Theoretical computer science

Tutorial - week 15

April 29, 2021

INNOPOLIS UNIVERSITY

Outline

- Static analysis
- Model checking
    - Motivation
    - Abstraction techniques
    - Verifiable properties
- Formal specification and verification

What is Static Analysis?

What is Static Analysis?

- Type checking
- Control flow analysis
- Dataflow analysis
- …

:(

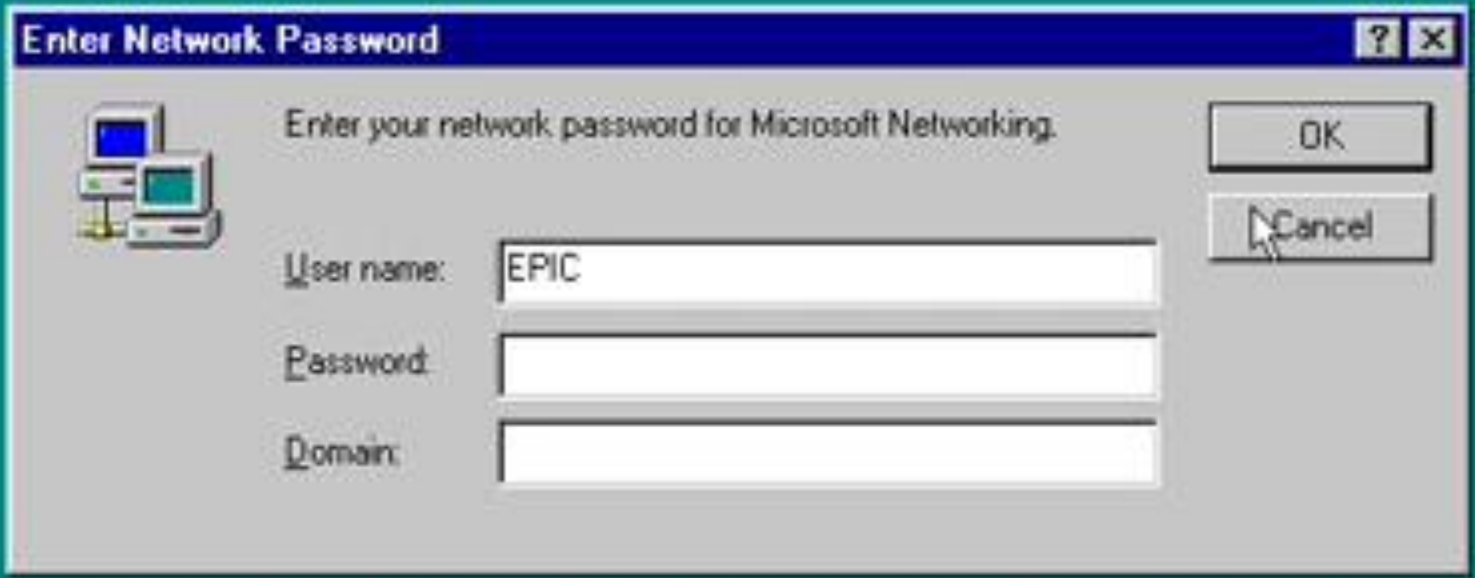
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: CRITICAL_PROCESS_DIED



Forgotten you password ?

No problem

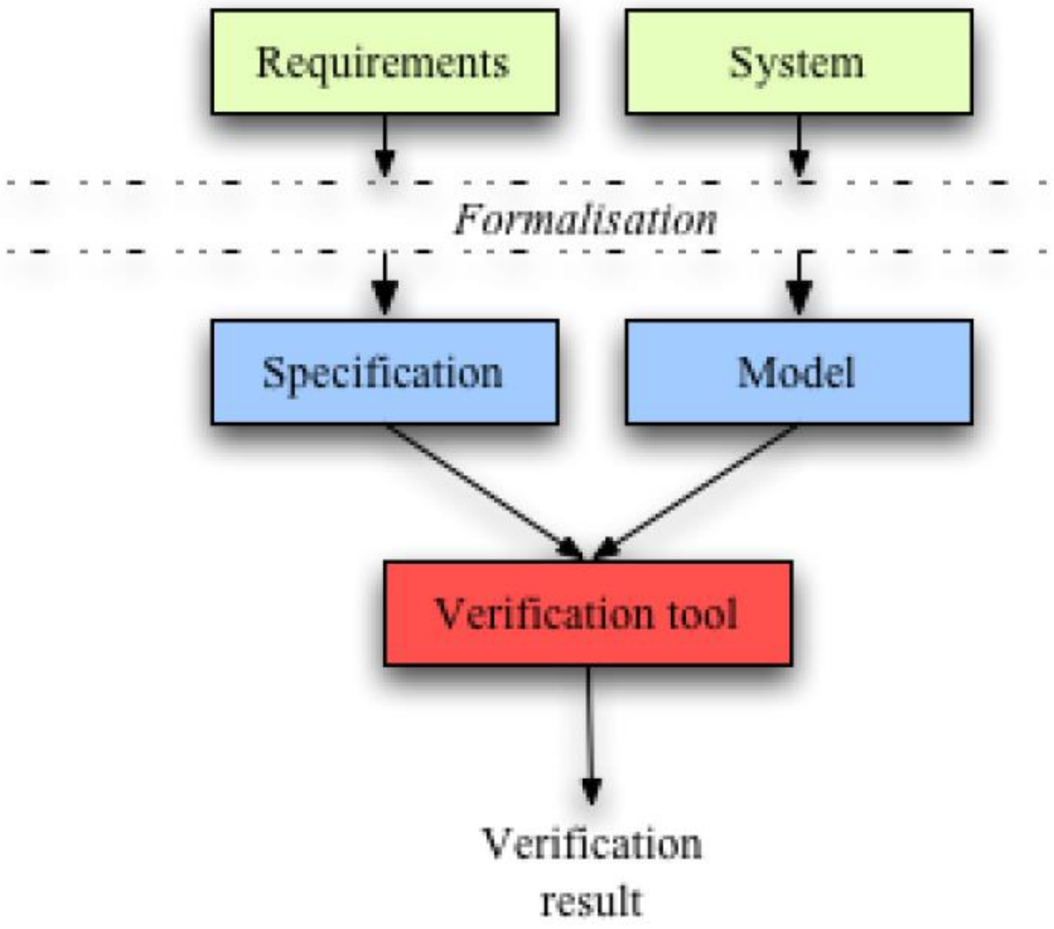
Motivation

- 50% - 80% of the time and resources are devoted to detecting and correcting mistakes
- Software that controls critical processes cannot fail
  - Cars, planes, medical equipment, air traffic control, etc.
- Some stuff is really complicated - like distributed systems, DBMS

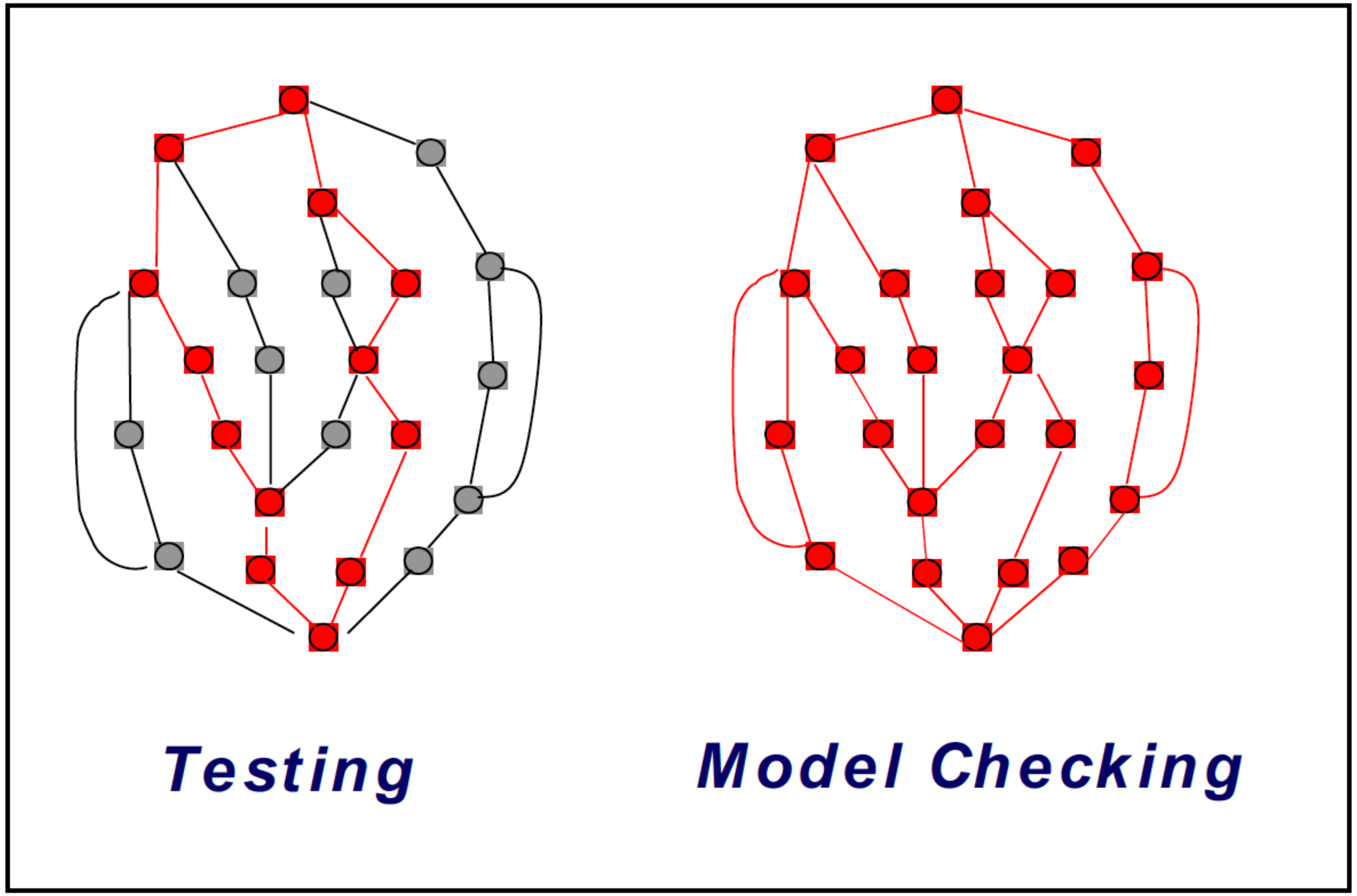
Recap from the lecture

- Specify program model and exhaustively evaluate that model against a specification
  - Check that properties hold:

    Ex.: *The system* ***will always*** *break* ***when*** *there is an obstacle in front*
  - Produce counter examples when properties do not hold



Model Checking Coverage



Program Model Checking - A Practitioner's Guide, Mansouri-Samani et al, 2008

When to Use Model Checking?

When to Use Model Checking?

- Safety Critical
- Vast Cost (space systems…)
- Concurrency (including distributed systems)

- Also!!!
    - Relatively simple (abstracted) state space, e.g., device driver
    - Generation of test cases, e.g., MCDC coverage

Model Building and Abstraction

Abstraction

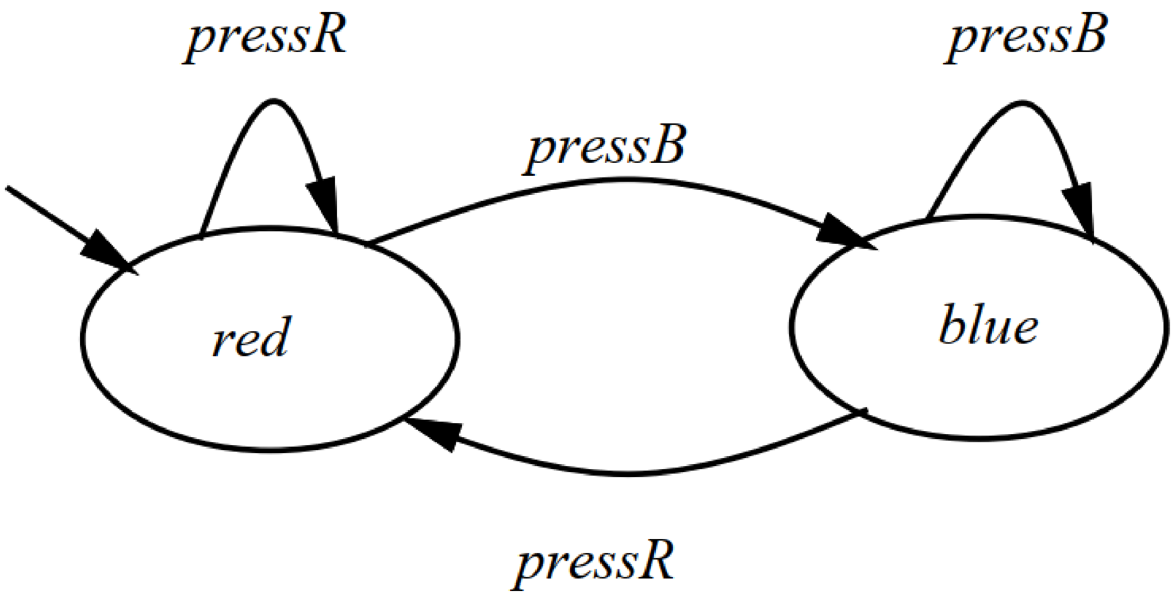
- Identify the Behaviors and Values ("Properties") that you care about
  - Temperature controller must stay between specified bounds
  - Altitude must stay above 0
  - Message queue must always be read
  - Database must never deadlock

Example: Microwave oven

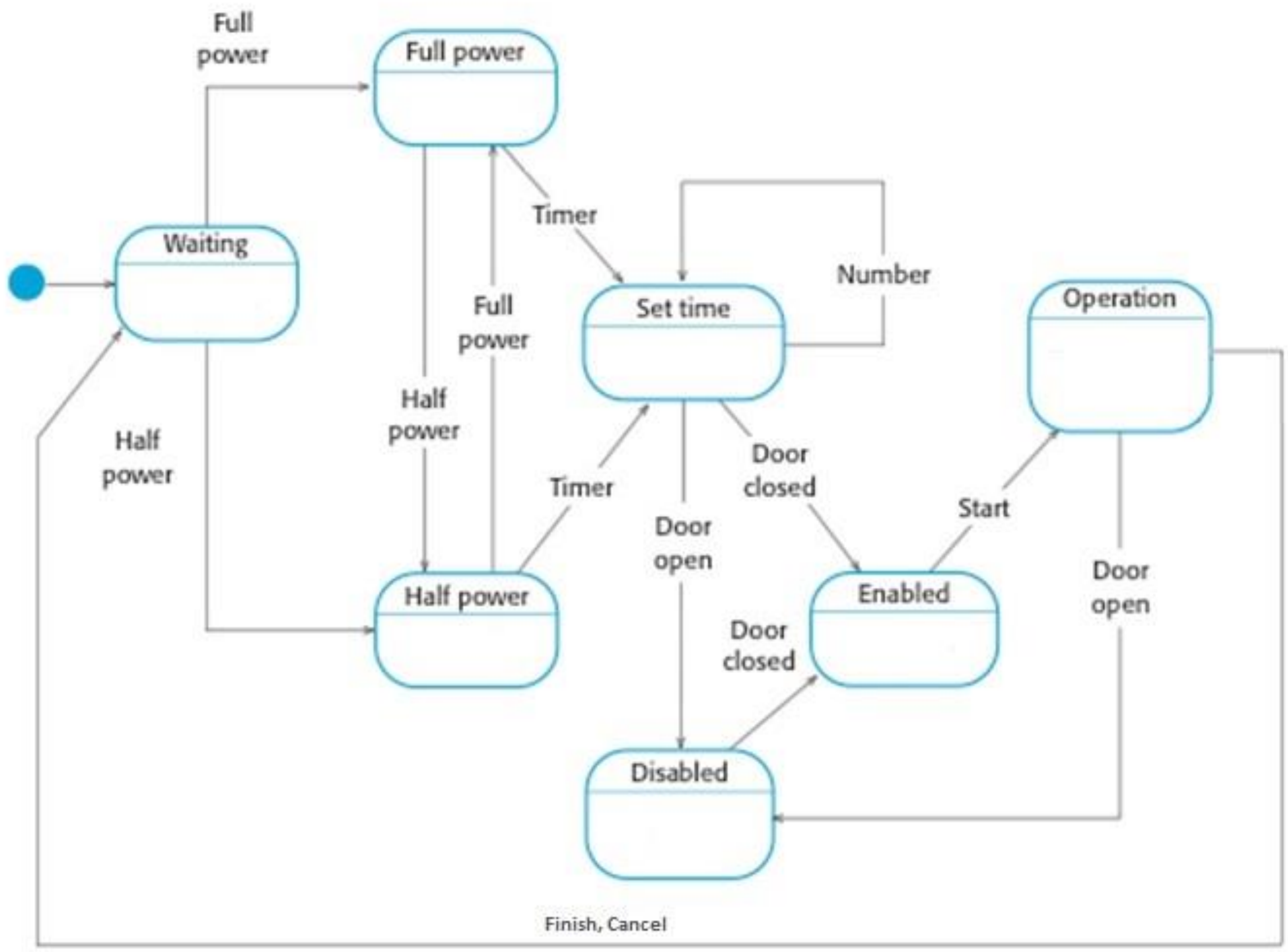
- Basically, a state machine
  - Starts in initial state and changes based on user interaction
- We need to know
  - What *states* the microwave oven can be in
  - How the model *transition* from one state to another
  - What the model starts with

Finite State Machine

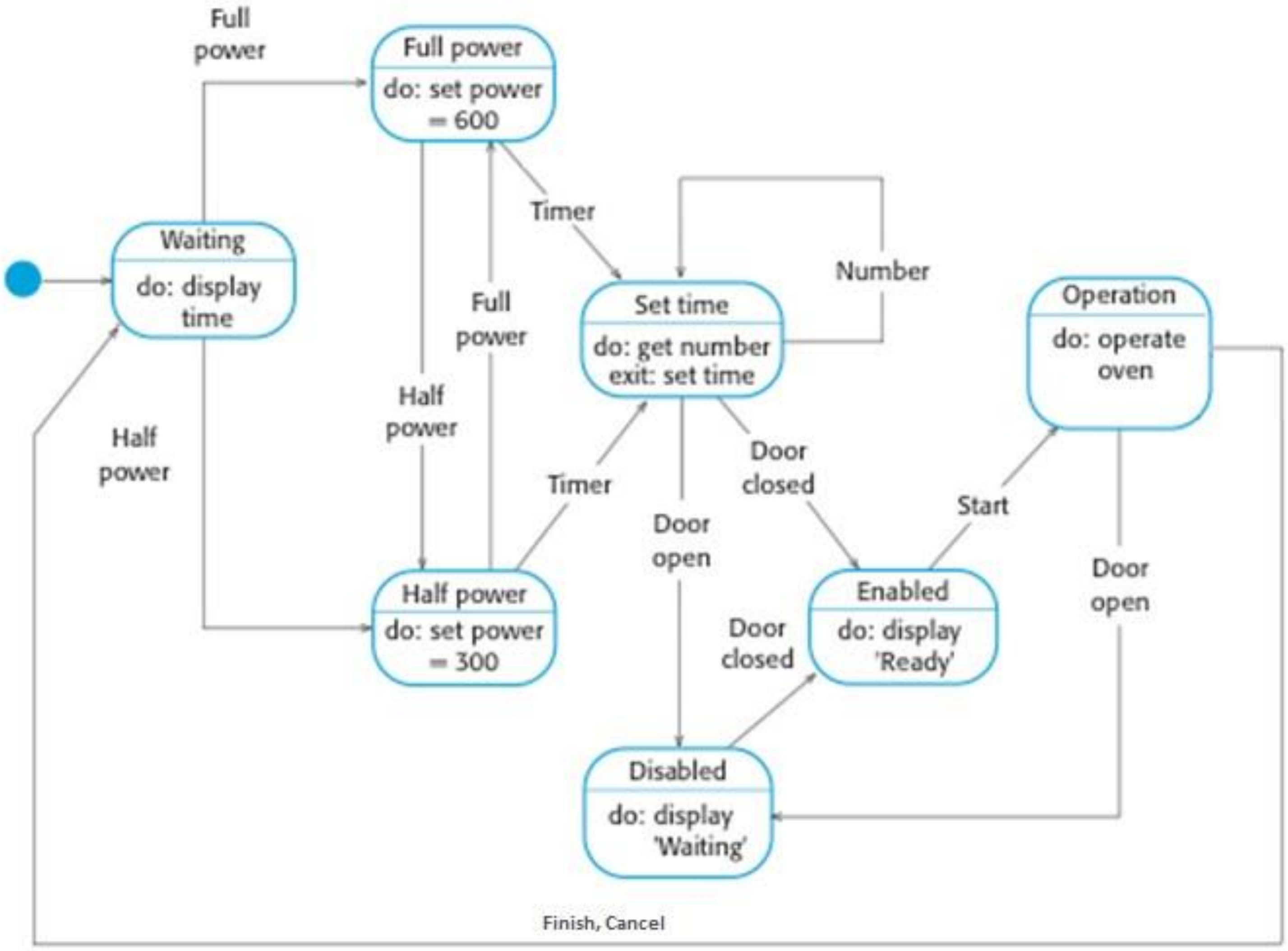
- S is a finite set of **states**,
- I ⊆ S is a finite set of **initial states**,
- A is a finite set of **actions**, and
- δ ⊆ S×A×S is a **state transition relation**



Finite State Machine: example



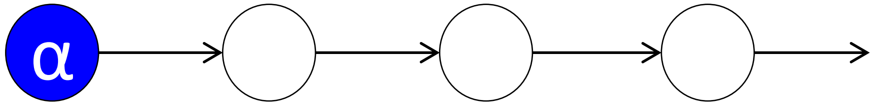
Finite State Machine: example



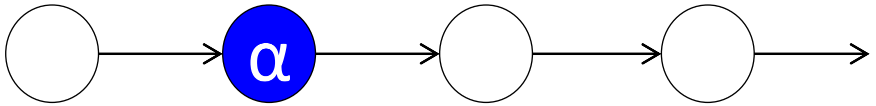
Property Specification

Linear Temporal Logic (LTL)

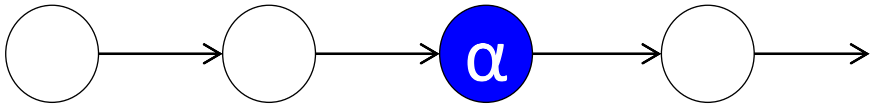
- **α:** α holds in current state



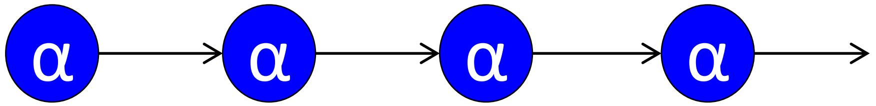
- **X α:** α holds in the next state
  - Sometimes written ∘ α



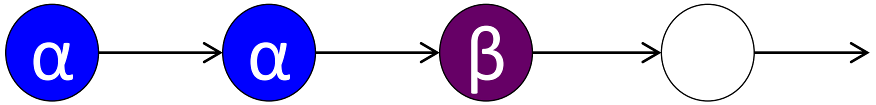
- **F α:** α holds eventually
  - Sometimes written ◊ α



- **G α:** α holds from now on
  - Sometimes written □ α



- **(α U β):** α holds until β
  - There is also *weak until*

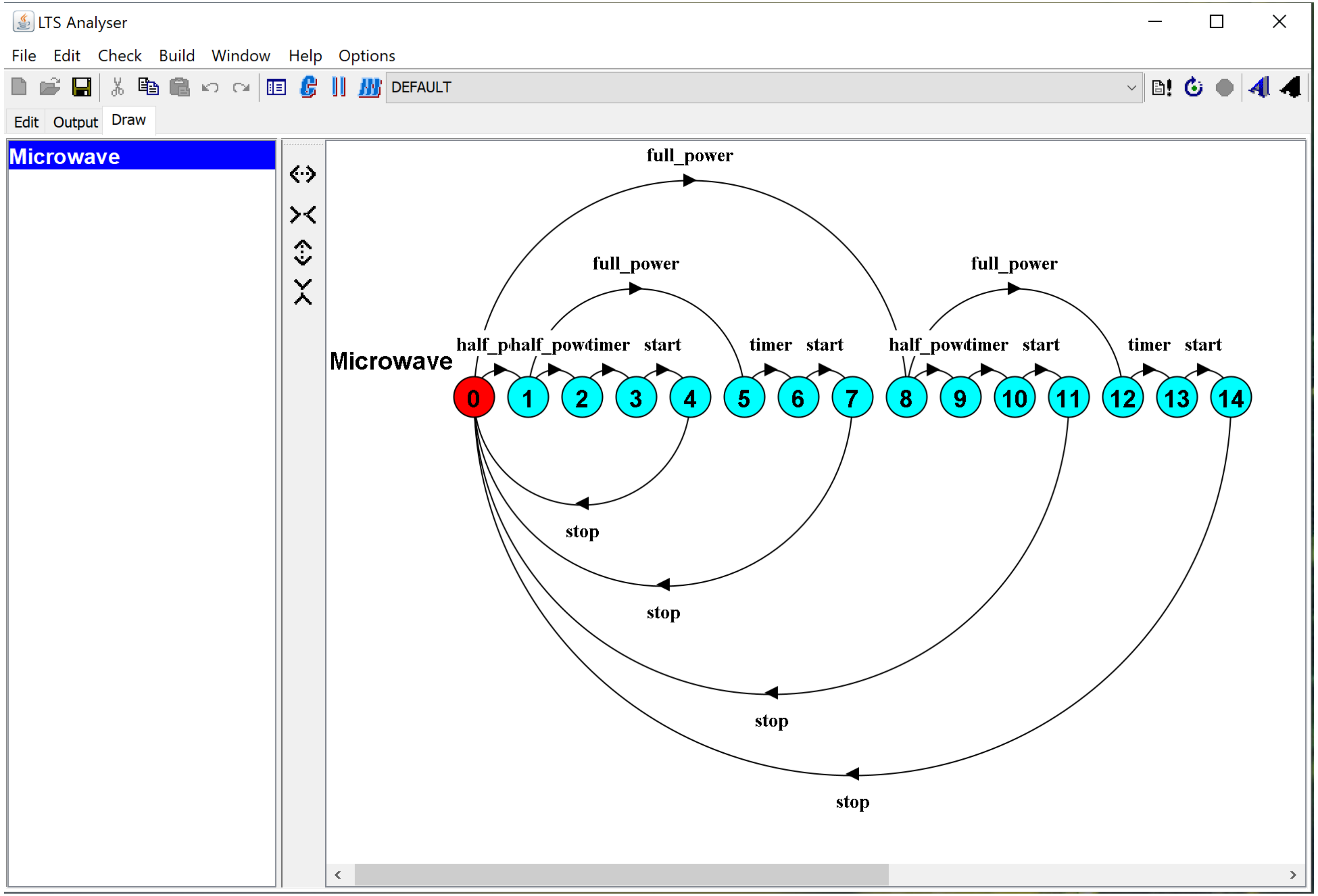


LTL refresher

- The microwave oven will not operate unless the door is closed

- The operating microwave oven will always eventually stop

LTSA


LTS Analyser
File Edit Check Build Window Help Options
DEFAULT
Edit Output Draw
Microwave
full_power
full_power
full_power
Microwave
half_p half_pow timer start
timer start
half_pow timer start
timer start
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
stop
stop
stop
stop

Two fundamental concepts

Programs as data

- Programs are just trees/graphs!
- …and CS has lots of ways to analyze trees/graphs

Abstraction

- Elide details of a specific implementation.
- Capture semantically relevant details; ignore the rest.

Formal specification and verification

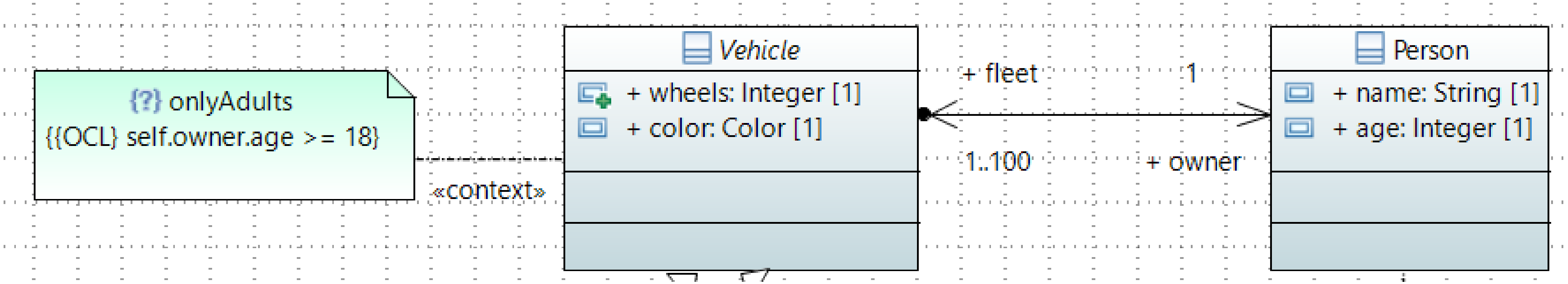
Semi-formal specification

Object Constraint Language (OCL)

Nat. lang.: *Vehicles can be owned by a Person who is at least 18 years old*

Predicate logic: $\forall$ *v:Vehicle • v.owner.age >= 18*

OCL: *context Vehicle inv: self.owner.age >= 18*



Specification languages

- Z notation

  is a formal **specification** language used for describing and modelling computing systems

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Hoar logic

- The Hoare triple: **{P} S {Q}**
  - P and Q are predicates
  - S is the program

<u>Examples:</u>

- { true } x := 5 { x = 5 }
- { x > -1 } x := x * 2 + 3 { x > 1 }
- { x = y } x := x + 3 { x = y + 3 }
- { i > 0} result j / i { result = j / i }

```
/*@ requires i > 0 */
/*@ ensures result == j / i */
public void div(int i, int j) {
  return j/i;
}
```

Design by Contract

- Preconditions
- Post-conditions
- Invariants:
  - Class
  - Loop invariants (variants?)

Example:

requires, ensures

're- and post-conditions for method can be specified.

```
/*@ requires amount >= 0;
    ensures  balance == \old(balance-amount)  &&
             \result == balance;
  @*/
 public int debit(int amount) {
   ...
 }
```

lere `\old(balance)` refers to the value of `balance`
ıefore execution of the method.

invariant

Invariants (aka *class* invariants) are properties that must be maintained by all methods, e.g.,

```
public class  Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
   /*@ invariant 0 <= balance &&
                     balance <= MAX_BAL;
     @*/
  ...
```

Invariants are implicitly included in all pre- and postconditions.

Invariants must *also* be preserved if exception is thrown!

Design by Contract

- Provides a short notation
- Identifies ambiguity
- Makes you ask the right questions
- Subject to proofs
- Proofs can be machine-checked

Design by Contract

https://en.wikipedia.org/wiki/Design_by_contract

Language support [edit]

Languages with native support [edit]

Languages that implement most DbC features natively include:

- Ada 2012
- Ciao
- Clojure
- Perl6
- Cobra
- D[9]
- Eiffel
- Fortress
- Kotlin
- Mercury
- Nice
- Oxygene (formerly Chrome and Delphi Prism[10])
- Racket (including higher order contracts, and emphasizing that
- Sather
- SPARK (via static analysis of Ada programs)
- Spec#
- Vala
- VDM

https://en.wikipedia.org/wiki/Design_by_contract

Languages with third-party support [edit]

Various libraries, preprocessors and other tools have been developed for existing programming languages without native Design by Contract support:

- Ada, via GNAT pragmas for preconditions and postconditions.
- C and C++, via Boost.Contract, the *DBC for C* preprocessor, *GNU Nana*, *eCv* and *eCv++* formal verification tools, or the Digital Mars C++ compi design by contract.
- C# (and other .NET languages), via *Code Contracts*[12] (a Microsoft Research project integrated into the .NET Framework 4.0)
- Groovy via GContracts
- Go via dbc
- Java:
  - Active:
    - OVal with AspectJ
    - Contracts for Java (Cofoja)
    - Java Modeling Language (JML)
    - Bean Validation (only pre- and postconditions)[13]
    - valid4j
  - Inactive/unknown:
    - Jtest (active but DbC seems not to be supported anymore)[14]
    - iContract2/JContracts
    - Contract4J
    - jContractor
    - C4J
    - Google CodePro Analytix
    - SpringContracts for the Spring Framework
    - Jass
    - Modern Jass (successor is Cofoja)[15][16]
    - JavaDbC using AspectJ
    - JavaTESK using extension of Java
    - chex4j using javassist
    - highly customizable java-on-contracts
- JavaScript, via AspectJS (specifically, AJS_Validator), Cerny.js, ecmaDebug, jsContract, dbc-code-contracts or jscategory.
- Common Lisp, via the macro facility or the CLOS metaobject protocol.
- Nemerle, via macros.
- Nim, via macros.
- Perl, via the CPAN modules Class::Contract (by Damian Conway) or Carp::Datum (by Raphael Manfredi).
- PHP, via PhpDeal, Praspel or Stuart Herbert's ContractLib.
- Python, using packages like icontract, PyContracts, Decontractors, dpcontracts, zope.interface, PyDBC or Contracts for Python. A permanent char
- Ruby, via Brian McCallister's DesignByContract, Ruby DBC ruby-contract or contracts.ruby.
- Rust via the Hoare library
- Tcl, via the XOTcl object-oriented extension.

Formal verification

- AutoProof:

```
    transfer (amount: INTEGER; other: ACCOUNT)
            -- Transfer `amount' from this account to `other'.
        require
            amount_not_negative: amount >= 0
            amount_available: amount <= available_amount
            modify (Current, other)
        do
            balance := balance - amount
            other.deposit (amount)
        ensure
            withdrawal_made: balance = old balance - amount
            despoit_made: other.balance = old other.balance + amount
            same_credit_limit: credit_limit = old credit_limit
            other_same_credit_limit: other.credit_limit = old other.credit_limit
        end

invariant
    credit_limit_not_negative: credit_limit >= 0
    balance_not_below_credit: balance >= -credit_limit
```

Model checking & formal verification

Verify abstract models (Spin, Alloy, LTSA etc.)

+ abstract from unnecessary details
+ try out ideas before implementation
+ cheaper that actual implementation

– concern on correspondence of abstract model to implementation
– handling changing requirements

Verify actual design descriptions (Event-B, AutoProof etc.)

+ can handle large set of problems
+ verify correctness of implementation

– specification expressed formally
– requires high expertise