0368.4435 Automatic Software Verification March 10, 2015

Lecture 1

Lecturer: Mooly Sagiv *Scribe: Shelly Grossman, Adi Gilboa, Itay Polack*

Lesson Plan

Introduction - what is automatic software verification? Why do we need it? How can we do it?

1. The world is full of bugs, or: what is automatic software verification, and why do we need it?
2. Fundamental limitations of software verifications.
3. Complementary approaches for software verification.

What Is Software Verification, And Why We Need It

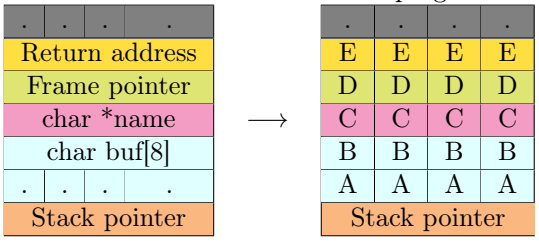
In today's world, software is omnipresent. It can be found not only in computers and similar devices, and not used only by heavy industries or large data centers. It becomes more and more prevalent in commodities such as cars, phones, clocks, TVs, and the list keeps growing. Industries and governments rely on it to automate tasks with higher efficiency and precision, and lesser costs. As a result, the correctness of these programs ever increases.

The risks of a buggy software are not theoretical but real. Only in 2014 several severe security bugs were discovered and publicized in the press (Heartbleed,Shellshock,POODLE). However, security bugs are not the only issue. Trivial bugs can have enormous effect in large systems. The following list of real life stories demonstrates it.

Example - Buffer Overflow
Consider the following program:

```
void print_name(char *name) {
   char buf[8];

    strcpy(buf, name);
    printf("%s\n", buf);
}

int main (int argc, char *argv[]) {
   print_name(argv[1]);
}
```

How will our stack look if we run this program with "AAAABBBBCCCCDDDDEEEE"?



This bug allows us to overwrite any value higher up in the stack. For example, we could change the value of a flag allocated on the stack that controls access grants. We could also change the return

address to the location of our own function, or to a system call, and even provide it with arguments. Even more simply, make the program crash.

Example - Leaks

There are numerous kinds of leaks in programs. Any kind of resource can leak, and in every programming language - memory, processes, connections.

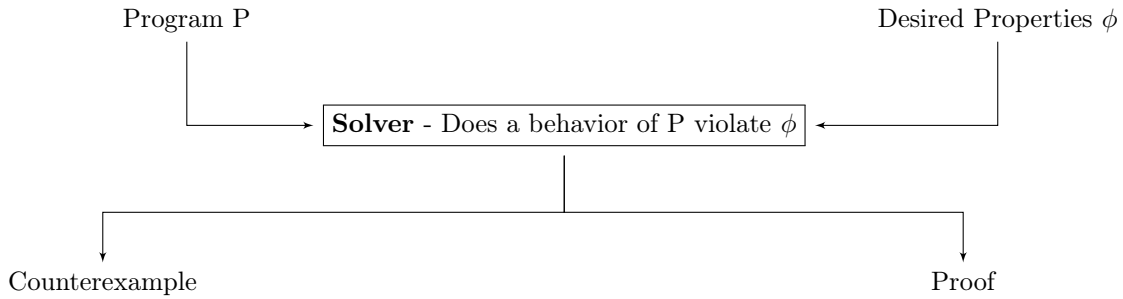
Below is an example for a socket leak in C:

```
int setup_socket() {
   int socket_id, client_socket_id;

    socket_id = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_id < 0) {
      printf("Failed to create a socket\n");
        return -1;
    }

    // setup serv_addr struct, port, etc...

    if (bind(socket_id, ...) < 0) {
      printf("Failed to bind\n");
        return -1;
    }

    // listen, accept, etc...

    close(socket_id);

    return client_socket_id;
}

void work_on_socket(int socket) { ... }

int main(int argc, char *argv[]) {
   int socket;

    socket = setup_socket();
    if (socket == -1) {
      printf("Failed to setup a socket\n");
    }

    work_on_socket(socket);

    return 0;
}
```

A leak of the server socket happens if the bind system call fails - the function will return, and $close(socket\_id)$ will not run. Of course, in this simplistic example nothing terrible will happen — the program will close and all its resources will be freed. But, put such a piece of code in a larger context, where a process opens and manages many connections and should always be up, put it to run in a highly loaded system, and this bug may manifest in unexpected ways. For example, the running process won't be able to open any files because it exhausted all its available file descriptors.

A reliable, efficient, automatic mechanism for verifying programs can aid us in avoiding such bugs.

Ideal scheme of automatic program verification



Such an ideal scheme will be able to run at compile time, where it has access to the program's source and properties. It is able to provide a correct answer, valid for any input given to the program. A behavior of a program P, is a single path of execution among all possible execution paths.
Unfortunately, such a solver cannot exist in practice, as we will see next.

Undecidability and Rice's Theorem

A classic example of undecidability is the **Halting problem**: *Given a program P and input I, will P(I) halt?*
Rice's theorem can be seen as a generalization of the halting problem.

Rice's Theorem - *For any non-trivial property of partial functions, there is no general and effective method to decide if a program computes a partial function with that property.*

Definition 1 *a **partial function** is a function that is defined only on a subset of its domain. For example, $f : X \to Y$ is defined only on $X' \subset X$, and for every $x \notin X'$, $f(x)$ is undefined.*

Definition 2 *a **property** of a function is a partition of the set of **computable** partial functions to non-empty sets - those who satisfy the property and those who don't.*

Definition 3 *a **non-trivial property** is a partition which does not include the empty set. Simply put, a property for which there exist both a program computing a partial function satisfying it, and a program computing a partial function that does not satisfy it.*

As a matter of fact, almost every property which is *interesting* is non-trivial. Because trivial properties translate to either all the partial functions or to none, even if they are phrased in a complex way. For example, the property of functions being computable by a Turing Machine (or a program) is trivial because we cannot have a program that computes a non-computable function.

Definition 4 ***general** means that the method provides a correct answer for all programs.*

Example - non trivial property of a partial function - take $f : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$. An example property of such functions is that it is positive ($f \geq 0$). Then by Rice's Theorem, if we are given programs that receives two unsigned integers and returns a signed integer, it is undecidable if the return value is always positive. The only sufficient reason for the validity of Rice's Theorem in this case is the fact that there are both positive functions and negative functions. For example, $f(n, m) = n + m$ is positive, but $f(n, m) = n - m$ is not. So the property of a function that it is positive is non-trivial.

To prove that it is undecidable (for this example only) we will show a reduction from the Halting problem which is known to be undecidable. The proof will assume positiveness of a function is a decidable

property and will show a program $P(f, n, m)$ that decides if $f(n, m)$ halts or not. P will build a function f' such that $f'(n, m) = run\, f(n, m); return\, 1$. Then, P will check if f' is positive or not. Obviously, if $f(n, m)$ halts, f' returns 1, thus it is positive. Otherwise, $f(n, m)$ doesn't halt, so neither will $f'(n, m)$, thus it is not positive. So P decides if $f(n, m)$ halts. Contradiction.

A generalized reduction is a bit more complex, but draws on the same ideas.

Handling Undecidability

To overcome undecidability we can employ several methods that harm the precision and the automation of our analysis but proved to be very effective in practice.

- **Permit occasional divergence** — a static analysis tool that is allowed not to halt on certain inputs. For example, the D language allows to execute functions (subjected to certain restrictions) during compile time. Thus, if these functions enter an infinite loop, so will the compiler. It is also possible to introduce divergence by extending the language's typing system. The following blog post by a member of the C# compiler team in Microsoft shows how a candidate feature to the language causes the compiler to enter an inifnite loop. Furthermore, a typing system where subtype equivalence is undecidable may be devised, as shown in this paper[1].
- **Limited programs** — programs which are not Turing Complete. We could limit to no loops language, or finite state machines, or limit the set of configurations.
  - For example, Bitcoin Script does not contain loops by design, so transactions defined by the scripts can be securely verified. A more intricated example is when considering multithreaded programs and trying to check whether there are errors when a specific thread scheduling is employed.
  - SMV[2] is a Symbolic Model Checking tool that can be used to verify correctness of temporal logic properties in finite state models of concurrent systems, which suffer from the problem of state explosion when trying to verify them automatically. Using SMV it is possible to detect deadlocks and starvations in a concurrent model of a system, specified in the SMV language.
  - SPIN[3] is a tool that is able to verify a given specification or find a counterexample in a concurrent system. It allows the user to specify the model to be tested, add various assertions (existence of deadlocks, state verification, resource starvation and more) and run it in one of the possible modes. The modes are either random simulation mode (like a regular program runs), interactive mode (asking the programmer to choose the next instruction), or verification mode - which traverses all possible states in search of a violation of the specification. When violations are found, SPIN supplies trace of a counterexample for assisting the programmers.
  - CHESS[4] is another tool, which, similarly to SPIN, systematically runs a program with different thread interleavings and scheduling, thus being able to locate and reproduce concurrency bugs which are very difficult to trace in a normal stress test, as they can't be reproduced in a deterministic way (also called "Heisenbugs" - bugs that disappear/change when studied with a traditional debugger). Like SPIN, it provides the trace of a counterexample, which can be studied on every step taken.
  - Both SPIN and CHESS assume a finite state machine — either because the model is finite (in SPIN) or is the number of threads and their states.
- **Unsound Verification** — similar to dynamic analysis, it works in a "Best Effort" mode. We may find certain bugs, but not all of them. Such methods work excellent in practice and are common in the industry.

  - A well known tool is Valgrind, capable of detecting memory leaks, invalid reads and writes (memory corruption), use of uninitialized variables in branching, and many more features. But, just as it is not perfect in detecting all errors existing in a program, it is also not guaranteed to not output false positives.
  - Eraser is a tool that by modifying a binary to include testing calls (without changing the binary's functionality and semantics) and running the modified binary, detects data races and improper handling of locks.
  - Bounded Model Checking and Concolic Testing are other methods that will be expanded upon in the coming sections.

- **Incomplete Verification** — Consider a superset of all execution paths, and allow false alarms on bugs. The contract of such tools will be that if it reports "no bug", then indeed there is no bug. It proves to be a challenging task, as this kind of verification is required to be sound, and with a minimal number of false alarms to be of practical use. SLAM[5] is an engine that verifies that a C program correctly uses interfaces of external libraries. Microsoft's Static Driver Verifier (SDV) is using the SLAM engine for verifying Windows device driver's usage of the Windows kernel APIs. Using SDV, driver's developers are able to greatly enhance the stability of their driver.

- **Programmer Assistance** — Inductive loop invariants (see next) are difficult to infer. Tools may ask for assistance from the programmers, using crafted questions, to be able to continue with the analysis in a more efficient way. The Boyer-Moore Theorem Prover[6] (or Nqthm) is using input from the user, such as lemmas and conjectures in order to devise a proof. It's interactiveness is in the sense that, in order to prove some grander theorem, the user provides a series of lemmas leading to it.

To begin with the formal introduction to various verification models, the basic terminology of logical formula representation will be presented.

The SAT Problem

Definition 5 ***A boolean expression*** *is a formula built from binary variables, operators (AND, OR, and NOT) and parentheses.*

Example $f(\phi_1, \phi_2) = (\phi_1 \vee (\phi_2 \wedge \neg\phi_1))$

Definition 6 *A boolean expression is* ***satisfiable*** *if there exists a variable assignment for which the formula evaluates to TRUE.*

Example $f(\phi_1, \phi_2, \phi_3) = (\phi_1 \wedge \phi_2 \wedge \phi_3)$ is satisfiable, since assigning $(\phi_1 = 1, \phi_2 = 1, \phi_3 = 1)$ evaluates the formula to *TRUE*. On the other hand, $f(\phi_1) = (\phi_1 \wedge \neg\phi_1)$ can never be *TRUE* for any possible assignment, therefore — it is not satisfiable.

Definition 7 *The SAT Problem (also called* ***boolean satisfiability problem****): Given a formula made of binary predicates and operators -*

1. *Is this formula satisfiable - is there any variable assignment for which the formula evaluates to "true"?*

2. *Is this formula valid - does this formula always evaluate to "true"?*

Example

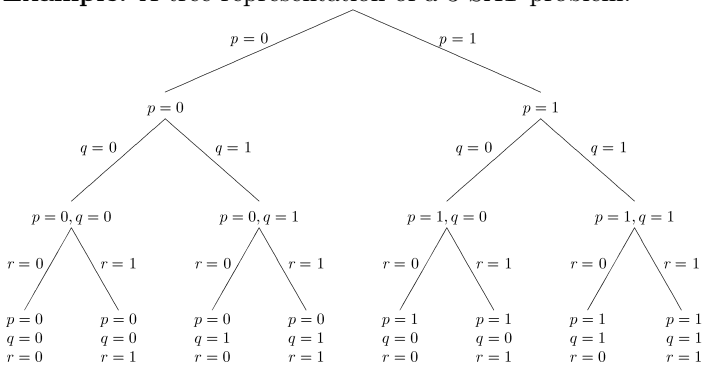
- $f(\phi_1, \phi_2) = (\phi_1 \vee \phi_2)$ can be true when $\phi_1 = 1, \phi_2 = 1$, therefore satisfiable, but is false when $\phi_1 = 0, \phi_2 = 0$, therefore not valid.

- $f(\phi_1, \phi_2) = (\phi_1 \vee \neg\phi_1) \wedge (\phi_2 \vee \neg\phi_2)$ is always true, therefore both valid and satisfiable.
- $f(\phi_1, \phi_2, \phi_3, \phi_4, \phi_5) = (\phi_1 \vee \neg\phi_2 \vee \neg\phi_3 \vee \phi_4 \vee \phi_5 \wedge \neg\phi_1)$ can never be true, therefore not valid and not satisfiable.

A naive solution for checking if a given N variables formula is satisfiable/valid consists of evaluating the formula with each of the 2^N possible values. We can see it as a search problem, where we have a full binary tree that describes all possible variable assignments, each node representing one assignment. The tree size is 2^N, and its height is N. If there is at least one leaf for which the formula evaluates to true, the formula is satisfiable. If all leaves evaluate to true, the formula is valid.

The SAT problem is a well known computer science problem, and is proven to be NP-hard. However, there are a lot of heuristic based algorithms that could provide much faster solution. There are many SAT solving programs, and even an annual competition for the fastest SAT solver. One common approach is backtracking based search (DPLL and Chaff algorithms). Another approaches, such as the PPSZ algorithm, combine randomness and heuristics to quickly find correct assignments. There is a lot of research in this field, and many relatively efficient SAT solvers are available.

Example: A tree representation of a 3-SAT problem.



Bounded Model Checking

Model checking describes the following problem: giving some model M and property P, does P hold in M? Can we calculate the answer automatically?

The bounded model checking approach depends on bounding the allowed program input to length K, then using a SAT solver for checking the properties that interest us. In the world of programs verification, such properties could be programmer's assertions, divide by zero errors, access to uninitialized variables and more.

Algorithm

1. Create formula F by performing boolean conjunction (AND) on all constrains imposed by running the program when the input is within a known bound K.
2. Create formula G by performing boolean conjunction (AND) on a set of expected properties.
3. Use a SAT solver to see if $F \wedge \neg G$ is satisfiable. If it is satisfiable - we know there exists a possible input for which the expected properties are not met, and we have a counterexample.

Example Take a look at the following program:

```
int x;
int y;
```

```
if (x == 1)
   y = 2;
else if (x == 22)
   y = 3;
else
   y = 5;
```

And the following expected properties (we will treat each assertion as a different properties set):

```
/* Assertion 1 */
assert( x == 1 || y == 3 );
/* Assertion 2 */
assert( x == 10 && y == 2 );
/* Assertion 3 */
assert( x == 1 || x == 22 || y == 5 );
```

For simplicity, we will use non-boolean values in our representation. They could be translated to boolean clauses by breaking the integers into bits, and creating a predicate for any bit. For example, $i = 7$ (assuming i is an 8-bit variable), can be represented as $i_0 = 1, i_1 = 1, i_2 = 1, i_3 = i_4 = ... = i_7 = 0$.
Formulating the program:
$F(x, y) = ((x = 1) \wedge (y = 2)) \vee ((x = 22) \wedge (y = 3)) \vee (\neg(x = 1) \wedge \neg(x = 22) \wedge (y = 5))$
Formulating assertion 1:
$G(x, y) = ((x = 1) \vee (y = 3))$

Formulating the rest of the assertion is exactly the same. The SAT solver can find that assertion 1 is satisfiable but not valid (e.g. for $x = 10$); assertion 2 is invalid (for any case where $x = 10$, the program assigns $y = 5$); assertion 3 is satisfiable and valid.

Pros:

- Powerful result: it provides an absolute guarantee that the expected properties are *always* met, and if not - exactly which input could do that.
- Flexible: not limited to computer programs, can be used for abstract design as well, as long as program constraints and expected properties can be symbolized using a formula.

Cons:

- Scalability: While SAT solvers can achieve reasonable performance, it might still not be feasible for large programs.
- Limited scope: Common features such as arithmetic operations, pointers and heap operations, procedures and concurrency are hard to symbolize using a formula and expected properties.

Examples For Existing Tools

- CBMC [7] (Carnegie-Mellon Bounded Model Checking) — a bounded model verifier for C/C++.
- Alloy [8] — a tool that analyzes models, represented in a special grammar, using SAT solver.

Concolic Testing

A major downside of bounded model checking is the scalability issue. For larger programs with many variables and a large execution tree, the cost of running a symbolic execution grows exponentially. Runtime testing, on the other hand, follows the path of an actual execution with a single input set each

time, suffering a major downside of not being able to fully cover all execution paths. Concolic testing is trying to overcome those limitations by combining the methods. Symbolic execution of the program is executed on selected input sets.

Algorithm 1 Concolic Testing

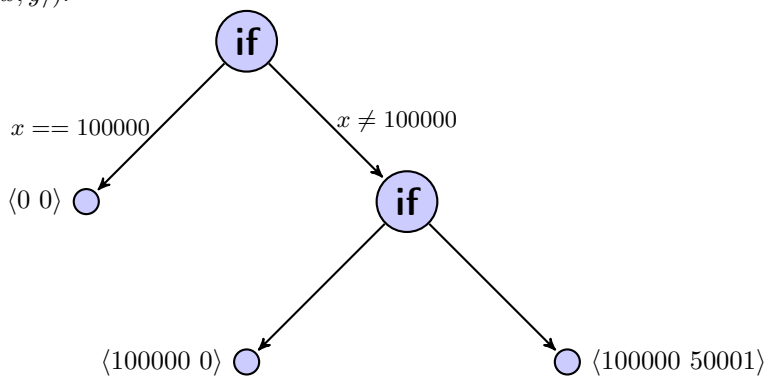
1: Classify program variables into **symbolic** input variables or **concrete** variables.
2: Instrument the program to record symbolic variables and path conditions.
3: Choose an arbitrary input set (for start).
4: Execute the program.
5: Perform symbolic execution of the program. Trace symbolic variables and path conditions.
6: Negate the unexplored last path condition in order to visit the execution path going through this condition. If there is no such unexplored path, terminate.
7: **if** SAT solver can find input set matching the following constraint **then** go to state 4.
8: **else** go to state 6 and try another execution path.

In simpler words, concolic testing is performed using runtime testing - actual execution of the program - while using SAT solver for finding proper inputs that would cover many execution paths, especially paths that are rarely reachable with normal inputs.

For example:

```
void f(int x, int y) {

   int z = 2*y;
   if (x == 100000) {
      if (x < z) {
         assert(0); /* error */
      }
   }

}
```

The program run can be represented with the following execution tree (variables states are represented as $\langle x, y \rangle$):



Pros:

- Combines the advantages of symbolic execution and runtime testing by focusing on relevant input only, while covering all possible execution paths.

Cons:

- The instrumentation required for capturing execution paths and variables is not easy.
- Scalability is still an issue. Programs that perform a lot of changes to their symbolic variables and then perform operations based on their values (for example: cryptography algorithms) could result in huge formula that practically cannot not solved.
- Coverage is still an issue. A few reasons: limitations in symbolic representation and the SAT solver; on very large or infinite execution trees, the algorithm must prune some cases and might not choose the "best" subset of states; non-deterministic programs.
- Complex data structures are hard to work with.
- For programs with non-deterministic behavior (such as random variables), it makes it hard to search through all possible execution paths, as even when the input is the same the program run can be different.

Examples For Existing Tools

- CUTE [9] — is based on the article "CUTE - A concolic unit testing engine", that coined the term "concolic testing" (and a few follow up articles). There are C and Java (jCUTE) implementations available.
- Microsft SAGE [10] — is an internal Microsoft tool, not available for public but used internally in Microsoft for validating certain components.
- KLEE [11] — is LLVM based tool that implements concolic testing. Instrumentation requires special compilation process.

Deductive Verification

Invariants

Definition 8 *An assertion I is an **Invariant** at program location if I holds whenever the execution reaches this location.*

Let's see an example:

```
void func(int x) {
   int y;
   if (x > 0)
      y = 1;
   else
      y = 2;

    // My invariant
    assert(y > 0);

   // ...
}
```

We can tell that our assertion holds regardless of x's value.

Definition 9 *An invariant is **inductive** at a loop "while B do C" if whenever C is executed on a state which satisfies B and I, it can only produce states satisfying I.*

In simpler words, an inductive invariant guarantee that a given invariant still holds as long as the loop runs, assuming it was correct at the time the program entered the loop.

For example:

```
void print_each_character(char * str, int str_len) {
   int pos = 0;

    if (str_len < 0)
      return;

    while (pos < str_len) {
        printf("%d --> %c", pos, str[pos]);
        ++pos;

        // Inductive invariant
        assert(pos > 0);
        // Another inductive invariant - still correct
        assert(pos > 1);
        // Incorrect invariant -
        // For pos = 0, does not hold after 1024 iterations
        assert(pos < 1024);
        // Incorrect invariant -
        // Can happen for example if initial value of pos is -124
        assert(pos != -123);
    }
}
```

As you can see in the example, there could be many inductive invariants. For example, the invariant "True" is correct for *any* loop. The aim is to make them as weak as possible in order to prove the desired property. For example, both $pos > 1$ and $pos > 0$ are correct - but $pos > 0$ is wider and therefore gives stronger assumption.

The inductive property of inductive invariants means that assuming the pre-condition, this invariant holds as long as the loop runs, **regardless of the rest of the program**.

For example:

```
int x = 0, y = 0;
int a = 1, b = 2;

while (x < 10000)
{
   x = a + 1;
    y = b;
    ++a, ++b;

    // Not inductive
    assert (x == y);
    // Inductive
    assert (x == y && (a == b - 1));
}
```

Is "$x == y$" a correct inductive invariant? No. Even though this is an **invariant** for this program, it is not an **inductive invariant**, since it depends on the initial values of a and b, which are outside the

scope of the loop. On the other hand, "$x == y \wedge (a == b - 1)$" is an inductive invariant as now it also includes sufficient constraints on all the values of a and b, including the initial ones.

Verifying Inductive Invariants

The method of deductive verification enables proofs of temporal properties of systems with infinitely many states (such as program states). We are using deductive verification for **verifying** inductive invariants in loops, with the programmer's assistance. As we could have seen before, inductive invariant are very powerful when analyzing a program, as it can guarantee that certain invariants are correct all through the loop, as long as the pre-condition apply. On the other hand, it is not easy to find non-trivial inductive invariant.

The input is program P and candidate invariant γ. We will use the SAT solver to test if the input invariant is indeed an inductive invariant, giving us either a proof that this invariant is inductive, or a counterexample.

For example, the following function is printing the digits of an unsigned number in reverse order.

```
#define ARR_SIZE <Some Value>

void print_reversed_unsigned_int(unsigned int n) {
   char s[ARR_SIZE];
   int i = 0;
   do {      /* generate digits in reverse order */
      s[i++] = n % 10 + '0'; /* get next digit */
   } while ((n /= 10) > 0);  /* delete it */
   s[i] = '\0';

    printf("%s", s);
}
```

A verification program that would want to make sure there are no out-of-bound array access, would need to verify that $(i \geq 0) \wedge (i \leq ARR\_SIZE)$ is an inductive invariant (correctness of this assumption depends on ARR_SIZE being at least the maximum size of unsigned int plus one). It is important to remember than in this case, ARR_SIZE should be treated as a constant - part of the program code - not as a variable.

Note that deductive verification assumes that the invariant is provided by the programmer. The process allows us to *verify* the assumption regarding this invariant (prove or provide a counter-example), not to *find* inductive invariants by itself - this is a much harder task.

Examples For Existing Tools

- ESC/Java [12] — Tool for finding common errors such as array out-of-bounds access in Java programs.
- Dafny [13] — By Microsoft research. Offers a programming language that has built-in support for invariants, together with a program verification tool.

Example: Running Dafny

Dafny is a Microsoft research programming language, that offers built-in support for verifying code correctness using special annotations. In the example below, a simple test code is written in special Dafny syntax, and was verified using Dafny online tool.

Notice the **invariant** keyword. This is a special Dafny annotation that describes the loop invariant. In the first program (program 1), the invariant was correct. In the second program (program 2), Dafny could identify that the program might never terminate, therefore the verification failed. In the third program (program 3), a loop counter ensured that the program terminates and Dafny could detect that the loop invariant does not hold.

```
1 method MyTestLoop()
2 {
3   var x := 2.0;
4   while x < 100000.0
5   invariant x > 1.0
6   {
7     x := (2.0 * x) - 1.0;
8   }
9 }
10
```

(a) Program 1

```
1 method MyTestLoop()
2 {
3   var x := 0.5;
4   while x < 100000.0
5   invariant x > 0.0
6   {
7     x := (2.0 * x) - 1.0;
8   }
9 }
```

(b) Program 2

```
1 method MyTestLoop()
2 {
3   var x := 0.5;
4   var t := 1;
5   while t < 1000
6   invariant x > 0.0
7   {
8     x := (2.0 * x) - 1.0;
9     t := t + 1;
10  }
11 }
```

(c) Program 3

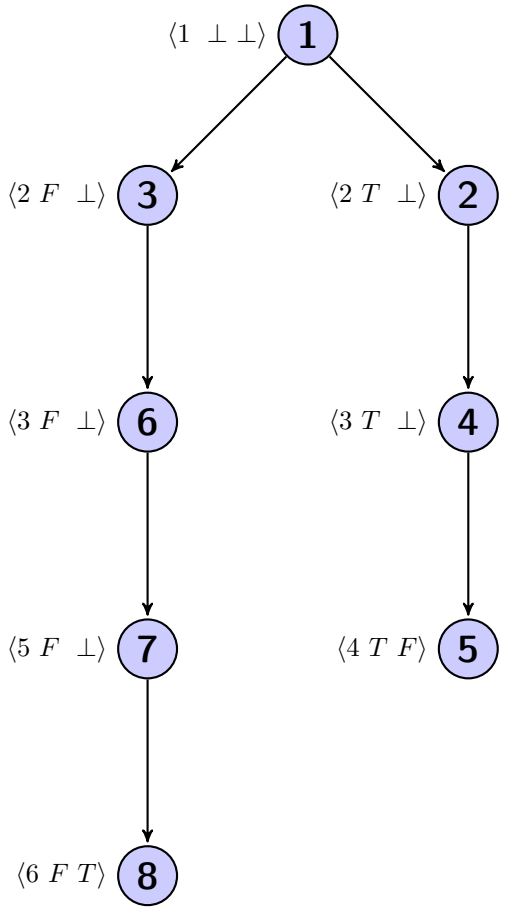
Transition Systems

Semantics of a program can be described as a directed graph of states, each state describes the current values of the program variables (this includes special variables such as the program counter, which points on the current line in the program). The relations between states are the program statements and conditions. Meaning: states A and B are connected if the program flow allows transitioning from state A to state B. Notice that this graph grows very fast, and in many cases it is infinite.

Example Take a look at the following program:

```
/* 1 */ bool x, y;
/* 2 */ x = read_input();
/* 3 */ if (x == true)
/* 4 */   y = false;
/* 5 */ else if (x == false)
/* 6 */   y = true;
```

The state consists of x's value, y's value, and the program counter. The program can be represented using a state vector made of $\langle PC, x, y \rangle$. And the following transitions:



A transition system can be used for proving various program properties, based on the reachability of certain states. A reachable state in the graph means that the program can reach this state. Therefore, it is possible to prove that certain "bad" states cannot be reached if this state is not reachable in the graph. Notice that we have the additional benefit of not only knowing that if such state is reachable, we can also tell what is the execution path leading to this state, so we can fix the program.

Observation 10 *Bounded model checking and concolic testing are covering a subset of the transition graph.*

In our case, this was a simple program with no loops or procedure calls. However, for real-world program, the transition graph can easily grow, in many cases to infinite size. For example, think of a program that searches arbitrary strings inside files. How many states can such program hold?

Abstract Interpretation

Concept

As stated before, methods which try to assess a program's correctness by representing a program's exact state throughout its execution may suffer from major scalability issues, since number of states may grow rapidly. With Abstract Interpretation, the idea is to represent a group of program-states by a single abstract state, thus minimizing the problem to a "solvable size".

However, this does not come without a price. The "unification" of multiple concrete states causes loss of information, **inherently** resulting in two major affects:

- false alarms: The abstraction of a group of concrete values may be a group that includes values that were not an actual concrete state (say, when representing 1,5, 9, 11.. by the group of all odd integers). if one of the "newly added" values happens to satisfy $\neg\phi$, then we have a false-alarm case.

- uncertainty: Obviously ϕ is sated with relations to the concrete program state. However, a single abstract state AS_1 may include both ϕ-satisfying concrete-states and $\neg\phi$-satisfying concrete-states. In this case, even if we were to determine that the program-state is described by AS_1, we still can't determine whether or not $\neg\phi$ is satisfied by the program - resulting in a "maybe" verification output.

Hence, it seems that the "art" of abstract interpretation is finding the right abstraction with regards to both the program's concrete states - adding minimal amount of "new" states - and the ϕ statement in question - creating as little ϕ-ambiguous abstract states as possible.

A Conceptual Example

say we have a group on people in a room, each represented by a date-of-birth and a name. we may represent the group by a few different abstract states:

- the range of birthday dates only (no year), from earlier to latest.(one abstract state)

- the range of birth years, from earlier to latest.(one abstract state)

- an abstract state for each birth year.(many abstract states)

now let's consider the following computational problems:

- at least one of the people in the room has a birth day today.

- at least one of the people in the room is entitled to a train-ride discount (either an elderly citizen or a minor).

- at least one of the people in the room is an army-age person.

each of these representations may result in a full match, false-alarm or ambiguity with regards to each of these ϕ statement:

ϕ/abstraction	DOB range	birth years range	birth year values
BD today	false alarm	ambiguity	ambiguity
train-ride discount	ambiguity	full match	full match
army-age	ambiguity	ambiguity	full match

As we mentioned, abstraction allows us to describe many concrete states by a single abstract state. Proving that a program is valid can now be achieved by showing that the system is constantly in an abstract state that is ϕ satisfying. Moreover, if we were to find such a state, then that abstract state would reflect the invariant that must be held in order to maintain ϕ correctness.

Inductive Invariant

we'll look at one specific type of invariant. We'll say that an argument γ is an inductive invariant of a loop L if for every program-run where γ holds at the entrance line to loop L, γ will also hold at the end of the loop execution.

Abstraction Functions

Let L be an ordered set, called a concrete set and let LâĂš be another ordered set, called an abstract set. These two sets are related to each other by defining functions that map elements from one to the other.

A function α is called an abstraction function if it maps an element x in the concrete set L to an element α(x) in the abstract set LâĂš. That is, element α(x) in LâĂš is the abstraction of x in L.

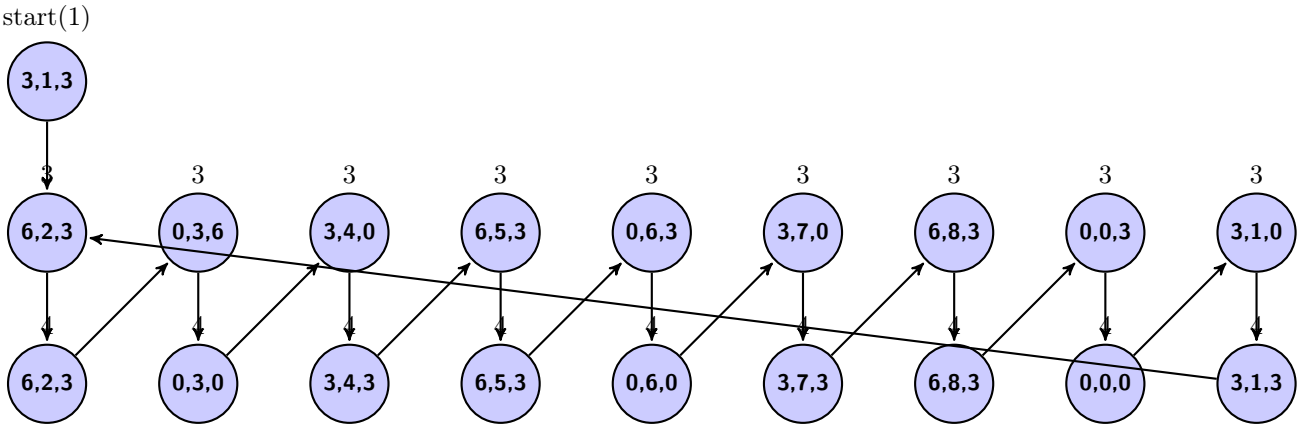
A function β is called a concretization function if it maps an element x' in the abstract set L' to an element β(x') in the concrete set L. That is, element β(x') in L is a concretization of x' in L'. α will be considered a valid abstraction if $\beta(\alpha(x)) > x$, and if $\alpha(\beta(\alpha(x))) = \alpha(x)$.

Example

we'll consider the following code, define its abstraction function, and find its inductive invariant from the abstraction interpretation.

```
/* 1 */ int x = 3, y = 10, z = x * y;
/* 2 */ while (true) {
/* 3 */  x = x + 3; y = y + 1;
/* 4 */  z = x * y;
/* 5 */    ASSERT( z == x * y)
/* 6 */ }
```

we'll define α(x) = (sum of x's digits) mod 9. now, we'll describe the program's run abstractivly, where each node represents PC:α(x), α(y), α(z).



and as we can see, me managed to:

- prove that z == x * y does hold for all possible runs.
- find an inductive invariant even stricter when what we proved - that in fact α(z) == $\alpha(\alpha$(x) * α(y)). See casting out nines method for quick caculation verification tricks.

References

[1] A. J. Kennedy and B. C. Pierce, "On decidability of nominal subtyping with variance," Sept. 2006. FOOL-WOOD '07.

[2] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," 1993. UMI Order No. GAX92-24209.

[3] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[4] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Tech. Rep. MSR-TR-2007-149, Microsoft Research, November 2007.

[5] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," Tech. Rep. MSR-TR-2004-08, Microsoft Research, January 2004.

[6] R. S. Boyer, M. Kaufmann, and J. S. Moore, "The boyer-moore theorem prover and its interactive enhancement," 1995.

[7] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (K. Jensen and A. Podelski, eds.), vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[8] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 256–290, Apr. 2002.

[9] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263–272, Sept. 2005.

[10] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, pp. 20:20–20:27, Jan. 2012.

[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 209–224, USENIX Association, 2008.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," *SIGPLAN Not.*, vol. 37, pp. 234–245, May 2002.

[13] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, (Berlin, Heidelberg), pp. 348–370, Springer-Verlag, 2010.