# Organizational Matters. Course Motivation


Mădălina Erașcu

West University of Timișoara and Institute e-Austria Timișoara
bvd. V. Parvan 4, Timișsoara, Romania

madalina.erascu@e-uvt.ro


*Based on: lecture notes Azadeh Farzan (University of Toronto) and Emina Torlak (University of Wahington)*

# Organizational Matters

`https://merascu.github.io/links/FMSD.html`

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Organizational Matters

`https://merascu.github.io/links/FMSD.html`

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Organizational Matters

https://merascu.github.io/links/FMSD.html

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Organizational Matters

`https://merascu.github.io/links/FMSD.html`

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Organizational Matters

`https://merascu.github.io/links/FMSD.html`

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Organizational Matters

`https://merascu.github.io/links/FMSD.html`

Attention: **Do not take this course if...**

- you do not know or do not want to learn English! (available for IR students)!
- you do not like logic (see Computational Logic course of Dr. Adrian Craciun) !
- you want an easy course !
- you are bad at working in a team!

# Motivation

- Software Validation: one of the toughest open problems in Computer Science.
- Verification has always been derived by academia
  - very rich theoretical basic: logics, algorithms, calculi, ...
  - a lot of room for pragmatism: theoretically-motivated heuristics

# Motivation

- Software Validation: one of the toughest open problems in Computer Science.
- Verification has always been derived by academia
  - very rich theoretical basic: logics, algorithms, calculi, ...
  - a lot of room for pragmatism: theoretically-motivated heuristics

# Motivation

- Software Validation: one of the toughest open problems in Computer Science.
- Verification has always been derived by academia
  - very rich theoretical basic: logics, algorithms, calculi, ...
  - a lot of room for pragmatism: theoretically-motivated heuristics

# Motivation

- Software Validation: one of the toughest open problems in Computer Science.
- Verification has always been derived by academia
  - very rich theoretical basic: logics, algorithms, calculi, ...
  - a lot of room for pragmatism: theoretically-motivated heuristics

# List of Bugs

- https://en.wikipedia.org/wiki/List_of_software_bugs

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile … make it more reliable
  - partial validation, intelligent testing, …
  - Next generation languages with better validation support.

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile ... make it more reliable
  - partial validation, intelligent testing, ...
  - Next generation languages with better validation support.

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile … make it more reliable
  - partial validation, intelligent testing, …
  - Next generation languages with better validation support.

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile ... make it more reliable
  - partial validation, intelligent testing, ...
  - Next generation languages with better validation support.

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile ... make it more reliable
  - partial validation, intelligent testing, ...
  - Next generation languages with better validation support.

# Our Holy Grail

- Make software (more) reliable.
  - Software is a product! – it needs industry standards.
  - A notion of certification for software is needed.
- Meanwhile ... make it more reliable
  - partial validation, intelligent testing, ...
  - Next generation languages with better validation support.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \land p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \land p \in \mathbb{Z} \land rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \land p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \land p \in \mathbb{Z} \land rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a,p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a,p,i,rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a,p,rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \land p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \land p \in \mathbb{Z} \land rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# What is (Formal) Verification

- Proving (in a formal way) that program satisfies a specification written in a logical language.
  - Formal models for programs.
  - Logics for specifications.
  - Algorithms for checking the model against the specification.

**Example**

```
int power (int a, int p)
      res = 1
      i = 0
      while i < p do
            i = i + 1
            rez = rez * a
      return rez
```

- Initially: $In(a, p) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z}$
- At each iteration of the loop: $Inv(a, p, i, rez) \iff a \in \mathbb{Z} \wedge p \in \mathbb{Z} \wedge rez = a^i$
- At loop exit: $Out(a, p, rez) \iff rez = a^p$

Use induction to prove the invariant and the specification.

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle [illegible]
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search space well
    - specification: CTL (and later LTL)
    - by far the most used for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Short History of Verification

In the 70s

- Proving programs correct
  - Floyd, Hoare, Dijkstra, ...
  - Philosophy: programmers write programs and prove them correct with a prover.
  - Failed
    - All or nothing approach: no way to find bugs.
    - heavily manual ... non-appealing!

Success Stories

- SPIN (Holzmann)
  - Explicit-state model checker
  - Heuristics to control state-space explosion
    - partial order reduction
    - hashing and approximate search
    - specification: LTL/automata
- SMV (Started by McMillan), later NuSMV
  - Symbolic model checker using binary decision diagrams (BDD)
  - handles large state spaces
    - heuristics to handle search spaces well
    - specification: CTL (and later LTL)
    - by far the most useful for hardware
- Big advances in SAT solvers
  - zChaff (Princeton)
  - can handle formulas with 100000 variables and millions of clauses!
- Boosted the idea of Bounded Model Checking (BMC)
  - NuSMV, and other more contemporary model checkers
- The SLAM tool from Microsoft Research (Ball and Rajamani)

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Verification (cont'd)

- Static Driver Verifier: big breakthrough
  - model checker that validates device derivers against formal spec.
  - Key ideas: predicate abstraction, algorithms for pushdown automata, BDDs for boolean programs.
- Light-weight static analysis
  - pointer chasing, data dependency analysis, ...
  - Codesurfer by GrammaTech
- Testing tools: PEX by Microsoft, DART, etc.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.

# Correct by Construction

Program Synthesis produces a program that satisfies a specification.

- Specifications: logical or examples.
- Algorithms for performing the synthesis.
- Formal models: to define state space of viable candidates.

Advantages:

- End user programming: for those who do not know programming (Excel Flashfill)
- Menial Programming Tasks: saving precious programmer time
- Removing Human Error.