Software Safety

Formal Verification & Static Analysis

Prof. Dr.-Ing. Patrick Mäder, M.Sc. Martin Rabe

Contents

Formal Verification and Model Checking

Landscape of V&V Activities

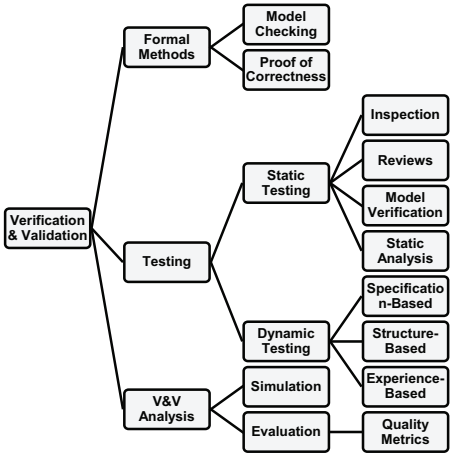


Figure A.1 — Hierarchy of Verification and Validation activities

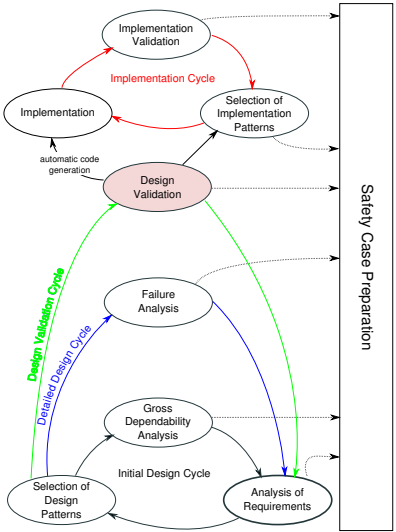
[ISO/IEC/IEEE 29119-1:2013. Standards catalogue. International Organization for Standardization. September 2013]

Development Questions to be Answered by Formal Verification

We have a design that meets our dependability requirements. Is it correct?

Could the protocol between the two processors lock up?

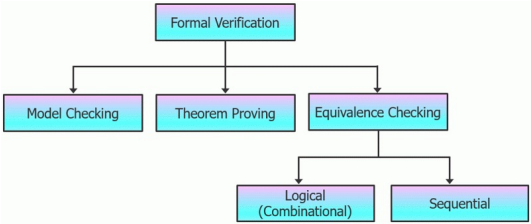
Will the algorithm calculating the speed of the train work under all sequences of inputs?



Formal Verification Techniques

- Formal verification is the act of **proving** or **disproving** the **correctness** of a **system** with respect to a certain formal **specification** or property [Wikipedia]

- **Potential benefits**
  - Detects problems early in the design cycle
  - Less time-consuming
  - Reliable
  - Exhaustive



[Aijaz Fatima: **Introduction to Formal Verification.**, EEWeb, 2019.]

Formal Methods and the Safety Standards

- Formal methods prescribed by safety standards
  - IEC 61508 “highly recommends” formal methods at SIL4 and semi-formal ones at SIL3.
  - EN 50128 “highly recommends” formal methods at SIL3 and SIL4.
  - ISO 26262 “recommends” formal methods at ASIL-C and ASIL-D.
  - IEC 62304 requires that software requirements “do not contradict [and] are expressed in terms that avoid ambiguity” but explicitly says that “This standard does not require the use of a formal specification language.”

- **Formal methods**: formal (mathematical) syntax and semantics (focus of this section)
- **Semi-formal methods**: formal (mathematical) syntax and informal semantics, e.g., PetriNets

Simulation vs. Formal Verification

- Simulation
  - deals with **probabilities**
  - can be used to build confidence in a system but not to prove its correctness
  - typically requires less mathematical thinking
  + typically handles a much larger scope than formal verification

- Formal Verification
  - deals with **possibilities**
  + can be used to prove properties about a system
  - requires more mathematical thinking

Simulation

Formal verification

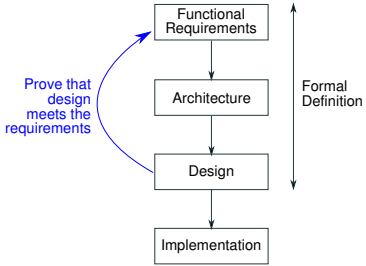
top [Darksiders Chaoseater, Sword],
bottom: [Kyojuro Rengoku demon slayer]

Formal Verification within the Development Process (1/2)

- **Method 1: Full Formal Specification**
  1. Write system specification in a formal language (e.g., Event-B, TLA+)
  2. Determine and specify system invariants
  3. Use model checker to prove invariants from specification
  4. Implement the system (or generate code automatically)

- e.g., “Patient’s pulse rate < configured value ⇒ alarm sounds within 100ms”

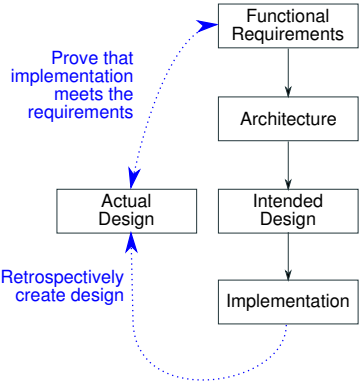
- “Highly recommended” for SIL3 and SIL4 in IEC61508 and EN50128



Formal Verification within the Development Process (2/2)

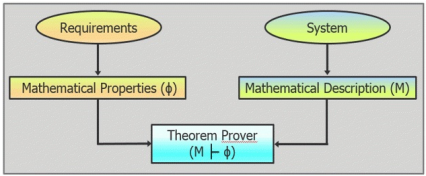
- **Method 2: Proving an Existing System**
  1. Study the system
  2. Determine the conditions that need to be true (or false)–the invariants
  3. Build a model of the system and include the invariants
  4. Ask a model checker to prove the conditions

- Suggested by the FDA for verifying implementation against design [Ganesan et al. 2011]



Theorem Proofing

- verifying that the implemented system meets design requirements (or specifications) using mathematical reasoning, i.e., a proof-based approach
- \+ can handle very complex systems
- – requires manual intervention and expertise to complete the proofs
- – in case of a failed proof, no counter-examples – locating the error can be difficult



[Aijaz Fatima: Introduction to Formal Verification., EEWeb, 2019.]

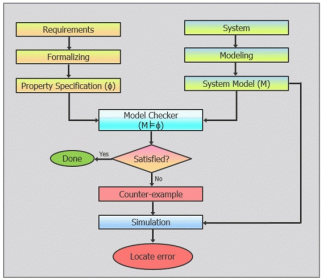
- Steps
  1. model the system as set of mathematical definitions in a formal mathematical logic
  2. derive the properties of the system that follow from the mathematical definitions
  3. use a theorem prover to verify that the system meets the specifications

Model Checking (1/2)

- **Model checking algorithms** are based on state space exploration, i.e., “brute force”
- state space describes all possible behaviors of the model
- state space $\sim$ graph:
  - nodes = states of the system
  - edges = transitions of the system
- in order to construct state space, the model must be closed, i.e., we need to model environment of the system

+ verification process is fully automatic, once the system model and property specification are formalized

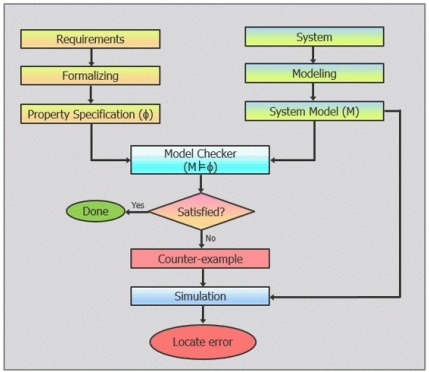
– only for small systems in terms of states to be processed by the model checker



[Aijaz Fatima: **Introduction to Formal Verification.**, EEWeb, 2019.]

Model Checking (2/2)

- **Model checking procedure**
  1. model system as model *M* as set of states with a set of transitions between them describing how the system responds to internal or external stimuli
  2. formalize a property, description of the system's behavior, using a specification language, such as PSL or SVA, as formula Φ
  3. run the model checker to find out whether model *M* satisfies formula Φ
  4. if the model does not satisfy the property, a counterexample is generated, i.e., a stimulus that violates the property
  5. run the counterexample with the system model in simulation to find the location of error



[Aijaz Fatima: Introduction to Formal Verification., EEWeb, 2019.]

- Observations at Amazon Web Services (AWS) after using formal methods for 4 years:
  - "Formal methods **find bugs in system designs that cannot be found through any other technique** we know of."
  - "Formal methods are **surprisingly feasible for mainstream software development** and give good returns on investment."
  - "At Amazon, formal methods are **routinely applied** to the design of complex real-world software, including public cloud services."

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

[Newcombe at al.: How Amazon Web Services Uses Formal Methods. Communications of the ACM, 2015, 58:4, 66–73]

Overview of Available Model Checkers

- Free and open-source
  - SMV (Symbolic Model Checker): Carnegie-Mellon University, USA
  - SPIN: Bell Labs, USA
  - TLA+: Leslie Lamport
  - Realtime Maude: University of Oslo
  - Design/CPN (Coloured Petri Nets): University of °Arhus, Denmark
  - UPPAAL: Aalborg University, Denmark and Uppsala University, Sweden
  - KRONOS: University of Grenoble and CNRS
  - HYTECH: Cornell University, USA
  - Rodin: European Union ICT Project DEPLOY
  - etc.
- Commercial products
  - VerumST's Dezyne
    - claims a reduction in womb-to-tomb costs from 40€ to 6€ per LoC
  - The Escher Verification Studio

The SPIN Model Checker

SPIN: Simple Promela Interpreter (1/2)

- SPIN (Simple Promela Interpreter) is a popular open-source model checker that can be used for the formal verification of asynchronous and distributed software systems
- Two core specification concepts
  - intuitive, C-like notation (Promela) for specifying the **finite-state abstraction of a system** unambiguously
  - A notation for expressing general **correctness requirements as LTL** (linear temporal logic) formulae
- **Identifies common flaws of distributed software**: deadlock, livelock, starvation, underspecification, overspecification

[SPIN Homepage]

Developed by Bell Labs since 1980, ACM software system award 2002

SPIN: Basic Concepts (2/2)

- Spin can be used in 2 basic modes
  - as a **simulator** to get a quick impression of the behavior captured by the system model
    - guided simulation
    - random and interactive simulation
  - as a **verifier**
    - when a counterexample is generated, it uses simulation to step through the trace
    - prove correctness of process interactions
    - thereby, processes refer to system components that communicate with each other, modeled as rendezvous primitives (synchronous), buffered channels (asynchronous), or shared variables

[SPIN Homepage]

Promela: Process Meta-language

- PROMELA (Process Meta-language), served as input to SPIN
- C-like specification language for specifying the system behavior of a distributed system as a finite-state model
- Orchestrates ideas from
  - CSP process algebra by Hoare for input/output
  - C language in some of the syntax and notational conventions
  - a non-deterministic guarded command language by Dijkstra
- Consists of **processes**, **data objects**, and **message channels**
  - each of theses is bounded as the model corresponds to a FSM

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
  ...
}

proctype Receiver() {
  ...
}

init {
  ...
}
```

process body

creates processes

[Theo C. Ruys: **SPIN Beginners' Tutorial**. Grenoble, 2002.]

Expressing Correctness Claims: Linear Temporal Logic

- Conditions are be expressed using Linear Temporal Logic (LTL) and will be converted into Büchi Automata

- **Types of correctness claims**
  - Basic assertions Basic assertions (checked during simulation run)
  - End-state labels
  - Progress-state labels
  - Accept-state labels checked during verification run
  - Never claims
  - Trace assertions

LTL examples

$\Box P$	always P	Invariance

There is always at least one free buffer

$\Diamond P$	eventually P	Guarantee

Eventually the buffer is released.

$P \rightarrow \Diamond Q$	if P then eventually Q	Response
$P \rightarrow Q \ U \ R$	if P then Q until R	Precedence
$\Box\Diamond P$	always eventually P	Progress

The system always sends another message.

$\Diamond\Box P$	eventually always P	Stability

Eventually the usage of the link drops below 20% and remains there.

Special Correctness Claims: Safety and Liveness

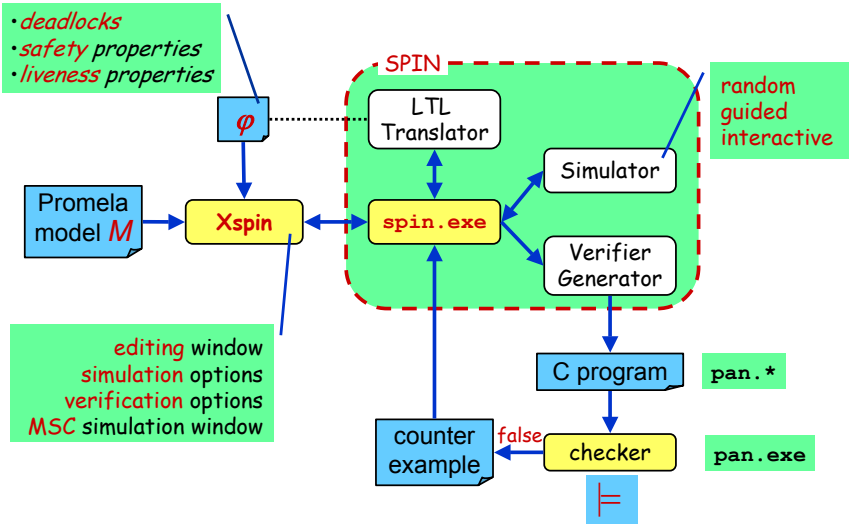
- **Safety properties** are difficult to prove; **liveness properties** are often impossible to prove
- Safety Property: "nothing bad ever happens"
  - invariant, e.g., x is always less than 5
  - deadlock freedom, i.e., the system never reaches a state where no actions are possible

  → **SPIN**: find a trace leading to the "bad" thing. If there is no such trace, the property is satisfied.
- Liveness Property: "something good will eventually happen"
  - termination, the system will eventually terminate
  - response, if action X occurs then eventually action Y will occur

  → **SPIN**: find a (infinite) loop in which the "good" thing does not happen. If there is no such loop, the property is satisfied.

[Cook et al.: "Proving That Programs Eventually Do Something Good"] and [Theo C. Ruys: **SPIN Beginners' Tutorial**. Grenoble, 2002.]

The Overall SPIN Approach



[Theo C. Ruys: **SPIN Beginners' Tutorial**. Grenoble, 2002.]

SPIN Example: The Problem (1/3)

Mutual Exclusion: Two threads access a shared resource, but only one thread should be allowed access at the same time to avoid race conditions.

Shared Resource:

```
int x = 0;
```

Thread A:

```
void * addOne(void *p)
{
  int y;
  int i;
  for (i=0; i<100; i++)
  {
    y = x + 1;
    x = y;
  }
  return (0);
}
```

Thread B:

```
void * addOne(void *p)
{
  int y;
  int i;
  for (i=0; i<100; i++)
  {
    y = x + 1;
    x = y;
  }
  return (0);
}
```

SPIN Example: The PROMELA Program (2/3)

```
bit turn = 0;
bit flag[2];
byte critical = 0;
proctype P(byte i) {
  flag[i] = 1;
  do
  :: turn != i ->
    if
    :: flag[1 - i] == 0 -> turn = i
    fi;
    :: else -> break
  od;
  critical++;
  /* critical section */
  assert(critical < 2);
  critical--;
  flag[i] = 0;
}
```

```
init {
  flag[0] = 0;
  flag[1] = 0;
  atomic {
    run P(0);
    run P(1);
  }
}
```

The program from last slide converted to Promela. With the assertion we represent the check that only one thread is allowed in the critical section at a time.

SPIN Example: Assertion Violated (3/3)

List of steps Spin identified as a counter-example to invalidate the assumption.

```
 1:proc 0 (:init::1) :53 (s1) [flag[0] = 0]
 2:proc 0 (:init::1) :54 (s2) [flag[1] = 0]
 3:proc 0 (:init::1) :56 (s3) [(run P(0))]
 4:proc 0 (:init::1) :57 (s4) [(run P(1))]
 5:proc 2 (P:1) :36 (s1) [flag[i] = 1]
   flag[0] = 0
   flag[1] = 1
 6:proc 2 (P:1) :38 (s2) [((turn!=i))]
 7:proc 2 (P:1) :40 (s3) [((flag[(1-i)]==0))]
 8:proc 1 (P:1) :36 (s1) [flag[i] = 1]
   flag[0] = 1
   flag[1] = 1
 9:proc 1 (P:1) :42 (s7) [else]
10:proc 2 (P:1) :40 (s4) [turn = i]
   turn = 1
11:proc 2 (P:1) :42 (s7) [else]
12:proc 2 (P:1) :44 (s12)
          [critical = (critical+1)]
   critical = 1
13:proc 2 (P:1) :47 (s13)
          [assert((critical<2))]
14:proc 1 (P:1) :44 (s12)
          [critical = (critical+1)]
   critical = 2
spin: .pml:47, Error: assertion violated
spin: text of failed assertion:
      assert((critical<2))
 15:proc 1 (P:1) :47 (s13)
           [assert((critical<2))]
spin: trail ends after 15 steps
#processes: 3
   turn = 1
   flag[0] = 1
   flag[1] = 1
   critical = 2
15:proc 2 (P:1) :48 (s14)
15:proc 1 (P:1) :48 (s14)
15:proc 0 (:init::1) :59 (s6)
          <valid end state>
```

Example 2: Looking at a Protocol (1/2)

- X.21 was state of the art in the late 1970s for communication between a piece of telecommunications equipment (e.g., a Teletype), DTE, and its modem (DCE).

→ Unfortunately, the protocol has a bug: it is incomplete.

- Colin West (of IBM) found the bug and published a paper: **General technique for communications protocol validation**
- At the time, this stretched the limits of what could be done – now the problem can be detected quickly on a laptop computer using SPIN

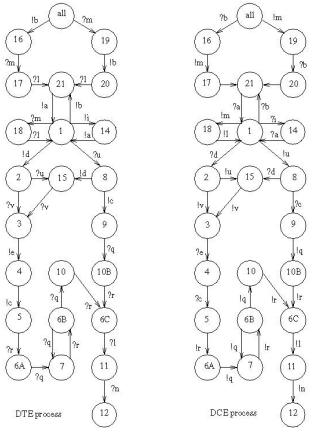


Figure 4 — X.21 Specification (1977)

[West, C. H.: **General technique for communications protocol validation**. IBM Journal of Research and Development 22:4, 393–404,1978]

Example 2: Reaching the Invalid State and Correcting the Protocol (2/2)

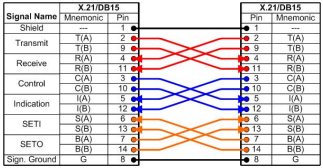
```
spin -v -t x21.pmli=2 pno 1
1: proc 1 (dce proc) x21.pml:126 (state 11) [dte!u] i=2 pno 0
2: proc 0 (dte proc) x21.pml:17 (state 11) [dte?u] i=2 pno 0
3: proc 0 (dte proc) x21.pml:83 (state 81) [dce!d] i=2 pno 1
4: proc 1 (dce proc) x21.pml:192 (state 81) [dce?d] i=2 pno 1
5: proc 1 (dce proc) x21.pml:186 (state 74) [dte!v] i=2 pno 0

...

17: proc 1 (dce proc) x21.pml:160 (state 46) [dte!q] i=2 pno
0
18: proc 0 (dte proc) x21.pml:52 (state 48) [dte?q] i=2 pno 1
19: proc 1 (dce proc) x21.pml:156 (state 43) [dte!r] i=2 pno 0
20: proc 0 (dte proc) x21.pml:58 (state 55) [dte?r]

spin: trail ends after 20 steps
```

→ The protocol was corrected



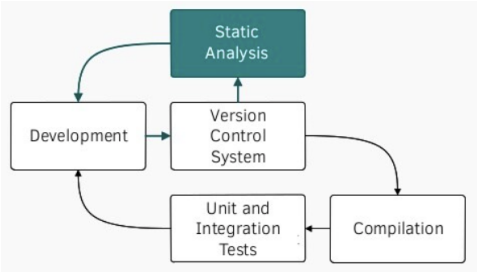
- Rerunning SPIN proves that the revised protocol is complete (testing would only prove that we had never stumbled across the incomplete situations)
- The SPIN model now represents a formal, proven specification of the protocol

Limitations of SPIN

- Cannot handle infinite-state systems
- Cannot handle time (actually not a limitation)
- As with any other model checker, it's limited in the size of model it can handle.
- As with any other model checker, it's hard to get the level of abstraction right: **this type of modeling is a skilled job**

Static Program Analysis

Static Program Analysis



[Dániel Stein: Graph-Based Source Code Analysis of JavaScript Repositories, Slideshare, 2016]

- No need for compilation or execution of the application
- Formatting, structural and semantic **rule checking**
- Can **extend the workflow** of continuous integration and improve it

Static Program Analysis: Types of Defects Discovered

- **How can it find implementation issues without awareness of requirements?**

- Examples of implementation issues that are not requirements-specific
  - computations that are **out of bounds** (cp. Ariane 5)
  - reading from **uninitialized memory** (places that no-one ever wrote to)
  - conditionally **accessing invalid memory** addresses (e.g. NULL pointer exceptions)
  - taking decisions ("if" statements) that depend on **invalid accesses**
  - … and many more

- ESA states: *"Based on our experience, there are many missions whose SW never went through such checks (and are therefore susceptible to this kind of issues)."*

Static Program Analysis: Recommended for Dependable Developments

- ISO 26262 highly recommends the use of static analysis
  - verifying that the coding standards have been followed during implementation [Table 7 of part 6]
  - verifying the software integration [Table 10 of part 6]
- IEC 61508 recommends static analysis for software verification [Table A.9 of part 3, paragraph B.6.4 of part 7]
  - "To avoid systematic faults that can lead to breakdowns in the system under test, either early or after many years of operation."
- Medical software: the FDA has identified the use of static analysis for medical devices [FDA2010]
- Nuclear software the UK office for nuclear regulation (ONR) recommends the use of static analysis on reactor protection systems [ONR2013]
- Aviation software recommended in combination with dynamic analysis [FAA2002]

Static Program Analysis: Approximation and Compromises

- [**Halting problem**]: Can we write an analyzer that can prove, for any program *P* and inputs to it, *P* will terminate?
  - → undecidable – any analyzer will fail to produce an answer for at least some programs and/or inputs
- Other undecidable properties ([Rice's theorem])
  - Does this SQL string come from a tainted source?
  - Is this pointer used after its memory is freed?
  - Do any variables experience data races?

→ All static analyses are therefore compromises

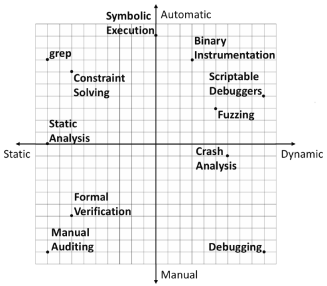
- they find **false positives** (report faults that do not exist)
- they do not find faults that do exist (**false negatives**)

[Alan Turing, Wikipedia]

Static Program Analysis: Analysis Types

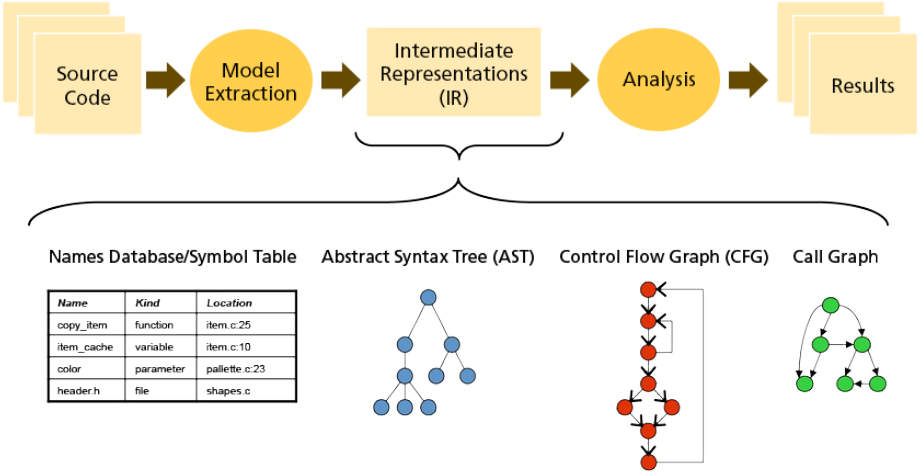
- **Abstract syntax tree (AST) based**
  - check program's syntax against coding rules
  - perform syntax checking with semantic knowledge of the programming language
- **Data mining based**
  - estimate the likelihood of faults in a particular program
- **Control flow graph based**
  - create correctness proofs against pre-defined invariants (e.g., call `free()` without earlier call to `malloc()`)
  - perform symbolic execution of the code (a combination of static code analysis and testing)

Program analysis landscape



[Julian Cohen: Contemporary Automatic Program Analysis. blackhat, 2014]

Static Program Analysis: Typical Analysis Pipeline



[Verifysoft: How Static Analysis Works]

AST Based Checking: Lint

- In general: **compilers are static analysis syntax checkers**
- **Lint-like checkers** are in use for a long time and provide more sophisticated checking than compilers
  - Examples: lint for C and C++, pylint for Python, jsl for Javascript, lint4j for Java
  - Lints can be customized to local coding rules

→ **Problems**: lint results contain **many false positives** (finding problems that are not there) – may need to add code to prevent known false positives

Pylint example

```
W: 57:doPearson: Redefining name 'rho' from outer
scope (line 267)
W:115:doPearson: Redefining name 't' from outer scope
(line 267)
R: 45:doPearson: Too many local variables (20/15)
W: 82:doPearson: Unused variable 'totalYRanking'
W:152:doLinRegression: Redefining name 'c' from outer
scope (line 262)
C:119:doLinRegression: Missing docstring
R:119:doLinRegression: Too many local variables
(20/15)
W:182:createCorrelationTable: Redefining name
'complexity' from outer scope (line 259)
W:202:scanComplexities: Redefining name 'complexity'
from outer scope (line 259)


Global evaluation
-----------------

Your code has been rated at 8.83/10
```

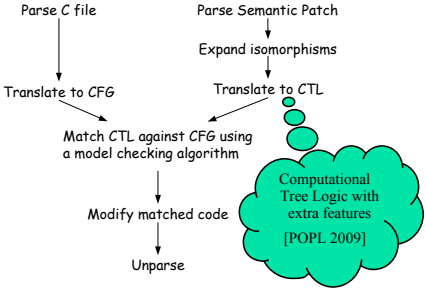
Semantic Syntax Checking

- Class of analyzers that **understands some of the programming language's semantics**, e.g., concepts such as "path", "expression", "identifier", "constant", and "statement"
- Allow the specification of rules in terms of theses concepts
  - e.g., "For every path, do ..."
- Due to this semantic knowledge, tools can transform programs smartly
  - e.g., "replacing identifier fred with joe" should not affect a comment or string with the word fred or freda in it

- **Prominent semantic syntax checkers** are: Coccinelle (next slides), XTRAN

Semantic Syntax Checking: Coccinelle (1/2)

- Program **matching** and **transformation** for unpreprocessed C code
- **Semantic Patch Language (SmPL)**
  - based on the syntax of patches
  - declarative approach to transformation
  - high level search that abstracts away from irrelevant details, i.e., spacing, indentation, comments and **variations in coding style**
    - e.g., `if(!y)` `if(y==NULL)` `if(NULL==y)`
  - s single **semantic patch** can modify many files across many code sites



top: [Coccinelle], bottom: [Nicolas Palix et al.: Coccinelle: A Program Matching and Transformation Tool for Linux, 2009]

Semantic Syntax Checking: Coccinelle (2/2)

- The “!&” bug arises since C allows **mixing booleans and bit constants**, e.g., !*x*&*y*

```
if (!state->card->ac97_status & CENTER_LFE_ON)
    val &= ~DSP_BIND_CENTER_LFE;
```

- SmPL rule

```
@@ expression E; constant C; @@
(
   !E & !C
|
- !E & C
+ !(E & C)
)
```

- **Matching**: Coccinelle detects all !*x*&*y*, but permits !*x*&!*y*, which may be meaningful
- **Transformation**: replaces the instances with the correct !(*x*&*y*)

Data Mining Based Analysis: Fault Density Assessment

- **Example analysis goal**: find the 20% modules where 80% of the faults are located to focus code inspection
- **Fault Density Analysis needs good data from the code repository**: When was the module changed? Who changed it? Who inspected the change? etc.
- **Data Mining Based Analyses**
  - Modules' cyclomatic complexity: more complex modules probably contain more faults
  - Factors known to lead to faults (e.g., number of global variables)
  - History of the module: what percentage of lines have been changed?
  - Social interaction graphing: how often does a change made by Fred and checked by Ethel lead to more changes later?

→ **Active research area**: machine learning will facilitate more sophisticated analyses in the future

Control Flow Analysis: Correctness Proofs

- Tools understand syntax and semantics of the language and prove that predefined invariants can never be broken when the program runs
  - e.g., x can never be less than 6.5, pointer p can never be NULL when passed to doit())
- Invariants are expressed as (cp. [DbC])
  - contract defined in the code (e.g., Ada, D)
  - comments in the code (e.g., C, C#, Java)
  - externally-defined invariants

```
function search(a, n, x)
PRE:    0 < n, and a[1] ≤ ... ≤ a[n]
POST:   if x = a[k] for 1≤k ≤n
            then return(k) else return(0)

b = 1;  e = n
INV: 0 < b ≤ e, a[b] ≤ ... ≤ a[e], and
       if x = a[k] for 1≤k ≤n then b ≤k ≤e
while b ≠ e do
    {m = (b + e)/2
     if x ≤ a[m] then e = m else b = m+1}
if x = a[k] for 1≤k ≤n then x = a[b]
if a[b] = x then return(b) else return(0)
```

[Hobbs, C. (2019). Embedded Software Development for Safety-Critical Systems (2nd ed.). CRC Press.]

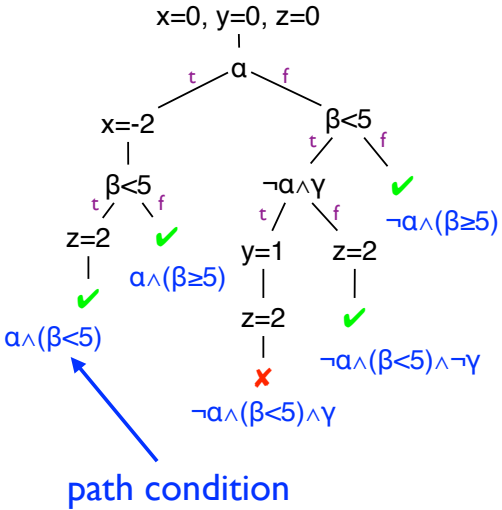
- Practical limitations
  - does not prove that the program, only what the analyst thought was important
  - if invariants are **not embedded** in code, they **become outdated** due to changes
  - adding new invariants is typically difficult due to little initial documentation

Symbolic Execution

- First proposed by King, CACM 1976
- Symbolic execution refers to **execution of program** with **symbols as argument**
- Unlike concrete execution, in symbolic execution the program can take **any** feasible path (limitation: constraint solver)
- During symbolic execution, program state consists of
  - symbolic values for some memory locations
  - path condition
- Path condition is a conjunct of constraints on the symbolic input values
- Solution of path-condition is an test-input that covers the respective path

Symbolic Execution: Example

```
1. int a = α, b = β, c = γ;
2.                    // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```



[John A. Paulson: Symbolic Execution – Course Slides CS252r, 2015]

Symbolic Execution: Program Analyses

- During symbolic execution, try determining whether certain formulas are satisfiable
  - E.g., is a particular program point reachable?
    - Figure out if the path condition is satisfiable
  - E.g., is array access $a[i]$ out of bounds?
    - Figure out if conjunction of path condition and $i < 0 \vee i \geq a.length$ is satisfiable
  - E.g., generate concrete inputs that execute the same paths
- This is enabled by powerful SMT/SAT solvers
  - SAT = Satisfiability
  - SMT = Satisfiability modulo theory = SAT++
  - E.g. Z3, Yices, STP

Symbolic Execution: KLEE

- Symbolic execution tool built on top of the LLVM compiler infrastructure
- Provides a library of filesystem and related functions
- Maintains variables and files as generally as possible (int x can take any integer value, file fred.txt can contain any data)
- Detects common faults: de-referencing a NULL pointer, etc.
- **Generates test cases with good path coverage**
- Contains a unit test environment to run the test cases

[KLEE Symbolic Execution Engine]

→[Run KLEE in browser]

Symbolic Execution: Example Program with Bug (1/2)

```
int doit(int length, char *string)
  {
  int   count = 0;
  char *ptr;
  int   lengthCount = 0;
  for (ptr = string; *ptr != '\0'; ptr++) {
    if (*ptr == 'g') {
      count++;
      if ((*(ptr-1) != 'g') &&
           (lengthCount > 0) && (length > 151)) {
        if ((count == 4) && (length % 17) == 0)
          ptr++;
        }
      lengthCount++;
      }
    }
  return count;
  }
```

This routine is not supposed to read beyond the limits of the string.

Do you spot the fault?

Symbolic Execution: Example Program with Bug (2/2)

- **Problem**: If …
  - the last character in the string parameter is the letter ‘g’
  - the character before the last in the string is not the letter ‘g’
  - there are exactly four ‘g’s in the string
  - the length parameter is greater than 151
  - the length parameter is a multiple of 17

→ **then the program reads memory outside the string**

- KLEE found parameters leading to the fault:

```
String: 0x98 0x98 0x98 0x98 g g g 0x98 0x98 0x98 0x98 0x98 0x98 g 0x00
Length: 536871118 ← note that 536871118 = 17 × 31580654
```

- KLEE also found an unintended fault, if the first character of the string is a ‘g’, then the program illegally accesses the character before the string

Static Program Analysis Gains Industry Attraction

- The software of NASA's curiosity rover was verified with static analysis (cp. [NASA2014])
- Microsoft OS' Blue Screens of Death became less frequent
  - device drivers, where any error would cause a BSoD, are now passing through static analysis (static driver verifier)
- The Linux kernel goes periodically through Coverity scans

→ Is this analysis bullet proof? No, of course not – nothing is!

Summary

Summary (1/2)

- Don’t put faults into a design or code: then you don’t have to find and remove them.
- Defining and designing the system using formal methods can, in principle, eliminate all faults.
- Defining and designing the system using semi-formal methods can eliminate some faults.
- Choose a language subset that helps you avoid faults.

Summary (2/2)

- Simulation handles **probability**, formal methods handle **possibility**.
- Simulation can be used to analyze a broad range of systems, formal model checking is an incredibly powerful tool for proving particular aspects of a system.
- Traditionally, formal model checking of a design is done before the design is implemented, it may be useful to analyze the implementation in order to check the actual design

Questions?