

# Static Analysis and Verification

SAV 2024/2025

**Tomáš Vojnar**

`vojnar@fit.vutbr.cz`

**Brno University of Technology  
Faculty of Information Technology  
Božetěchova 2, 612 66 Brno**

# Some introductory notes

- ❖ These slides are intended for the course of **Static Analysis and Verification** at FIT BUT.
- ❖ Students are assumed to have a basic knowledge of (and, perhaps even more importantly, *to like* or, at least, *not to be afraid of*) **theories of automata, graphs, logics, algebra, and modelling**.
- ❖ Students are expected to themselves **actively search and study** further information available in recommended textbooks and on the Internet.
- ❖ Students are also strongly advised to **experiment with available tools** for static (formal) analysis and verification in order to get deeper understanding of the capabilities and limits of the presented techniques.
- ❖ Note that many of the presented subjects are **relatively new, still under a very active research**, they are **rapidly developing**, and also the **terminology is not always completely uniform and stable**.

# References, Inspiration

- E.M. Clarke, T.A. Henzinger, H. Veith, and R. Bloem (Eds.). *Handbook of Model Checking*. Springer, 2018.
- C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
- E.M. Clarke, O. Grumberg, D. Kroening, D. Peled., H. Veith. *Model Checking*, 2nd edition, MIT Press, 2018.
- X. Rival, K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.
- U. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.
- X. Rival and K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.
- A. Møller and M.I. Schwartzbach. *Static Program Analysis*. Available on-line at <https://cs.au.dk/~amoeller/spa/>. Last revision 2021.
- F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*, Springer-Verlag, 2005.
- A. Biere, M. Heule, H. Van Maaren, and T. Walsh (Eds.). *Handbook of Satisfiability*. IOS Press, 2009.
- D. Kroening and O. Strichman. *Decision Procedures (An Algorithmic Point of View)*. Springer, 2016.

# Basic Terminology

# Analysis and Verification

- ❖ **Verification** is a process of checking whether a given **system** (a real system or a model) satisfies a given **correctness specification** (property).
  - For example, given a concurrent program, we ask whether it is deadlock-free.
  - A verification method provides (ideally) **yes/no answers** having the sense that the system is/is not correct wrt. the given specification.
  - The yes/no answer may be complemented by some **diagnostic information** (e.g., when a deadlock is possible, a run leading to some deadlock situation is provided, or a certificate/proof that no deadlock is possible is produced).

# Analysis and Verification

❖ **Verification** is a process of checking whether a given **system** (a real system or a model) satisfies a given **correctness specification** (property).

- For example, given a concurrent program, we ask whether it is deadlock-free.
- A verification method provides (ideally) **yes/no answers** having the sense that the system is/is not correct wrt a given specification.
- The yes/no answer may be complemented by some **diagnostic information** (e.g., when a deadlock is possible, a run leading to some deadlock situation is provided, or a certificate that is not possible is produced).

❖ **Analysis** finds answers to **more general questions** about a given system—either questions which are not Boolean, or even if they are, they do not necessarily directly speak about correctness of the system.

- For example, given a concurrent system, we ask whether there is some ordering over the shared resources in which they are always locked.
- Answers provided by analysis may be an **input for further reasoning** leading to the verification of the system. However, they can also be used for completely **different purposes**, e.g., for optimisation, synthesis, code generation, etc.

# Formal Methods

- ❖ Verification and analysis methods include:
  - “bug hunting”: simulation, testing and dynamic analysis, some forms of static analysis, bounded model checking, ...
  - formal analysis and verification: model checking, some forms of static analysis, theorem proving, ...
  
- ❖ Formal methods are naturally based on formal, mathematical roots—*this does not imply that they are necessarily manual and not usable by ordinary users (!)*.
  
- ❖ Unlike other approaches, formal verification is (at least potentially) capable of proving correctness of a given system wrt. a given specification—and not just disprove the property based on some observed behaviour.
  
- ❖ Similarly, formal analysis is capable of providing answers universally (conservatively) covering all possible behaviours of a given system (not just giving existential answers—though perhaps extrapolated—based on some observed behaviours).

# The Ideal of Formal Verification

- ❖ **Full automation**: no human help needed.
- ❖ **Soundness**: if a verification method claims that a system is correct wrt. a given specification, it is indeed correct.
- ❖ **Completeness**: if a verification method claims that a system is not correct, there is indeed an error in the system—i.e., no **false alarms** (false positives).



# The Ideal of Formal Verification

- ❖ **Full automation**: no human help needed.
- ❖ **Soundness**: if a verification method claims that a system is correct wrt. a given specification, it is indeed correct.
- ❖ **Completeness**: if a verification method claims that a system is not correct, there is indeed an error in the system—i.e., no **false alarms** (false positives).
  - Some sources also require that a complete method must always terminate; sometimes **termination** is considered as an independent property.
  - Likewise, some sources tolerate that an **indefinite answer** “don’t know” may appear, some exclude it.

# The Ideal of Formal Verification

- ❖ **Full automation**: no human help needed.
- ❖ **Soundness**: if a verification method claims that a system is correct wrt. a given specification, it is indeed correct.
- ❖ **Completeness**: if a verification method claims that a system is not correct, there is indeed an error in the system—i.e., no **false alarms** (false positives).
  - Some sources also require that a complete method must always terminate, sometimes **termination** is considered as an independent property.
  - Likewise, some sources tolerate that an **indefinite answer** “don’t know” may appear, some exclude it.
- ❖ **Hard to achieve in general**:
  - the **state explosion problem**—i.e., an exponential growth of the number of reachable states wrt. the source description of a finite-state system:
    - `int n`; can have  $2^{32}$  (or  $2^{64}$ ,  $2^{128}$ , ...) possible values,
    - $m$  such variables:  $2^{m \cdot 32}$  (or  $2^{m \cdot 64}$ ,  $2^{m \cdot 128}$ , ...) possible values,
    - $n$  concurrent processes each with  $m$  states can generate  $m^n$  states,
  - **undecidability**: it suffices to have two *unbounded* integer variables, operations `++` and `--`, and branching according to equality with zero.

# Relaxing the Ideal

- ❖ A verification method needs **not guarantee termination** and/or can produce **false alarms**.
- ❖ Alternatively, a method is allowed to stop with a “**don't know**” answer and/or becomes **not fully automated** (i.e., some human help is required).
- ❖ Sometimes, even soundness is sacrificed to efficiency leading to an **error detection method** with formal verification roots.
  - For example, possible errors are **ranked** according to chances they are real, and only warnings of at least some rank are shown.
- ❖ Note that even **if full formal verification of a system fails, it may still be useful as it can find some errors** in the mean time.
- ❖ The **errors found by formal methods can differ** from those found by other methods due to other principles on which the methods work.
  - Hence, it is often a good idea **to use as many different approaches as possible**.

# Relaxing the Ideal

- ❖ A verification method needs **not guarantee termination** and/or can produce **false alarms**.
- ❖ Alternatively, a method is allowed to stop with a “**don't know**” answer and/or becomes **not fully automated** (i.e., some human help is required).
- ❖ Sometimes, even soundness is sacrificed to efficiency leading to an **error detection method** with formal verification roots.
  - For example, possible errors are **ranked** according to chances they are real, and only warnings of at least some rank are shown.
- ❖ Note that even **if full formal verification of a system fails, it may still be useful as it can find some errors** in the mean time.
- ❖ The **errors found by formal methods can differ** from those found by other methods due to other principles on which the methods work.
  - Hence, it is often a good idea **to use as many different approaches as possible**.

# Soundness

- ❖ Introduced in [Livshits *et al*, CACM'15].
- ❖ Soundness not necessary for most modern analysis applications:
  - many clients can tolerate unsoundness.
- ❖ A **soundy analysis** aims to be **as sound as possible** without excessively compromising precision and/or scalability.
- ❖ One may **distinguish**
  - analyses that **are soundy**, i.e., mostly sound, with specific, well-identified unsound choices (e.g., reflection when analysing Java code),
  - analyses that **do not concern themselves with soundness at all**.

# Systems and Properties To Be Checked

# Systems To Be Checked

- ❖ Systems to be verified/analysed can be classified in many different ways:
  - **Real systems** (SW in common programming languages, HW in VHDL, Verilog, ...) and **models** (based on process algebras, Petri nets, specialised modelling and specification languages like SMV, Promela, Simulink, EARS, CLEAR, OPM ...).
    - Note: we may distinguish high-level specification of systems and specification of properties (next slide).
  - **Finite-state** and **infinite-state systems**—dealing with parameterisation, unbounded recursion (stacks), queues (channels), counters (integer variables), dynamic instantiation, continuous phenomena (time in **real-time systems**, ...), ...
  - **Sequential** and **concurrent systems**—in the latter case, **asynchronous or synchronous**, with **interleaving or true concurrency**, ...

# Systems To Be Checked

- ❖ Systems to be verified/analysed can be classified in many different ways:
  - **Real systems** (SW in common programming languages, HW in VHDL, Verilog, ...) and **models** (based on process algebras, Petri nets, specialised modelling and specification languages like SMV, Promela, Simulink, EARS, CLEAR, OPM ...).
    - Note: we may distinguish high-level specification of systems and specification of properties (next slide).
  - **Finite-state** and **infinite-state systems**—dealing with parameterisation, unbounded recursion (stacks), queues (channels), counters (integer variables), dynamic instantiation, continuous phenomena (time in **real-time systems**, ...), ...
  - **Sequential** and **concurrent systems**—in the latter case, **asynchronous or synchronous**, with **interleaving or true concurrency**, ...
  - **Control-intensive** and **data-intensive systems**.



# Systems To Be Checked

- ❖ Systems to be verified/analysed can be classified in many different ways:
  - **Real systems** (SW in common programming languages, HW in VHDL, Verilog, ...) and **models** (based on process algebras, Petri nets, specialised modelling and specification languages like SMV, Promela, Simulink, EARS, CLEAR, OPM ...).
    - Note: we may distinguish high-level specification of systems and specification of properties (next slide).
  - **Finite-state** and **infinite-state systems**—dealing with parameterisation, unbounded recursion (stacks), queues (channels), counters (integer variables), dynamic instantiation, continuous phenomena (time in **real-time systems**, ...), ...
  - **Sequential** and **concurrent systems**—in the latter case, **asynchronous or synchronous**, with **interleaving or true concurrency**, ...
  - **Control-intensive** and **data-intensive systems**.
  - **Transformational** and **reactive systems**:
    - **Transformational systems** are designed to transform a certain input to a certain output (compilers, database queries, ...).
    - **Reactive systems** typically transform an infinite sequence of inputs to an infinite sequence of outputs (control systems, operating systems and their sub-systems, ...).

# Properties To Be Checked

- ❖ Properties to be verified/analysed can again be classified in various ways:
  - The **specification language**:
    - the property may be fixed for a given method/tool/plugin,
    - a choice out of a list,
    - various kinds of **labels/inscriptions** (assertions, end-state labels, progress labels, invariants, ...),
    - **automata** (FSA, Büchi automata, ...), **temporal logic formulae** (LTL, CTL, CTL\*,  $\mu$ -calculus, ...),
    - various specialised textual or graphical specification languages (PSL),
    - ...

# Properties To Be Checked

❖ Properties to be verified/analysed can again be classified in various ways:

- The **specification language**:

- the property may be fixed for a given method/tool/plugin,
- a choice out of a list,
- various kinds of **labels/inscriptions** (assertions, end-state labels, progress labels, invariants, ...),
- **automata** (FSA, Büchi automata, ...), **temporal logic formulae** (LTL, CTL, CTL\*,  $\mu$ -calculus, ...),
- various specialised textual or graphical specification languages (PSL),
- ...

- The underlying **notion of time**:

- **logical** vs. **physical**,
- **linear** vs. **branching**.

# Properties To Be Checked

- ❖ Properties to be verified/analysed can again be classified in various ways:
  - The **specification language**:
    - the property may be fixed for a given method/tool/plugin,
    - a choice out of a list,
    - various kinds of **labels/inscriptions** (assertions, end-state labels, progress labels, invariants, ...),
    - **automata** (FSA, Büchi automata, ...), **temporal logic formulae** (LTL, CTL, CTL\*,  $\mu$ -calculus, ...),
    - various specialised textual or graphical specification languages (PSL),
    - ...
  - The underlying **notion of time**:
    - **logical** vs. **physical**,
    - **linear** vs. **branching**.
  - The form of **counterexamples** (finite or infinite):
    - **safety** vs. **liveness** (or their mixture).

# Properties To Be Checked

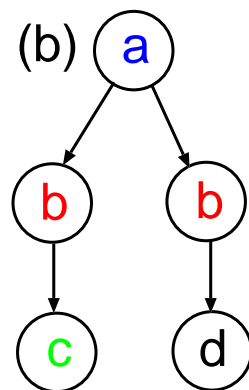
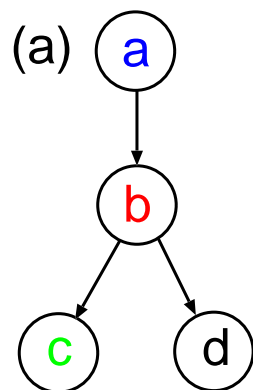
- ❖ Properties to be verified/analysed can again be classified in various ways:
  - The **specification language**:
    - the property may be fixed for a given method/tool/plugin,
    - a choice out of a list,
    - various kinds of **labels/inscriptions** (assertions, end-state labels, progress labels, invariants, ...),
    - **automata** (FSA, Büchi automata, ...), **temporal logic formulae** (LTL, CTL, CTL\*,  $\mu$ -calculus, ...),
    - various specialised textual or graphical specification formalisms (PSL),
    - ...
  - The underlying **notion of time**:
    - **logical** vs. **physical**,
    - **linear** vs. **branching**.
  - The form of **counterexamples** (finite or infinite):
    - **safety** vs. **liveness** (or their mixture).
  - A special kind of properties to be checked are various forms of **equality/refinement/simulation** of two system versions.

# Notions of Time

- ❖ Logical time imposes a (partial) ordering on states/events in the behaviour of a system.
- ❖ Physical time allows one to measure how much time passed between two states/events.

# Notions of Time

- ❖ **Logical time** imposes a (partial) ordering on states/events in the behaviour of a system.
- ❖ **Physical time** allows one to measure how much time passed between two states/events.
- ❖ **Linear time** allows one to speak about particular linear traces of the state space of a given system only.
  - In all traces, *b* must happen immediately followed by *c*. (1)
  - In all traces, *b* must happen immediately followed by *c* or *d*. (2)
- ❖ **Branching time** allows one to quantify (existentially and universally) over the possible futures of a given state. We view the state space unfolded into an infinite tree.
  - *b* must happen, and immediately after, *c* may happen and *d* may happen. (3)



	(a)	(b)
(1)	No	No
(2)	Yes	Yes
(3)	Yes	No

# Linear Time Safety and Liveness

- ❖ **Safety** properties require that **something bad never happens**.
  - Their violation always has a **finite witness**. In other words, if there is a counterexample, there is a finite counterexample.
  - **Examples:**
    - *Processes mutually exclude each other when accessing a shared variable.*
    - *A program never returns a wrong result.* (partial correctness)
    - *No null pointer exception can arise.*
    - *A signal  $s_1$  never arrives before a signal  $s_2$ .*



# Linear Time Safety and Liveness

- ❖ **Safety** properties require that **something bad never happens**.
  - Their violation always has a **finite witness**. In other words, if there is a counterexample, there is a finite counterexample.
  - **Examples:**
    - *Processes mutually exclude each other when accessing a shared variable.*
    - *A program never returns a wrong result.* (partial correctness)
    - *No null pointer exception can arise.*
    - *A signal  $s_1$  never arrives before a signal  $s_2$ .*
  
- ❖ **Liveness** properties require that **something good eventually happens**.
  - Their violation can only have an **infinite witness**, or a **finite but complete witness**, i.e., a witness that cannot be extended any more.
  - **Examples:**
    - *A program terminates for any input.* (total correctness)
    - *A process can never starve.*
    - *After a signal  $s_1$  is received, a signal  $s_2$  is eventually sent.*
  - Liveness is typically **harder to verify** than safety.
  
- ❖ **General properties** can mix (conjoin) safety and liveness.

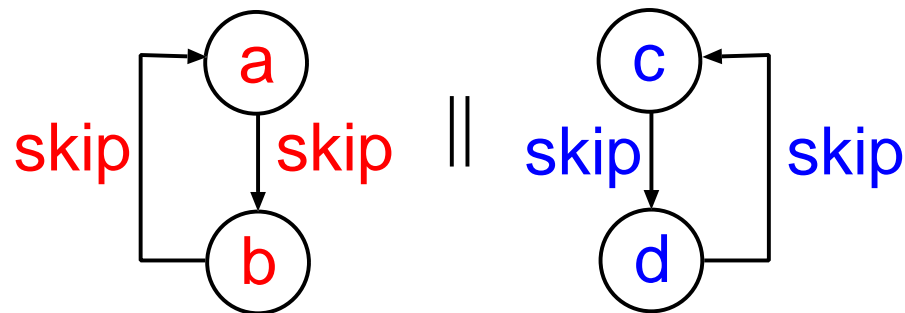
# Branching Time Safety and Liveness

- ❖ The situation is **more complicated**: a computation tree can be infinite and incomplete at the same time!
  
- ❖ One can distinguish [Manolios, Trefler – LICS 2001]:
  - **Universally safe properties**: linear time safety over all computations.
  - **Existentially safe properties**: guarantee at least one safe computation.
  - **Universally live properties**: linear time liveness properties over all computations.
  - **Existentially live properties**: linear time liveness for at least one computation.
  
- ❖ **General properties** are **intersections** of some of the above kinds of properties.

# Fairness

❖ In order to exclude practically infeasible, trivial counterexamples to liveness properties, one usually has to apply some fairness assumptions.

- For example, assume that we have two concurrent, infinitely looping processes, not requesting anything from their environment.



- We ask whether each of them will always eventually do a step.
- We somehow naturally expect the answer to be **yes**, but this is based on that we implicitly assume that the processes are scheduled by **some fair scheduler**.
- However, if we do not build the scheduler into the system, the answer will be **no**.
- Alternatively, we can perform the verification under a **suitable fairness assumption**—either expressed as a part of the property being checked or hard-wired into the verification algorithm.

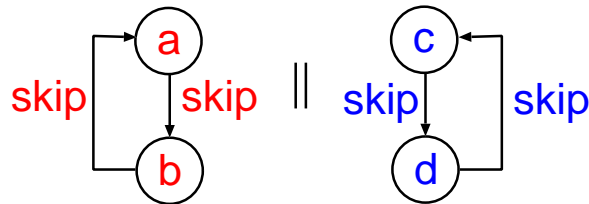
# Fairness

❖ Fairness limits sources of non-determinism in a system (e.g., in the scheduling or in reading random data from the environment).

❖ Two most common kinds of fairness:

- Weak fairness:

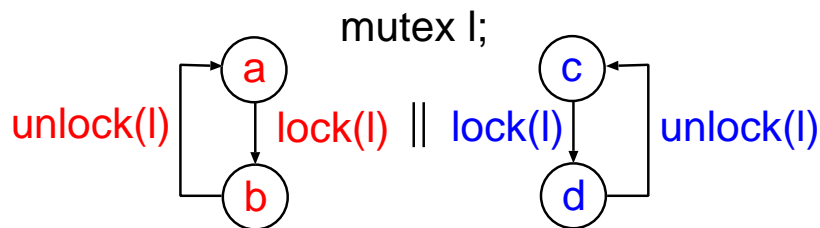
- an action that is eventually always enabled must always eventually be taken,



- enough for non-synchronizing processes.

- Strong fairness:

- an action that is always eventually enabled must always eventually be taken,
- needed when dealing with fair resource allocation,



- more complicated to handle.

# Different Approaches to Formal Verification and Analysis

# Model Checking

[Clarke, Emerson 81], [Quielle, Sifakis 81]

❖ An algorithmic approach of checking whether a given system satisfies a given property through a systematic exploration of the **state space** of the system.

- The system may be a **real system** or a **model**.
  - Sometimes the use of models is stressed.
- Usually, **finite-state systems** are considered.
  - Dealing with infinite-state systems is less common.
- Properties classically specified using **temporal logics** (LTL, CTL, CTL\*, ...); other forms of specification are also possible.

# Model Checking

## ❖ Advantages include:

- a (relatively) high degree of automation—fully automated up to the need of modelling the system, its parts, and/or its environment,
- (relatively) easy to use,
- quite general as for the systems and their properties that can be checked,
- provides counterexamples.

# Model Checking

## ❖ Advantages include:

- a (relatively) high degree of automation—fully automated up to the need of modelling the system, its parts, and/or its environment,
- (relatively) easy to use,
- quite general as for the systems and their properties that can be checked,
- provides counterexamples.

## ❖ Disadvantages:

- the **state explosion problem** (and dealing with infinite-state spaces),
- a need to model the environment of the checked (sub-)system.



# Model Checking

- ❖ Dealing with the **state explosion problem**—just a brief overview, we will get back to (at least some of) the mentioned approaches later on:
  - **Efficient storage** of state spaces (hierarchical storage of states, BDDs, ...).
  - State space **reductions** (symmetries, partial-order reduction, ...).
  - **Abstraction**, counterexample-guided abstraction refinement (CEGAR).
  - **Compositional methods**, assume-guarantee reasoning.
  - [ **Bounded model checking**: exploring the state space up to some bound only (sacrificing soundness), may leverage advances in SAT/SMT solving. ]
- ❖ Especially suitable for **reactive, concurrent, control-intensive systems**.
- ❖ Supported by many **tools**, including industrial-strength tools:
  - Spin, Divine, Blast, CPAchecker, CBMC, JBMC, Ultimate Automizer, JPF (NASA), NuXMV, ABC, Incisive Formal Analysis (Cadence), Questa (Siemens), VC Formal (Synopsys), Uppaal, Prism, ...

# Static Analysis

- ❖ Collects some information about the behaviour of a system from its **source code** without actually executing it under its original semantics.
- ❖ This description is very general, it can include even model checking and theorem proving (at least in some of their forms), which is sometimes (but not usually) done.
- ❖ Static analyses **range** from simple syntactic checks to iterative fixpoint computations over an abstraction of the examined system (e.g., into a form of equations that need to be solved).
- ❖ Different **forms of static analysis**:
  - bug pattern (anti-pattern) searching,
  - dataflow analysis,
  - constraint-based analysis,
  - (extended) **type analysis** (type and effect systems),
  - abstract interpretation,
  - **symbolic execution**, ..., mixtures of the above.

# Static Analysis

❖ Static analysis is **not exclusively** intended for checking correctness of systems only—it is used also for optimisation, code generation, code understanding, ...

# Static Analysis

❖ Static analysis is **not exclusively** intended for checking correctness of systems only—it is used also for optimisation, code generation, code understanding, ...

❖ **Advantages** include:

- can often handle **very large systems**,
- does often not need a model of the environment (input/output, libraries, other modules),
- a high degree of automation (unless one needs to build a custom analysis).

# Static Analysis

- ❖ Static analysis is **not exclusively** intended for checking correctness of systems only—it is used also for optimisation, code generation, code understanding, ...
- ❖ **Advantages** include:
  - can often handle **very large systems**,
  - does often not need a model of the environment (input/output, libraries, other modules),
  - a high degree of automation (unless one needs to build a custom analysis).
- ❖ **Disadvantages**:
  - can produce many **false alarms**
    - construction of more and more precise analyses leads to similar efficiency problems like in model checking,
    - often resolved by accepting unsoundness,
  - various analyses are often **specialised** just for a certain specific task,
  - can have in principle **restricted applicability** for some properties of interest (e.g., not everything can be simply expressed by a bug pattern).

# Static Analysis

- ❖ There exist **many tools** for static analysis too:
  - FindBugs/SpotBugs, Synopsys/Coverity, Klocwork, CodeSonar (GramaTech), PolySpace, Code Analysis in VisualStudio, AbsInt/Astrée, Clang Static Analyser, gcc static analysis, Cppcheck, cppclean, Sparse, Meta/Facebook Infer, Facebook SPARTA, Frama-C, Predator, KLEE, Loopus, Cost, Symbiotic, ...

# Theorem Proving

- ❖ Deductive verification often similar to the classical mathematical way of proving theorems starting with axioms and inferring further facts using rules of correct inference.
  
- ❖ Advantages and disadvantages:
  - Very general.
  - Usually **semi-automated**, requires a significant manual effort.
  - Problems with diagnostic information for incorrect systems.
  
- ❖ There exist many **interactive theorem provers** including:
  - PVS, Coq, Hol, Isabelle, ACL2, Forte, ...
  
- ❖ Used, e.g., by Intel, AMD to verify complex designs of various arithmetic units (sometimes combined with model checking). Ongoing attempts to arrive to formally verified operating system kernels (seL4), compilers (CompCert), ...

# Theorem Proving

- ❖ Recently there has been a lot of progress on fully automated **decision procedures** for various decidable theories:
  - **propositional logic** (SAT solving),
  - various **first-order theories** (SMT solving)
    - uninterpreted functions,
    - linear integer/real arithmetic,
    - theories of arrays, bitvectors, ...
  - WS1S, WS2S, ...
- ❖ There exist many **tools** implementing various decision procedures:
  - **SAT solvers**: Kissat, CaDiCaL, MapleSAT, glucose, ...
  - **SMT solvers**: Z3, CVC5, MathSat, Yices2, Boolector, SMTinterpol, ...
  - **WS1S/WS2S**: Mona, Gaston, ...
- ❖ There exist **fully automated theorem provers** for **undecidable logics** as well: e.g. Vampire for predicate logic (no guarantee of termination).



# Theorem Proving

- ❖ Decision procedures often serve as a **back-end** for other verification approaches, e.g.:
  - **Predicate abstraction** for model checking.
  - Deductive verification based on discharging **verification conditions**.
    - Code is **annotated** by loop invariants, pre-/post-conditions of the program and procedures, and/or various assumptions.
    - Annotations can be **(partially) obtained automatically**—e.g., from analysing runs of the system and generalising them, solving constraints describing loops, using LLMs, etc.
    - Annotations allow verification to concentrate on loop-free fragments.
    - **Verification conditions**: checking that when one starts under some precondition and executes a loop-free code fragment, the postcondition will be met.
    - Used, e.g., in attempts to verify operating system kernels—PikeOS, Microsoft's Hypervisor (see the tool VCC).
  - **Symbolic execution** – iterating over longer and longer program paths possibly leading to an error, encoding them as formulae, checking satisfiability.
    - Klee, Symbiotic, SPF, Java Ranger, JDart, ...