception of the acceptance test) are performed by an independent test team, whereas integration / unit tests are performed by the programmer, because they require detailed code knowledge and this allows immediate bug fixing.

7.2 Test Strategies

First, one should select what will be tested (what parts / aspects of the system). Then, the test strategy (integration strategy, how is test data determined) should be defined. Based on this, the test cases (test data, how is the test carried out?) should be defined. Finally, the test oracle (what are the expected results?) should be created.

There are multiple possible strategies:

1. **Exhaustive Testing**: Check UUT for all possible inputs (most of the time not feasible).
2. **Random Testing**: Select test data uniformly. This avoids designer/tester bias and tests robustness (especially handling of invalid input / unusual actions). However, it treats all inputs as equally valuable (which often isn't the case).
3. **Functional Testing**: Here, requirements knowledge is used to determine the different test cases (e.g. testing all cases of the specification). The goal is to cover all the requirements. It attempts to find incorrect / missing functions, interface or performance errors. However, it doesn't effectively detect design / coding errors (buffer overflow, memory management, ...) and doesn't reveal errors in the specification.
4. **Structural Testing**: In structural testing, design knowledge about the system structure, algorithms, and data structures is used to determine test cases that exercise a large portion of the code. It focuses on thoroughness and the goal is to cover all the code. However, it's not well suited for system test as it focuses on code and not on requirements, requires design knowledge and thoroughness would lead to highly-redundant test for the system test.



Chapter 8

Functional Testing

8.1 Partition Testing

In this approach, the input is divided into equivalence classes and each possible input belongs to one equivalence class. The goal is that some classes have a higher density of failures. Test cases are chosen for each equivalence class.

8.2 Selecting Representative Values

After the partitioning of the input values, we need to select concrete values for the test cases for each equivalence class. If the inputs are from a range of valid values, we want values below, within and above the range. If they are from a discrete set of valid values, we want valid / invalid discrete values.

Because a large values tend to occur at boundaries of the input domain (overflows, comparisons, missing emptiness checks, wrong number of iterations), we want to additionally select elements at the ”edge” of each equivalence class (boundary testing), i.e. lower / upper limits and empty sets / collections.

8.3 Combinatorial Testing

Combining equivalence classes / boundary testing leads to many values for each input. There are different approaches to reduce the test cases:

- **Semantic Constraints**: Use problem domain knowledge to remove unnecessary combinations (e.g. for some values of parameters, we may know / assume that the behavior is independent of another parameter).

- **Combinatorial Selection**: Empirical evidence suggests that most errors do not depend on the interaction of many variables. Instead of testing all possible combinations of all inputs, focus on all possible combinations of each pair of inputs (pairwise-combination testing; can be generalized to k-way testing for k-tuples). For n parameters with d values per parameters, the number of test cases grows logarithmically in n and quadratic in d.
- **Random Selection**

Chapter 9

Structural Testing

Detailed design and coding (algorithms / data structures / optimizations) introduce many behaviors that are not present in the requirements and functional testing generally doesn't thoroughly exercise these behaviors.

9.1 Control Flow Testing

A basic block is a sequence of statements such that the code in a basic block has one entry point (no code within it is the destination of a jump instruction) and one exit point (only the last instruction causes the program to execute code in a different basic block). Therefore, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order.

An intraprocedural control flow graph (CFG) of a procedure p is a graph (N, E) where N is the set of basic blocks in p plus designated entry / exit blocks and E contains:

- an edge from a to b with condition c iff the execution of basic block a is succeeded by the execution of basic block b if condition c holds
- an edge $(\text{entry}, a, \text{true})$ if a is the first basic block of p
- edges $(b, \text{exit}, \text{true})$ for each basic block b that ends with a (possibliy implicit) return statement

Intraprocedural CFGs treat method calls as simple statements, but calls invoke different code depending on the dynamic type of the receiver. Testing should cover the possible behaviors. If we regard a dynamically-bound method call as a case distinction on the type of the receiver, branch testing can be applied (but this leads to combinatorial explosion; therefore semantic constraints / combinations testing should be applied).

Exceptions add a control flow edge from the basic block where the exception is thrown to the exit block or the block where the exception is caught. Exceptional control flow (for documented / checked exceptions) should be covered like normal control flow during testing. Unchecked exceptions should only be tested if they are explicitly thrown by the method under test (argument validation, precondition check), not by methods being called (because they can be a defect in method under test or in the called method).

Several coverage metrics can be defined based on the CFG:

9.1.1 Statement Coverage

It measures how much of the CFG is executed by a test suite and is defined as:

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

(or alternatively with basic blocks).

9.1.2 Branch Coverage

An edge (m, n, c) in a CFG is a branch iff there is another edge (m, n', c') with $n \neq n'$. Branch coverage is defined as:

$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

(and 100% if the code has no branches)

Complete branch coverage implies complete statement coverage, however "at least n% branch coverage" doesn't imply "at least n% statement coverage.

9.1.3 Path Coverage

A path is a sequence of nodes $n_1, \ldots, n_k$ such that $n_1 = \text{entry}$, $n_k = \text{exit}$ and there is an edge (n_i, n_{i+1}, c) in the CFG. Path coverage is defined as:

$$\text{Path Coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}}$$

If the number of loop iterations is not known statically, an arbitrarily large number of test cases is needed for complete path coverage.

Complete path coverage implies complete statement coverage and complete branch coverage, however "at least n% path coverage" does not generally imply "at least n% statement coverage" or "at least n% branch coverage".

9.1.4 Loop Coverage

For each loop, zero, one, and more than one consecutive iterations are tested:

$$\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, more iterations}}{\text{Total number of loops * 3}}$$