Rigorous Software Engineering

by dcamenisch

1 Introduction

This document is a summary of the 2023 edition of the lecture *Rigorous Software Engineering* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at `https://github.com/dcamenisch/rse-summary`. This work is published as CC BY-NC-SA.

2 Documentation

Source code alone is not enough to document software engineering projects. Developers need additional information that is difficult to extract from source code, such as possible results of a method, possible side effects of methods, and consistency conditions of data structures. Therefore documentation is key to ensure that software is reliable and maintainable over time.

2.1 What to document

There are two stakeholders, clients and implementors. For clients it is important to document the interface, how to use the code. For stakeholders it is important to document how the code works. For interfaces document the methods and constructors that clients can use to interact with the software, as well as any preconditions and postconditions that must be met. When documenting the implementation for implementors include the algorithms and data structures used in the software, as well as any invariants and assertions that must be maintained.

2.2 How to document

Documentation does not only consist of comments. Using type annotations, modifiers, assertions and effect systems can also be considered documentation. Still, comments are a large part of good documentation. They provide simple, flexible way of documenting interfaces and implementations. A good documentation using comments can look as follows:

```
/**
 * Returns the value to which the
 * specified key is mapped, or
 * {@code null} if this map contains no
 * mapping for the key.
 *
 * @param key the key whose associated
 *        value is to be returned
 * @return the value to which the
 *         specified key is mapped, or
 *         {@code null} if this map contains
 *         no mapping for the key
 * @throws NullPointerException if the
 *         specified key is null and this map
 *         does not permit null keys
 */
V get( Object key );
```

Using diagrams and examples to illustrate complex concepts can help to make the documentation easier to understand. Using consistent formatting and naming conventions can further improve it. Finally, reviewing and updating the documentation regularly to ensure that it remains accurate and up-to-date.

3 Modularity

The idea of modularity is to partition the overall development effort, support independent testing and analysis of components, decouple parts so they can be modified individually and dividing a large system in mind-sized chunks.

3.1 Coupling

Coupling measures interdependence between different modules. A thigh / high coupling means that modules cannot be developed, tested, changed, understood or reused in isolation. Therefore we want low coupling for correct and maintainable software.

3.1.1 Data Coupling

Modules that expose their internal data representation become tightly coupled to their clients (**representation exposure**). It prevents modules from maintaining strong invariant and concurrency requires complex synchronization. Similarly, data representations often include sub-objects, exposing these can lead to unexpected side effects.

One way of preventing this is to restrict the access to data - forcing clients to access the data representation through a narrow interface. Avoid exposure to sub-objects and prevent the leaks of any references to these sub-objects.

Modules get couples by operating on shared data structures (e.g. compiler working on syntax tree). This can be avoided by making the data structure immutable, however changing the data representation remains a problem and having to copy it can lead to run-time and memory overhead.

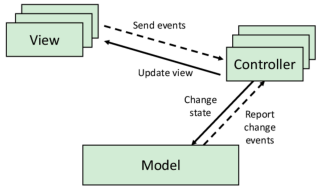
The flyweight pattern is one example that tries to maximize sharing of immutable objects, it is for example used in Java for constant strings.

3.1.2 Procedural Coupling

Modules are coupled to other modules whose methods they call. Callers cannot be reused without callee modules and changing a signature in the callee requires changing the caller.

This can be prevented by moving code and even duplicating functionality to avoid dependency. Another solution could be to change to a event based system. Components may generate events and register for events from other components using a callback. Event generators do not know which components will be affected by their events (loss of control).

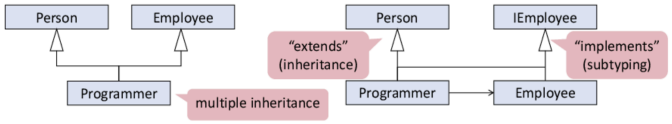
One common event based architecture is Model-View-Controller. The model contains the core functionality and data, the view is responsible for displaying information and the controller handles user inputs. All communication happens via events and the model and the view are decouples through the controller.



3.1.3 Class Coupling

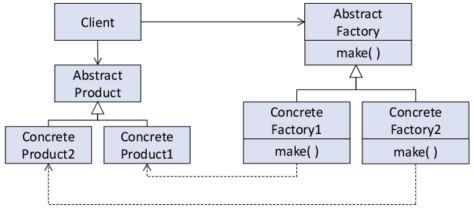
Inheritance couples the subclass to the superclass, changes in the superclass may break the subclass.

To solve this, we can replace (multiple) inheritance by subtyping and delegation.



Using class names in declarations of methods, fields, and local variables couples the client to the used class. To avoid this, one can replace class names by supertypes (interfaces). Using the most general supertype that offers all required operations.

Lastly, allocations couples the client to the instantiated class. We fix this by using dedicated classes called abstract factories to handle allocation.



3.2 Adaption

Changes often erode the structure of the system. Modules can be prepared for change by allowing clients to influence their behavior. This is done by making the module parametric in:

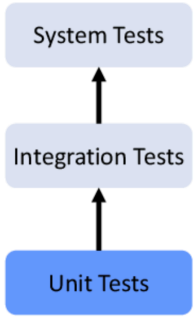
- The value they manipulate
- The data structures they operate on
- The types they operate on
- The algorithms they apply

In object-oriented programs, behaviors can be specialized via overriding and dynamic method binding. Dynamic method binding is a case distinction on the dynamic type of the receiver object.

4 Testing

Testing is the process of executing a program with the intent of finding an error. An error is a deviation of the observed behavior from the required behavior. Testing can only show the presence of bugs and not their absence.

4.1 Test Stages

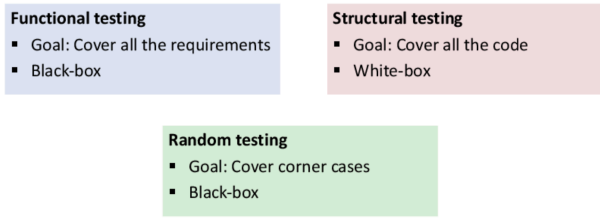


Unit testing is used to test individual subsystems (collection of classes, or single class). To achieve a reasonable test coverage, one has to test each method with several inputs. Parameterized test methods take arguments for test data and help to decouple the test driver (logic) from the test data. They also help to avoid boiler-plate code, they are most useful when test data is generated automatically.

4.2 Test Strategies

There are different strategies to testing:

- Exhaustive testing - check the unit under test for all possible inputs.
- Random testing - select test data uniformly at random. Can be automized but it treats all inputs as equally valuable.
- Functional testing - use requirement knowledge to derive test cases. The goal is to cover all requirements. Does not effectively detect design and coding errors or errors in the specification.
- Structural testing - use design knowledge about the system structure, algorithms, and data structures to derive test cases that exercise a large portion of the code. Focuses on covering all the code. Not well suited for system tests, due to high redundancy.



4.3 Functional Testing

Functional testing black-box tests a unit against its requirements.

4.3.1 Partition Testing

Divide the inputs into equivalence classes and choose test cases for each equivalence class. An example for this would be to divide months into equivalence classes based on the number of days they have.

4.3.2 Selecting Representative Values

After partitioning, we need to select the input values from each class. A large amount of errors tend to occur at the boundaries of the input domain (overflows, comparisons $<$ instead of $\leq$, wrong number of iterations, or missing emptiness checks). Boundary testing selects elements at the edge of each equivalence (in addition to values in the middle).

4.3.3 Combinatorial Testing

Combining equivalence classes and boundary testing can lead to combinatorial explosion. To reduce the amount of test cases, one can use semantic constraints, combinatorial selection or random selection.

Semantic constraints use domain knowledge to remove unnecessary combinations.

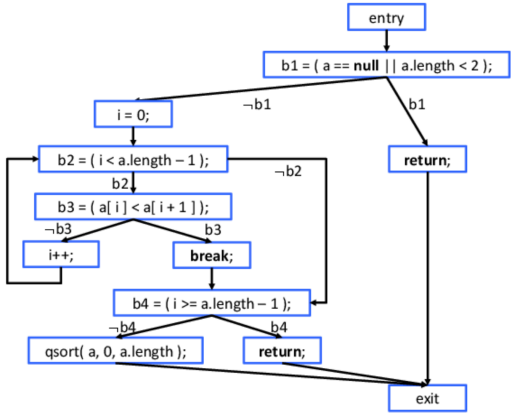
Empirical evidence suggests that most errors do not depend on the interaction of many variables (mostly two or three variables). Therefore it might be useful to only focus on all possible combinations of each pair of inputs.

4.4 Structural Testing

Detailed design and coding introduce many behaviors that are not present in the requirements, this comes down to the choice of data structures, algorithms and optimizations. Functional testing generally does not thoroughly exercise these behaviors. This is where structural testing comes in handy.

Basic blocks are a sequence of statements such that the code has one entry point and one exit point. Whenever the first instruction in a basic block is executed, the rest of the instructions are also executed.

An intraprocedural control flow graph (CFG) of a procedure p is a graph (N, E) where N is the set of basic blocks and E contains the edges from one to another.



Statement coverage assesses the quality of a test suite by measuring how much of the CFG it executes.

$$\text{Statement Coverage} = \frac{\#\text{Executed Statements}}{\#\text{Statements}}$$

Still, 100% statement coverage does not guarantee that we detect all the bugs, since we might not execute all edges. This is why we introduce branch coverage.

$$\text{Branch Coverage} = \frac{\#\text{Executed Branches}}{\#\text{Branches}}$$

Branch coverage leads to more thorough testing than statement coverage and is the most widely-used adequacy criterion in industry. Still, it does not guarantee that there are no bugs. Therefore we introduce path coverage. Path coverage has the idea to test all possible paths through the CFG.

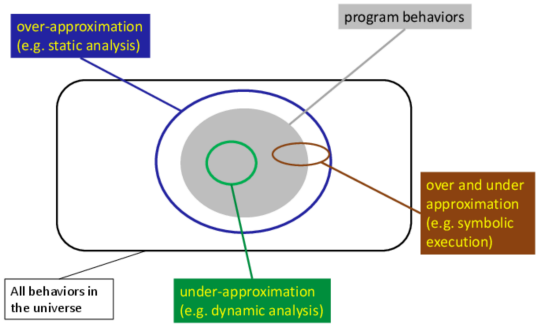
$$\text{Path Coverage} = \frac{\#\text{Executed Paths}}{\#\text{Paths}}$$

However, if for example the number of loop iterations is not known statically, an arbitrary large number of test cases is needed for complete path coverage. This leads to the final idea, loop coverage.

$$\text{Loop Coverage} = \frac{\#\text{Executed Loops w. 0, 1 and >1 iter.}}{\#\text{Loops} * 3}$$

5 Analysis

The goal is to build an automatic analyzer which takes as input an arbitrary program and an arbitrary property such that the analyzer can answer if the property holds or not. But this problem is undecidable, so we have to make some sacrifices. We only want the analyzer to decided if the property holds for sure.



Static program analysis can run the program without giving a concrete input and does not need any manual annotations such as loop invariants.

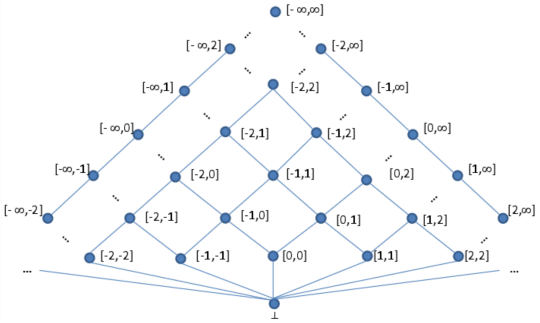
5.1 Abstract Interpretation

Abstract interpretation works by:

1. Select / define an abstract domain
2. Define abstract semantics for the language w.r.t. to the domain
3. Iterate abstract transformers over the abstract domain until a fixed point is reached

It is important to remember that abstract transformers are defined per programming language once and for all, and not per program. A correct abstract transformer should always produce results that are a superset of what a concrete transformer would produce. In general it is easy to be sound and imprecise, being sound and precise is hard.

One example for this type of interpretation uses the interval domain given by:



When we have two abstract elements A and B, we can join them to produce their (least) upper bound, denoted by $A \sqcup B$. For this we have to define the join operation.

With the interval abstraction we can have cases where we cannot reach a fixed point, e.g. loop that always counts up. To fix this we introduce the widening operator, it ensures termination at the expense of precision. If the bound of an interval is increasing, we simply go to ∞ instead of widening the interval multiple times.

5.2 Mathematical Concepts

5.2.1 Structures

A partial order is a binary relation $\sqsubseteq \subseteq L \times L$ on a set L with the properties of being reflexive, transitive and antisymmetric. The intuition is that it captures implications between facts. Later, we will say that if $p \sqsubseteq q$, then p is more precise than q. Given a poset, we can construct a Hasse diagram.

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.

The greatest element $\top$ is defined analogous. The least and 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 $\forall p \in Y : p \sqsubseteq u$. $\bigsqcup Y \in L$ is a least upper bound of Y if it is an upper bound of Y and $\bigsqcup Y \sqsubseteq u$ whenever u is another upper bound of Y. We define the lower bound and greatest lower bound analogously.

A complete lattice $(L, \sqsubseteq, \bigsqcup)$ is a poset where $\bigsqcup Y$ and $\bigsqcap Y$ exist for any $Y \subseteq L$. The interval domain from above is a complete lattice.

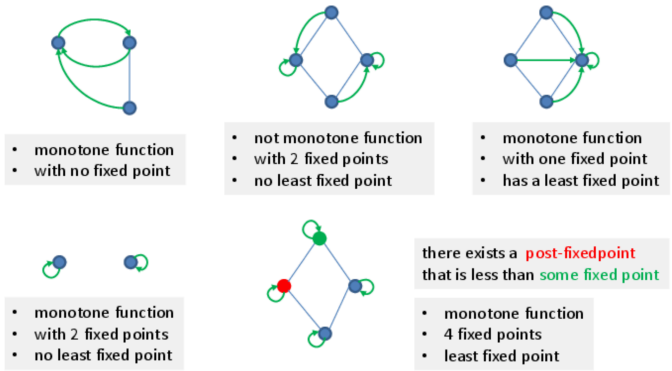
5.2.2 Functions

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

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

For a poset $(A, \sqsubseteq)$, a function $f : A \to A$, and element $a \in A$, a is a fixed point iff $f(a) = a$. Further a is a post-fixedpoint iff $f(a) \sqsubseteq a$. The set of all fixed points is denoted by Fix(f) and the set of all post-fixedpoints is denoted by Red(f).

For a poset $(A, \sqsubseteq)$ and a function $f : A \to A$, we say that $\text{lfp} \sqsubseteq f \in A$ is a least fixed point of f if $\text{lfp} \sqsubseteq f$ is a fixed point and $\forall a \in A : a = f(a) \Rightarrow \text{lfp} \sqsubseteq f \sqsubseteq a$.



If $(A, \sqsubseteq), \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice and $f : A \to A$ is a monotone function, then lfp $\sqsubseteq$ f exists and $\text{lfp} \sqsubseteq f = \bigsqcap \text{Red}(f) \in \text{Fix}(f)$.

Given a poset of finite height, a least element $\bot$, a monotone f. Then the iterates $f^0(\bot), f^1(\bot), f^2(\bot), ...$ form an increasing squenece which eventually stabilizes.

$$\text{lfp} \sqsubseteq f = f^n(\bot)$$

5.2.3 Approximating Functions

Let $[[P]]$ be the set of reachable states of a program P. Let function F be (I is the initial state and $\to$ is the transition relation) :

$$F(S) = I \cup \{c' \mid c \in S \wedge c \to c'\}$$

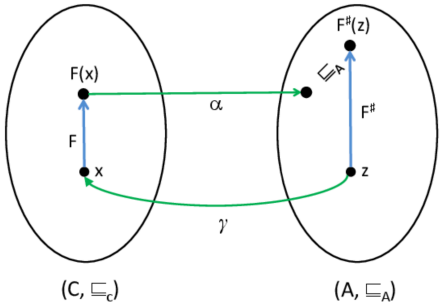
Then, $[[P]]$ is a fixed point of F, in fact it is the least fixed point of F. In static program analysis we want to approximate a programming language. For this, we define a function $F^\#$ that approximates F. Then, using existing theorems, approximate the least fixed point of F by computing the least fixed point of $F^\#$.

A function $F^\# : C \to C$ approximates $F : C \to C$ if:

$$\forall x \in C : F(x) \sqsubseteq_c F^\#$$

If $F : C \to C$ and $F^\# : A \to A$, we need to connect the concrete C and abstract A. We do this via two function $\alpha : C \to A$ (abstraction function) and $\gamma : A \to C$ (concretization function). If we know that α and γ form a Galois Connections, then we can use the following definition of approximation:

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



To approximate F, we can always define $F^\#(z) = \top$. This solution is always sound, however it is too imprecise.

The most precise approximation is given by $F^\#(z) = \alpha(F(\gamma(z)))$. The problem is that we often cannot implement such a $F^\#(z)$. However, we can come up with a $F^\#(z)$ that has the same behavior but a different implementation. Any such $F^\#(z)$ is referred to as the best transformer.

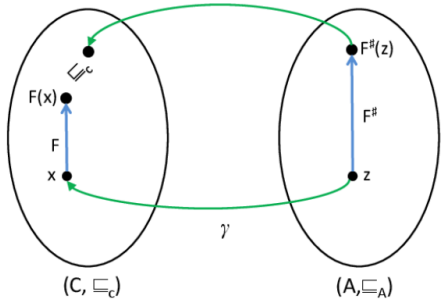
Least Fixed Point Approximation - If we have the following properties:

1. monotone functions $F : C \to C$ and $F^\# : A \to A$
2. $\alpha : C \to A$ and $\gamma : A \to A$ form a Galois Connection
3. $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$ (that is, $F^\#$ approximates F)

Then $\alpha(\text{lfp}(F)) \sqsubseteq_A \text{lfp}(F^\#)$. This is important as it goes from local function approximation to global approximation.

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

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



If we have the following properties:

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

Then $\text{lfp}(F) \sqsubseteq_C \gamma(\text{lfp}(F^\#))$

So what is $F^\#$ then? $F^\#$ is to be defined for the particular abstract domain A. The domain A can be Sign, Parity,

Interval, Polyhedra, and so on. In our setting, we simply keep a map from every label in the program to an abstract element in A. Then $F^\#$ simply updates the mapping from labels to abstract elements.

$$F^\# : (\text{Lab} \to A) \to (\text{Lab} \to A)$$

5.3 Applications of Analysis: Intervals

In this part, we will put these things together to build static analyzers. For this we first select a abstract domain, define the abstract semantics and then iterate the abstract transformer over a program until a fixed point is reached.

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:

$$(\mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma) \qquad \Sigma = \text{Lab} \times \text{Store}$$

5.3.1 Select Abstract Domain

If we are interested in properties that involve the range of values that a variable can take, we can abstract the set of states into a map that captures the range of values each variable can take.

Let the interval domain be:

$$(L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, [-\infty, \infty])$$

We denote $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, \infty\}$ and $L^i = \{[x, y] \,|\, x, y \in \mathbb{Z}^\infty, x \leq y\} \cup \{\bot_i\}$. Further we define a max and min function. Lastly, we can define the operations:

- $[a, b] \sqsubseteq_i [c, d]$ if $c \leq a$ and $b \leq d$
- $[a, b] \sqcup_i [c, d] = [\min(a, c), \max(c, d)]$
- $[a, b] \sqcap_i [c, d] = \text{wdi}[\min(a, c), \max(c, d)]$ where wdi (well-defined interval) returns $[a, b]$ if $a \leq b$

The L^i domain defines intervals, but to apply it we need to take program labels and program variables into account. Therefore, for programs, we use the domain: Lab $\to$ (Var $\to L^i$). This domain is also a complete lattice. The operators are lifted directly to both domains.

$$\alpha_i : \mathcal{P}(\Sigma) \to (\text{Lab} \to (\text{Var} \to L^i))$$

$$\gamma_i : (\text{Lab} \to (\text{Var} \to L^i)) \to \mathcal{P}(\Sigma)$$

Using α_i, we abstract a set of states into a map from program labels to interval ranges for each variable. Using γ_i, we concretize the intervals maps to a set of states.

5.3.2 Define Abstract Semantics

We still need to compute $\alpha_i[[P]]$ or an over approximation of it. We want a function:

$$F^i : (\text{Lab} \to (\text{Var} \to L^i)) \to (\text{Lab} \to (\text{Var} \to L^i))$$

With the property $\alpha_i(\text{lfp F}) \sqsubseteq \text{lfp}F^i$.

Generic Template for $F^\#$:

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

This results in the following F^i which approximates the best transformer but only works on the abstract domain.

$$F^i(m)l = \begin{cases} \lambda v.[-\infty, \infty] & \text{if } l \text{ is initial label} \\ \bigsqcup_{(l', \text{action}, l)} [[\text{action}]]_i(m(l')) & \text{otherwise} \end{cases}$$

(l', action, l) are the edges in the CFG, an example could be (1, x := 5, 2). Next we would need to define all the effects for these.

$F^i(m)_1 = \quad \lambda\, v.\, [-\infty, \infty]$

$F^i(m)_2 = \quad [\![x := 5]\!]_i(m(1))$

$F^i(m)_3 = \quad [\![y := 7]\!]_i(m(2)) \sqcup [\![\text{goto } 3]\!]_i(m(6))$

$F^i(m)_4 = \quad [\![i \geq 0]\!]_i(m(3))$

$F^i(m)_5 = \quad [\![y := y + 1]\!]_i(m(4))$

$F^i(m)_6 = \quad [\![i := i - 1]\!]_i(m(5))$

$F^i(m)_7 = \quad [\![i < 0]\!]_i(m(3))$

5.3.3 Iterate the Transformer

In the final step, we know compute the least fixed point for a given program. However, especially with loops we can encounter the problem where a variable indefinitely changes (counts up or down). To avoid this we have to use widening. The widening operator $\nabla : L \times L \to L$ is defined as:

- $\forall a, b \in L : a \sqcup b \sqsubseteq a \nabla b$
- if $x^0 \sqsubseteq x^1 \sqsubseteq ... \sqsubseteq x^n \sqsubseteq ...$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq ... \sqsubseteq y^n$ stabilizes after a finite number of steps

Note that widening is completely independent of F^i, similar to join it is defined for a specific domain. If ∇, F are monotone then the sequence $y^0 = \bot, y^1 = y^0 \nabla F(y^0), ...$ will stabilize after a finite number of steps n with y^n being a post-fixed point of F.

For the interval domain, we define the widening operator ∇_i as $[a, b] \nabla_i [c, d] = [e, f]$ where $e = -\infty$ if $c < a$ and $f = \infty$ if $b < d$.

Iteration does not have to be in-order, there is also chaotic (asynchronous) iteration.

5.4 Applications of Analysis: Pointers

Pointer and Alias Analysis is fundamental to reasoning about heap manipulation programs. First we define the concrete store:

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

A store is now:

$$\sigma =< p, r, h > \in \text{Store} = \text{PrimEnv} \times \text{PtrEnv} \times \text{Heap}$$

Before the store was only p. As before we have:

$$\Sigma = \text{Lab} \times \text{Store}$$

Aliases are two pointers p and q that point to the same object. A points-to pair (p, A) means that p holds the address of object A. If (p, A) and (r, A) are points-to pairs, then p and r are aliases.

A program can create an unbounded number of objects. Therefore we need to use some abstraction when allocating

a new object. That is, we need some static naming scheme for dynamically allocated objects.

Allocation sites divide the heap into a fixed partition based on the statement label. All objects allocated at the same program point (label) get represented by a single abstract object.

If this is too imprecise, we can also use the calling context. If we use allocation sites (labels), we can now define the abstract objects as:

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

That is, this is just those labels in the program where allocation of an object occurs. Here alloc^l is just the name of the allocation instruction.

5.4.1 Flow Sensitive Pointer Analysis

This type of analysis respects the program control flow. This leads to a separate set of points-to pairs for every program point. Further, the set at a program point represents possible maybe-aliases on some path from entry to the program point.

We use the same step-by-step approach as before. The abstract domain is a complete lattice:

$$\begin{aligned}\text{Labs} \to (\ &(\text{PtrVar} \to \mathcal{P}(\text{AbsObj})) \times \\ &(\text{AbsObj} \times \text{Field} \to \mathcal{P}(\text{AbsObj}))\ )\end{aligned}$$

The abstract domain keeps two maps at every program label. The first one contains a mapping from a pointer variable to a set of abstract objects. The second map contains a mapping from the fields of abstract objects to the set of abstract objects they point to. Since this lattice is of finite height, we will not need widening. $\sqsubseteq, \sqcup, \sqcap, \bot, \top$ are all lifted appropriately.

Using α, we abstract a set of states into the two kinds of maps. Similarly, using γ, we concretize the pointer maps to a set of states.

We now need to define the effect of program statements manipulating pointers on the abstract domain. It can be summarized as:

Statement/Expression	Description
`p = q`	compare two pointers
`p := alloc`$^\ell$	create new object
`p := q`$^\ell$	assign pointers
`p.f := q`$^\ell$	pointer heap store
`p := q.f`$^\ell$	pointer heap load

Let ustake a look at the most tricky one, pointer heap store. Given `p.f := q` and `p` $\to$ {`A`}, `A.f` $\to$ {`B`}, and `q` $\to$ {`C`}. Is `A.f` $\to$ {`C`} the correct result? No, it is not. The correct solution would be `A.f` $\to$ {`B, C`}, this is called weak updates.

5.4.2 Flow Insensitive Pointer Analysis

This type of analysis assumes all execution orders are possible, it abstracts away order between statements. This is good for concurrency and is scalable, but it may be too imprecise.

The abstract domain does not keep information per label, essentially ignoring the control flow of the program.

$$\begin{aligned}&(\text{PtrVar} \to \mathcal{P}(\text{AbsObj})) \times \\ &(\text{AbsObj} \times \text{Field} \to \mathcal{P}(\text{AbsObj}))\end{aligned}$$

To account for the loss of control-flow, all updates have to be weak. This can lead to very imprecise results. To avoid this, we can initialize local variables with `null` instead of $\top$.

5.5 Application of Analysis: Static Concurrency Checker

Another use case is to check if parallel algorithms are deterministic. However, since proving determinism is hard, we prove a stronger property. We prove data-race freedom.

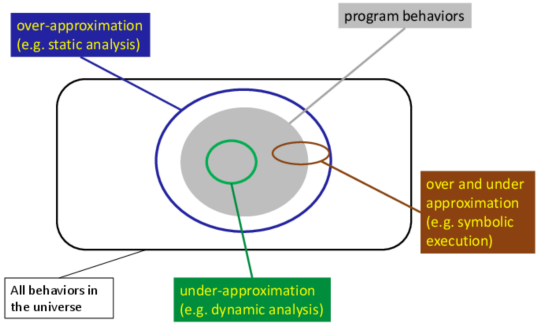
For static analysis there are three sources of unboundedness we have to deal with. These are the heap, the range of array indices, and the number of threads. For the heap we can use flow-insensitive points-to analysis and for the range of array indices we use numerical abstraction.

We first perform sequential analysis on a per thread basis. We then take the cartesian product of these states and filter out all the ones that are not possible. Leaving us with the concrete program states.

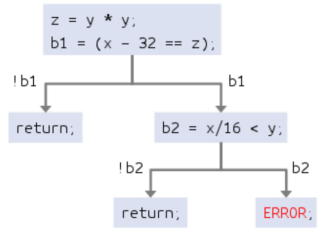
Now, we can check the property we are interested in proving.

6 Symbolic & Concolic Execution

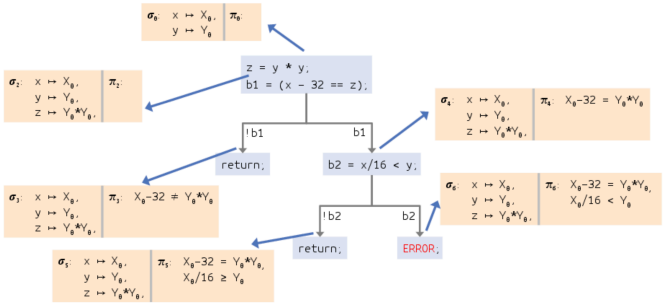
Remember the following diagram:



If we compute the symbolic constraints per path, and then solve these constraints we will have all concrete inputs that explore all paths. This is the core idea of symbolic execution.



A symbolic store σ maps variables to symbolic expressions, which are updated by assign statements. Path conditions π are the conditions under which a path is taken. Combined we have a symbolic state per program point.



Solving the final constraint sets, we can conclude which path will be taken for an input.

Eagerly exploring all paths up-front may include many infeasible paths and can be too time consuming. Symbolic execution engines therefore apply different exploration strategies.

SMT solvers combine SAT-solving with theory reasoning. They can produce models, satisfying assignments. However, there are limits. Many theories are not decidable, e.g. non-linear integers. Let us assume the solver fails to find a model for b_1 or $!b_1$, therefore we will not get concrete test inputs and cannot determine if the branch is infeasible.

Concolic execution combines concrete and symbolic execution. The high-level idea is that the concrete execution drives the symbolic one. Programs are executed with concrete inputs, but additionally maintain a symbolic state. Concrete values are used to simplify path conditions.

A concolic execution may not be able to explore all paths, but any additionally explored path increases the chances of uncovering problems. We use concolic execution to expand sets of manually determined test inputs.

Symbolically executing native code is difficult if not impossible. Concolic execution can help by invoking the functions with concrete values. Still, it might not be able to explore all paths.

Further, one has to consider that symbolic expressions can become large, in particular along deep paths. Optimizations typically employed compilers are subexpression elimination and constant folding.

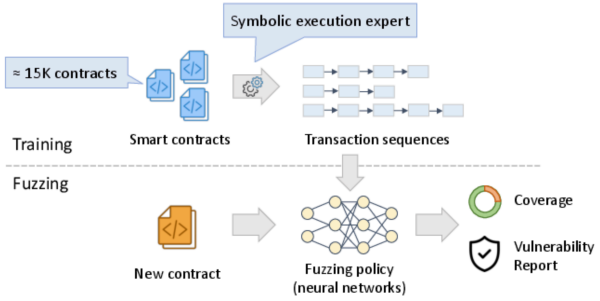
6.1 Fuzzing and Symbolic Execution for Smart Contracts

The idea behind fuzzing is to run programs on abnormal inputs (e.g. randomly generated). This should help finding potential exploits that attackers could use.

We want to find transactions that thoroughly explore the state space. The problem is the exponential number of block states. Both, symbolic execution and fuzzing, have their drawbacks. Imitation learning based fuzzers try to solve their problems.

	Random Fuzzing	Symbolic Execution	ILF
Speed	✓ Fast	✗ Slow	✓ Fast
Inputs	✗ Ineffective	✓ Effective	✓ Effective
Coverage	✗ Low	✗ Low	✓ High

A neural network is used to learn to fuzz from symbolic execution.



This approach performs better than anything comparable.

7 Modeling and Specification

Formal modeling use notations and tools from mathematics to be precise. Formal models enable the use of automatic analysis. They are good at finding ill-formed examples and checking properties.

Alloy is a formal modeling language based on set theory. An alloy model specifies a collection of constraints. The alloy analyzer takes the constraints of a model and tries to find a structure that satisfies them. There are three levels of abstraction in alloy: the object oriented paradigm, set theory, and atoms and relations.

There are four key ideas:

- Everything is a relation. All data types and structures are relations. A key operator is dot join.
- Non-specialized logic. No special constructs for state machines, traces, synchronization, concurrency etc.
- Counterexamples & Scope. Observations about design analysis: most assertions are wrong and most flaws have small counterexamples.
- Analysis by SAT. SAT is easy, so reduce the problem to SAT.

7.1 Logic

Atoms are alloy's primitive entities (indivisible, immutable and uninterpreted). Relations are use to associate atoms with one another. Every value in alloy logic is a relation.

Sets are unary relations, scalars are singleton sets and binary relations are sets of two columns. For these relations it holds that rows are unordered while columns are ordered but unnamed. Further, all relations are first order relations, meaning they cannot contains other relations (no set of sets).

We have the following constants: `none` is the empty set, `univ` is the universal set and `iden` is the identity relation.

7.1.1 Operators

Set Operators

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

Join Operators

. dot join, column that is joind on is left out of the result
[] box join, `a.b.c[d] = d.(a.b.c)`

Restriction and Override

`<:`	*domain restriction*
`:>`	*range restriction*
`++`	*override*

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

```
Name    = {(N0), (N1), (N2)}
Alias   = {(N0), (N1)}
Addr    = {(A0)}
address = {(N0, N1), (N1, N2), (N2, A0)}

address :> Addr  = {(N2, A0)}
Alias <: address = address :> Name  = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

workAddress = {(N0, N1), (N1, A0)}
address ++ workAddress = {(N0, N1), (N1, A0), (N2, A0)}
```

```
m' = m ++ (k -> v)
update map m with key-value pair (k, v)
```

Unary Operators

`~`	*transpose*
`^`	*transitive closure*
`*`	*reflexive transitive closure*

apply only to binary relations

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

```
first = {(N0)}
rest =  {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
```

Boolean Operators

`!`	**not**	*negation*
`&&`	**and**	*conjunction*
`||`	**or**	*disjunction*
`=>`	**implies**	*implication*
	else	*alternative*
`<=>`	**iff**	*bi-implication*

four equivalent constraints:

```
F => G else H

F implies G else H

(F && G) || ((!F) && H)

(F and G) or ((not F) and H)
```

Quantifiers

`all x: e | F`	*F holds for* ***every*** *x in e*
`some x: e | F`	*F holds for* ***at least one*** *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*

```
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

```
some n: Name, a: Address | a in n.address
some name maps to some address — address book not empty

no n: Name | n in n.^address

all n: Name | lone a: Address | a in n.address

all n: Name | no disj a, a': Address |
                (a + a') in n.address
```

If and Let

```
f implies e1 else e2
let x = e | formula
let x = e | expression
```

four equivalent constraints:

```
all n: Name |
  (some n.workAddress implies n.address = n.workAddress
        else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

Cardinalities

`#r`	*number of tuples in r*
`0,1,...`	*integer literal*
`+`	*plus*
`-`	*minus*

`=`	*equals*
`<`	*less than*
`>`	*greater than*
`=<`	*less than or equal to*
`>=`	*greater than or equal to*

```
sum x: e | ie
sum of integer expression ie for all singletons x drawn from e
```

```
all b: Bag | #b.marbles =< 3
all bags have 3 or less marbles

#Marble = sum b: Bag | #b.marbles
the sum of the marbles across all the bags equals the total number of marbles
```

7.2 Static Models

A signature declares a set of atoms (can be thought of like a class) `sig FSObject{}`, extends-clauses declare subset relations `sig File extends FSObject{}`. Like classes, signatures can be `abstract`. Further, they may constrain the cardinality of the declared set by `one, lone, some` (singleton, singleton or empty, none empty set).

Fields declare relations on atoms `sig A { f: e}`. `f` is a binary relation on domain `A` and the range given by `e`. Range expressions may denote multiplicities `one, lone, some, set`. Fields may range over relations, with the relation declaration may including multiplicities on both sides `enrollment: Student set -> one Program`.

Predicates are named, parameterized formulas: `pred p[x1:e1, ..., xn:en]{F}`.

Functions are named, parameterized expressions: `fun f[x1:e1, ..., xn:en]:e {F}`

Facts add constraints that always hold, they express value and structural invariants on the model.

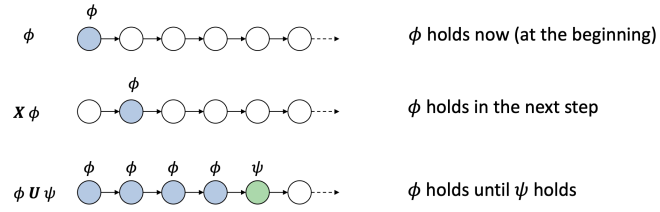
The alloy analyzer can search for structures that satisfy a constraint `C` in a model using `run C`. The existence of structures that satisfy constraints in a model is generally undecidable. The alloy analyzer searches exhaustively for structures up to a given size, therefore the problem becomes finite and decidable (e.g. `run show for 5`).

Exploring by manual inspection can be cumbersome. Alloy supports searching for structures that violate a given property by using `assert a {F}` and `check a scope`. Under-constraint models permit undesirable structures, while over-constraint models exclude desired ones.

7.3 Dynamic Models

Alloy 6 has a built-in notion of time based on Linear Temporal Logic (LTL). This allows us to model things like a game of ping pong.

LTL is boolean logic augmented with two temporal operators `next` (X or $\circ$) and `until` (U or $\cup$).



From this we can derive two more operators `eventually` (F or $\Diamond$) and `always` (G or $\Box$).

Combined with mutable fields, dynamic behavior in Alloy can be summarized like this:

LTL	Alloy
X (or ○)	**after** (or ')
G (or □)	**always**
F (or ◊)	**eventually**
U (or U)	**until**

Mutable fields

```
sig Array {
  length: Int,
  var data: { i: Int | 0 =< i && i < length } -> lone E
}
{ 0 =< length }
```

Time horizon constraints

```
run p
run p for M .. N steps
run p for N steps  // 1 .. N steps
```

```
pred update[ a: Array, i: Int, e: E ] {
  not {i -> e} in a.data
  a.data' = a.data ++ i -> e
}
```

Precondition

Equality, not assignment

Alloy specifications are purely declarative, they describe what is done, not how it is done. Traces define the temporal behavior of the model. A first state is initialized and subsequent states are constraint using LTL operators.

7.4 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 leas one of these structures:

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

A formula F is **valid** is 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 sidesteps this problem by only checking within a given scope, defining a finite bound on the size of the sets in the model.

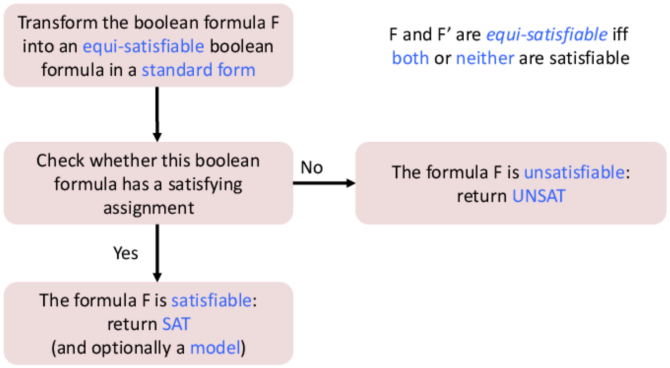
In practice, SAT solvers are extremely efficient at checking consistency (performed using the `run` command). A satisfying assignment can be translated back to relations and then visualized. However, if the SAT solver returns unsatisfiable, there may exist larger structures that satisfy the conditions.

For validity instead of checking directly, the Alloy analyzer checks for invalidity, that is, it looks for counterexamples (performed using the `check` command).

8 SAT/SMT-based Analysis

8.1 SAT Solving Algorithms

The process of SAT solving looks as follows:



As the standard form we use the conjunctive normal form (CNF). A formula F is in CNF iff it is a conjunction of clauses. A clause is a disjunction of literals (a literal being a variable or its negation). A formula in CNF cloud look like this:

$$(p \vee \neg q) \wedge (q \vee r \vee \neg p) \wedge (s \vee p)$$

Simple conversion to CNF works as follows:

1. Rewrite all $A \Rightarrow B$ to $\neg A \vee B$ and $A \Leftrightarrow B$ to $(\neg A \vee B) \wedge (A \vee \neg B)$
2. Push all negations inwards
3. Rewrite all $\neg\neg A$ to A
4. Eliminate $\top$ and $\bot$
5. Distribute disjunctions over conjunctions, e.g. rewrite $A \vee (B \wedge C)$ to $(A \vee B) \wedge (A \vee C)$
6. Remove duplicate clauses and duplicate literals from clauses

The CNF formula can be rewritten until the set is empty or a clause is empty, by using the Davis-Putman-Logemann-Loveland (DPLL) Algorithm. If the set is empty return SAT (and a model M, if a clause is empty return UNSAT. The rules are as follows:

- Pure Literal: If p occurs only positively in F, delete the clauses in which p occurs, $M = M \cup \{p\}$ (similar if p only occurs negatively.
- Unit propagation: If u is a unit clause in F, delete the clauses in which u occurs, update all clauses containing $\neg u$ as a disjunct by removing that disjunct, $M = M \cup \{u\}$.
- Decision: If p occurs both positively and negatively in F:
  - apply the algorithm to $F \wedge p, M \cup \{p\}$: if we get (SAT, M) then return this result
  - otherwise, apply the algorithm to $F \wedge \neg p, M \cup \{\neg p\}$ and return the result

8.2 Encoding Integers into SAT

Alloy supports integers. Bounded integers are encoded using bit-blasting. The idea is that a 32-bit integer is a sequence of 32 individual bits. This is similar to how a circuit defines integer operations. In Alloy the bit width defines the bound for the maximum size of an integer.

8.3 Universal Quantifiers in SMT

When dealing with uninterpreted (user-defined) functions, the idea is to find a candidate model M for the non-quantified part of the formula F and check if M satisfies all the universal quantifiers from F. This is called Model-Based Quantifier Instantiation (MBQI). In general, termination is not guaranteed when using MBQI.

9 Modeling and Analysis of Memory Models using Alloy

Memory models are the interface between programs and their possible executions. They determine which behavior the memory and thus a program is allowed to have. In particular: from which writes can the read operations read from. A memory model would be sequential consistency or x86 Total Store Order (TSO) (a superset of sequential consistency).

9.1 Axiomatic Semantics

A rigorous description of the memory model. Axiomatic semantics defines events, relations, and rules. For x86 TSO we consider the following events: `read, write, fence`.

Program order relations po orders the events in the same thread (given by program). Reads from relations rf associate each read event which exactly one write. Happens before relations hb guarantee that the effect of e_1 are visible to e_2. This leads us to the following axiomatic semantic rules for x86 TSO:

$$\frac{w \xrightarrow{rf} r}{w \xrightarrow{hb} r} \qquad \frac{e_1, e_2 \text{ access same variable}}{e_1 \xrightarrow{hb} e_2 \vee e_2 \xrightarrow{hb} e_1}$$

$$\frac{w, r, w' \text{ access same variable } w \xrightarrow{rf} r \wedge w \xrightarrow{hb} w'}{w \xrightarrow{rf} r \wedge w \xrightarrow{hb} w' \wedge r \xrightarrow{hb} w'}$$

$$\frac{e_1 \xrightarrow{po} e_2 \quad e_1, e_2 \text{ access same variable}}{e_1 \xrightarrow{hb} e_2}$$

$$\frac{e_1 \xrightarrow{po} f \quad f \text{ is a fence}}{e_1 \xrightarrow{hb} f} \qquad \frac{f \xrightarrow{po} e_1 \quad f \text{ is a fence}}{f \xrightarrow{hb} e_1}$$

With these rules we can check if a given execution is allowed, by iteratively applying the rules until no new relations are added. The execution is allowed iff hb is acyclic.

9.2 Axiomatic Semantics in Alloy

The events and axiomatic rules over the po, rf and hb relations can be encoded in the Alloy solver:

```
sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig InitialValue extends Write {
}
```

From there we can add all the rules as constraints.

9.3 Memory Model Evaluation Framework

Based on documentation, we can build a formal Remote Memory Access (RMA) model in Alloy. Then automatically generate tests, together with their allowed executions by the formal RMA model. After running the tests on real RMA networks we can compare the real executions with executions allowed by the model.