Automated Verification for Functional and Relational Properties of Voting Rules

Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber, Mattias Ulbrich | July 26, 2016

KARLSRUHE INSTITUTE OF TECHNOLOGY – INSTITUTE OF THEORETICAL INFORMATICS

Motivation: An Example

Exemplary election for candidates A, B, and C, and nine voters

Ballot Profile

Voter	Ballot
1	A
2	A
3	A
4	A
5	B
6	B
7	B
8	C
9	C

Motivation: An Example

Exemplary election for candidates A, B, and C, and nine voters

Ballot Profile

Voter	Ballot
1	A
2	A
3	A
4	A
5	B
6	B
7	B
8	C
9	C

What should be the election outcome?

Motivation: An Example

Exemplary election for candidates A, B, and C, and nine voters

Ballot Profile

Voter	Ballot
1	A
2	A
3	A
4	A
5	B , C
6	B , C
7	B , C
8	C
9	C

What should be the election outcome?

Motivation: An Example

Exemplary election for candidates A, B, and C, and nine voters

Ballot Profile

Voter	Ballot
1	A > B > C
2	A > B > C
3	A > B > C
4	A > B > C
5	B > C > A
6	B > C > A
7	B > C > A
8	C > B > A
9	C > B > A

What should be the election outcome?

Motivation: An Example

Exemplary election for candidates A, B, and C, and nine voters

Ballot Profile

Voter	Ballot
1	A > **B** > C
2	A > **B** > C
3	A > **B** > C
4	A > **B** > C
5	**B** > C > A
6	**B** > C > A
7	**B** > C > A
8	C > **B** > A
9	C > **B** > A

What should be the election outcome?
Candidate B?

Motivation: An Example

Exemplary election for candidates A, B, C, D, and E, and nine voters

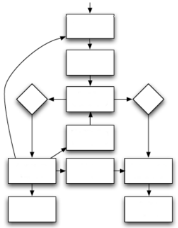
Ballot Profile

Voter	Ballot
1	A > **B** > **D** > **E** > C
2	A > **E** > **D** > **B** > C
3	A > **B** > **E** > **D** > C
4	A > **D** > **B** > **E** > C
5	**B** > **E** > **D** > C > A
6	**E** > **D** > **B** > C > A
7	**B** > **D** > **E** > C > A
8	C > **E** > **D** > **B** > A
9	C > **E** > **B** > **D** > A

What should be the election outcome?
Candidate B?
What if B is actually a coalition of the three candidates B, D, and E?

Motivation: The General Idea

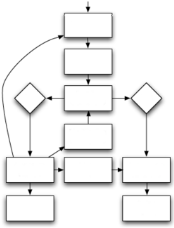
Voting Rule *V*



Motivation: The General Idea

Voting Rule *V*

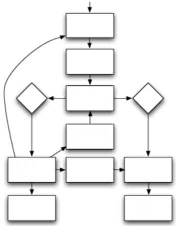
Ballot Profile *B*



Motivation: The General Idea

Voting Rule *V*

Ballot Profile *B*

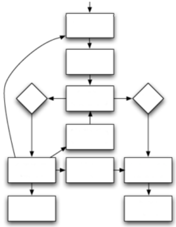


Outcome
V(B)

Motivation: The General Idea

Voting Rule V

Ballot Profile B



Outcome
$V(B)$

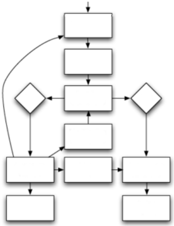
Axiomatic Property P

$$\forall x, y. \exists z \ldots$$

Motivation: The General Idea

Voting Rule *V*

Ballot Profile *B*



Outcome
V(B)

Axiomatic Property *P*

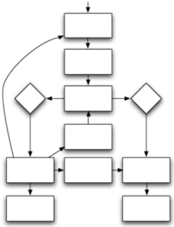
$\forall x, y. \exists z \ldots$

Does *V* satisfy *P*?

Motivation: The General Idea

Voting Rule *V*

Ballot Profile *B*



Outcome

V(B)

Axiomatic Property *P*

$\forall x, y.\exists z \ldots$

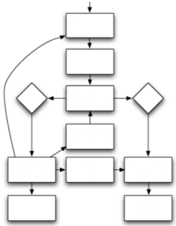
Does *V* satisfy *P*?

- Tedious, non-trivial and error-prone
- Especially for multiple properties
- Can this be automated?

Motivation: The General Idea

Voting Rule *V*

Ballot Profile B



Outcome
$V(B)$

Axiomatic Property *P*

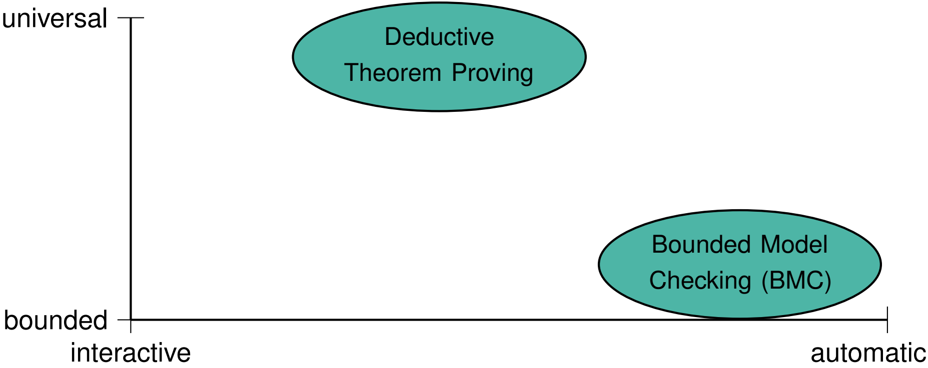
$\forall x, y . \exists z \ldots$

Does *V* satisfy *P*?

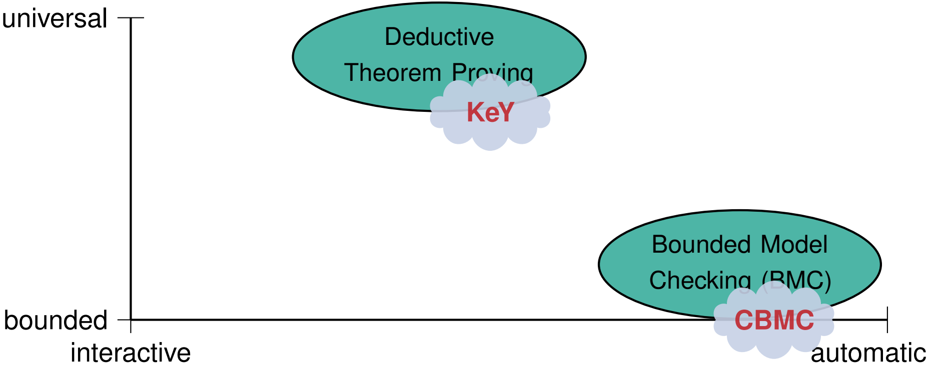
- Tedious, non-trivial and error-prone
- Especially for multiple properties
- Can this be automated?

Computer-aided verification
for **trustworthy** voting rules!

Used Verification Techniques

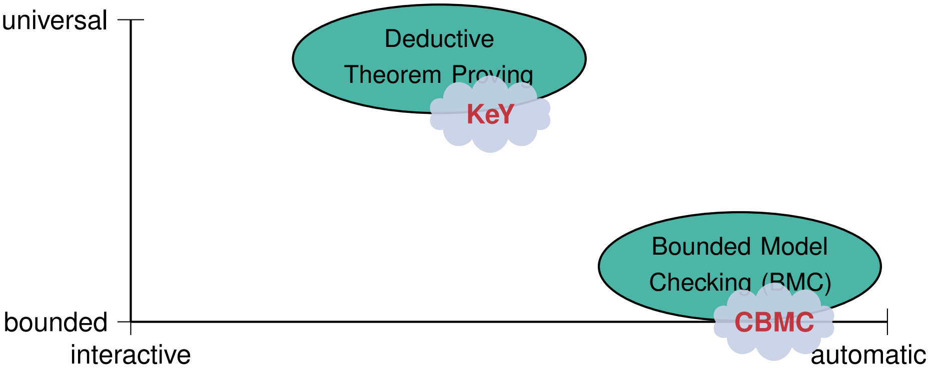


Used Verification Techniques



- *Established* verification techniques

Used Verification Techniques



- *Established* verification techniques
- *Expressive* languages for imperative algorithms (C / Java) and properties ($FOL_{\mathbb{N}}$)

Functional and Relational Properties

Functional Properties (intra-profile (Fishburn 1973))

- Consider individual election evaluations (one profile with outcome)
- Examples: **majority criterion**, Condorcet criterion

Relational Properties (inter-profile (Fishburn 1973))

- Consider multiple election evaluations (two profiles with outcomes)
- Examples: **anonymity property**, monotonicity property

Functional and Relational Properties

Functional Properties (intra-profile (Fishburn 1973))

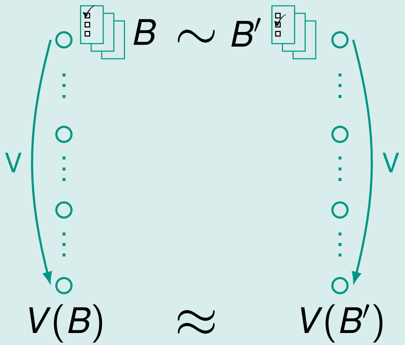
- Consider individual election evaluations (one profile with outcome)
- Examples: **majority criterion**, Condorcet criterion

Relational Properties (inter-profile (Fishburn 1973))

- Consider multiple election evaluations (two profiles with outcomes)
- Examples: **anonymity property**, monotonicity property

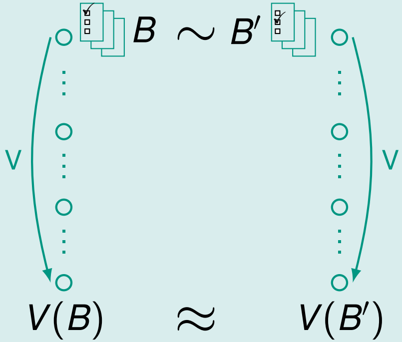
Verification of Relational Properties

Separate Evaluations



Verification of Relational Properties

Separate Evaluations

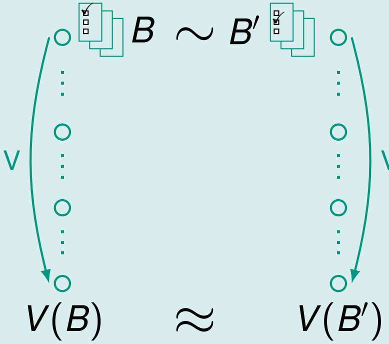


Example

$$\max_c \sum_{i=0}^{N} B_{i,c} = \max_c \sum_{i=0}^{N} B'_{i,c}$$

Verification of Relational Properties

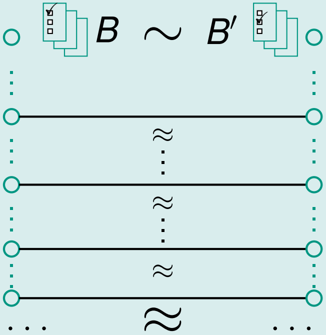
Separate Evaluations



Example

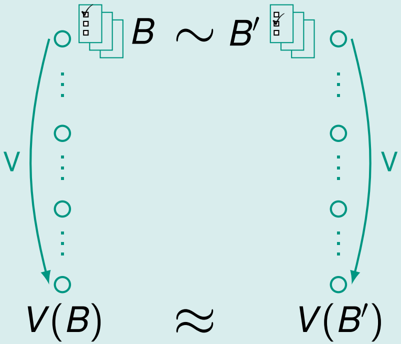
$$\max_c \sum_{i=0}^{N} B_{i,c} = \max_c \sum_{i=0}^{N} B'_{i,c}$$

Coupling Evaluations



Verification of Relational Properties

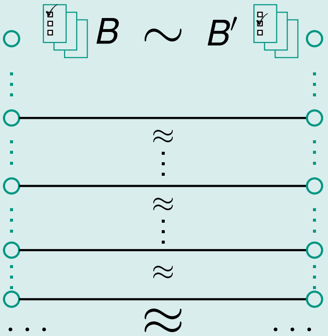
Separate Evaluations



Example

$\max_c \sum_{i=0}^{N} B_{i,c} = \max_c \sum_{i=0}^{N} {B'}_{i,c}$

Coupling Evaluations



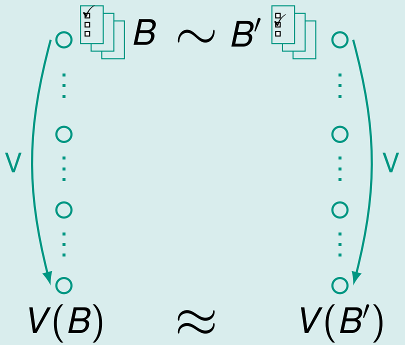
Example

result1 = *result2*

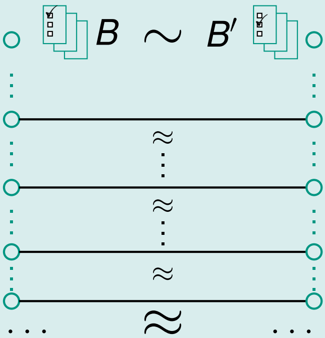
Verification of Relational Properties

Karlsruhe Institute of Technology

Separate Evaluations



Coupling Evaluations



Relational Verification

- Often enables short and concise specifications (only **differences**)
- Eases verification effort

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N ∗ V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N∗V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @         votes1[v] == votes2[k + v ∗ N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

Example: JML method contract for homogeneity

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N * V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N*V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @       votes1[v] == votes2[k + v * N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated N times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N ∗ V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N∗V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @       votes1[v] == votes2[k + v ∗ N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated N times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N * V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N*V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @        votes1[v] == votes2[k + v * N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

`res1` and `res2`: arrays for counting the candidates' votes

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N * V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N*V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @       votes1[v] == votes2[k + v * N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

`result1` and `result2`: fields storing the elected candidates

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N ∗ V ;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N∗V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @       votes1[v] == votes2[k + v ∗ N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

Wellformedness conditions

Specification with Coupling Evaluations

Example: Homogeneity for plurality rule

V: Each voter chooses one candidate, candidate with most votes wins

P: Outcome only depends on **proportion** of each ballot type, i.e., if every ballot is replicated *N* times, the outcome is indifferent

```
/*@ requires votes1.length == V ∧ votes2.length == N * V;
  @ requires (∀ int a; 0 ≤ a < V; 0 ≤ votes1[a] < C);
  @ requires (∀ int a; 0 ≤ a < N*V; 0 ≤ votes2[a] < C);
  @ requires (∀ int v,k; 0 ≤ v < V ∧ 0 ≤ k < N;
  @       votes1[v] == votes2[k + v * N]);
  @ assignable res1, res2, result1, result2;
  @ ensures result1 == result2;
  @*/ void voting(int[] votes1, int[] votes2);
```

Precondition for homogeneity

Coupling Loop Invariants: Homogeneity

Example: Summing up individual votes into arrays

```
/*@ loop_invariant 0 ≤ i1 ≤ V ∧ i1 * N == i2
  @ ∧ (∀ int c; 0 ≤ c < C; res2[c] == N*res1[c]);
  @ assignable res1[*], res2[*];
  @ decreases V − i1;
  @*/
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{     if (i1 < V) res1[votes1[i1++]]++;
      while (i2 < i1 * N) res2[votes2[i2++]]++;
}
```

Coupling Loop Invariants: Homogeneity

First evaluation: One single run

```
/*@ loop_invariant 0 ≤ i1 ≤ V ∧ i1 ∗ N == i2
  @ ∧ (∀ int c; 0 ≤ c < C; res2[c] == N∗res1[c]);
  @ assignable res1[*], res2[*];
  @ decreases V − i1;
  @*/
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V ∗ N;)
{     if (i1 < V) res1[votes1[i1++]]++;
      while (i2 < i1 ∗ N) res2[votes2[i2++]]++;
}
```

Coupling Loop Invariants: Homogeneity

Second evaluation: One run replicated N times

```
/*@ loop_invariant 0 ≤ i1 ≤ V ∧ i1 * N == i2
  @ ∧ (∀ int c; 0 ≤ c < C; res2[c] == N*res1[c]);
  @ assignable res1[*], res2[*];
  @ decreases V − i1;
  @*/
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{     if (i1 < V) res1[votes1[i1++]]++;
      while (i2 < i1 * N) res2[votes2[i2++]]++;
}
```

Coupling Loop Invariants: Homogeneity

Coupling invariant: Relationship between both arrays

```
/*@ loop_invariant 0 ≤ i1 ≤ V ∧ i1 ∗ N == i2
  @ ∧ (∀ int c; 0 ≤ c < C; res2[c] == N∗res1[c]);
  @ assignable res1[*], res2[*];
  @ decreases V − i1;
  @*/
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V ∗ N;)
{     if (i1 < V) res1[votes1[i1++]]++;
      while (i2 < i1 ∗ N) res2[votes2[i2++]]++;
}
```

Coupling Loop Invariants: Homogeneity

Coupling evaluations: Loop invariant for replicated run

```
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{   if (i1 < V) res1[votes1[i1++]]++;
    /*@ loop_invariant 0 < i1 ≤ V ∧ i2 ≤ votes2.length
      @ ∧ (i1 − 1) * N ≤ i2 ≤ i1 * N
      @ ∧ (∀ int c; 0 ≤ c < C ∧ c ≠ votes1[i1 − 1];
      @              res2[c] == N*res1[c])
      @ ∧ (i2 < i1 * N ==> votes1[i1 − 1] == votes2[i2])
      @ ∧ res2[votes1[i1 − 1]]
      @    == res1[votes1[i1 − 1]] * N + (i2 − i1 * N);
      @ assignable res2[*];
      @ decreases (i1 + 1) * N − i2;
      @*/
    while (i2 < i1 * N) res2[votes2[i2++]]++; }
```

Coupling Loop Invariants: Homogeneity

Range restrictions

```
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{   if (i1 < V) res1[votes1[i1++]]++;
    /*@ loop_invariant 0 < i1 ≤ V ∧ i2 ≤ votes2.length
      @ ∧ (i1 − 1) * N ≤ i2 ≤ i1 * N
      @ ∧ (∀ int c; 0 ≤ c < C ∧ c ≠ votes1[i1 − 1];
      @                res2[c] == N*res1[c])
      @ ∧ (i2 < i1 * N ==> votes1[i1 − 1] == votes2[i2])
      @ ∧ res2[votes1[i1 − 1]]
      @    == res1[votes1[i1 − 1]] * N + (i2 − i1 * N);
      @ assignable res2[*];
      @ decreases (i1 + 1) * N − i2;
      @*/
    while (i2 < i1 * N) res2[votes2[i2++]]++; }
```

Coupling Loop Invariants: Homogeneity

Framing invariant for results from previous rounds

```
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{    if (i1 < V) res1[votes1[i1++]]++;
     /*@ loop_invariant 0 < i1 ≤ V ∧ i2 ≤ votes2.length
       @ ∧ (i1 − 1) * N ≤ i2 ≤ i1 * N
       @ ∧ (∀ int c; 0 ≤ c < C ∧ c ≠ votes1[i1 − 1];
       @                res2[c] == N*res1[c])
       @ ∧ (i2 < i1 * N ==> votes1[i1 − 1] == votes2[i2])
       @ ∧ res2[votes1[i1 − 1]]
       @    == res1[votes1[i1 − 1]] * N + (i2 − i1 * N);
       @ assignable res2[*];
       @ decreases (i1 + 1) * N − i2;
       @*/
     while (i2 < i1 * N) res2[votes2[i2++]]++; }
```

Coupling Loop Invariants: Homogeneity

Relationship for current round, not strictly necessary

```
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{   if (i1 < V) res1[votes1[i1++]]++;
    /*@ loop_invariant 0 < i1 ≤ V ∧ i2 ≤ votes2.length
      @ ∧ (i1 − 1) * N ≤ i2 ≤ i1 * N
      @ ∧ (∀ int c; 0 ≤ c < C ∧ c ≠ votes1[i1 − 1];
      @              res2[c] == N*res1[c])
      @ ∧ (i2 < i1 * N ==> votes1[i1 − 1] == votes2[i2])
      @ ∧ res2[votes1[i1 − 1]]
      @    == res1[votes1[i1 − 1]] * N + (i2 − i1 * N);
      @ assignable res2[*];
      @ decreases (i1 + 1) * N − i2;
      @*/
    while (i2 < i1 * N) res2[votes2[i2++]]++; }
```

Coupling Loop Invariants: Homogeneity

Current result array relationship, $i1 * N$ is distance from "compartment" start

```
for (int i1 = 0, int i2 = 0; i1 < V || i2 < V * N;)
{   if (i1 < V) res1[votes1[i1++]]++;
    /*@ loop_invariant 0 < i1 ≤ V ∧ i2 ≤ votes2.length
      @ ∧ (i1 − 1) * N ≤ i2 ≤ i1 * N
      @ ∧ (∀ int c; 0 ≤ c < C ∧ c ≠ votes1[i1 − 1];
      @               res2[c] == N*res1[c])
      @ ∧ (i2 < i1 * N ==> votes1[i1 − 1] == votes2[i2])
      @ ∧ res2[votes1[i1 − 1]]
      @    == res1[votes1[i1 − 1]] * N + (i2 − i1 * N);
      @ assignable res2[*];
      @ decreases (i1 + 1) * N − i2;
      @*/
    while (i2 < i1 * N) res2[votes2[i2++]]++; }
```

Verification with Coupling Evaluations

Example: Verification using KeY (including required lines of specification)

	Plurality V.	Approval V.	Range V.	Borda Count
Anonymity	33	43	44	44
Neutrality	42	56	57	57
Monotonicity	46	47	48	52
Participation	28	50	51	50
Homogeneity	53	70	71	71

- Case study for multiple rules and properties
- Breaks down verification effort (roughly) to functional verification
- Verification using separate evaluations often not feasible
- Concise specifications also useful for bounded model checking
  $\rightarrow$ Guides solver to achieve higher bounds

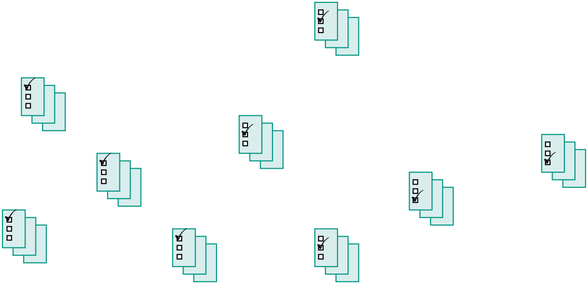
Verification with Coupling Evaluations

Example: Verification using KeY (including required lines of specification)

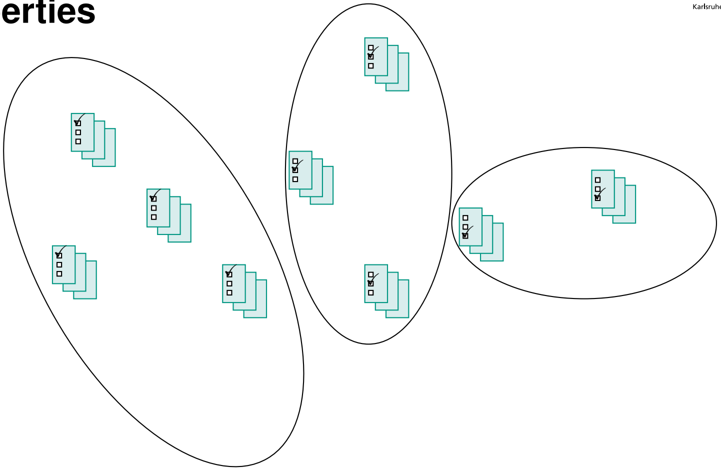
	Plurality V.	Approval V.	Range V.	Borda Count
Anonymity	33	43	44	44
Neutrality	42	56	57	57
Monotonicity	46	47	48	52
Participation	28	50	51	50
Homogeneity	53	70	71	71

- Case study for multiple rules and properties
- Breaks down verification effort (roughly) to functional verification
- Verification using separate evaluations often not feasible
- Concise specifications also useful for bounded model checking
  $\rightarrow$ Guides solver to achieve higher bounds

Exploiting Symmetries on Functional Properties

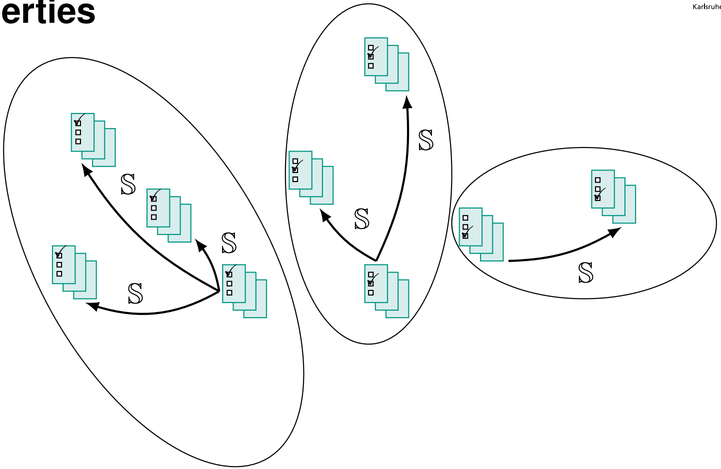


Exploiting Symmetries on Functional Properties



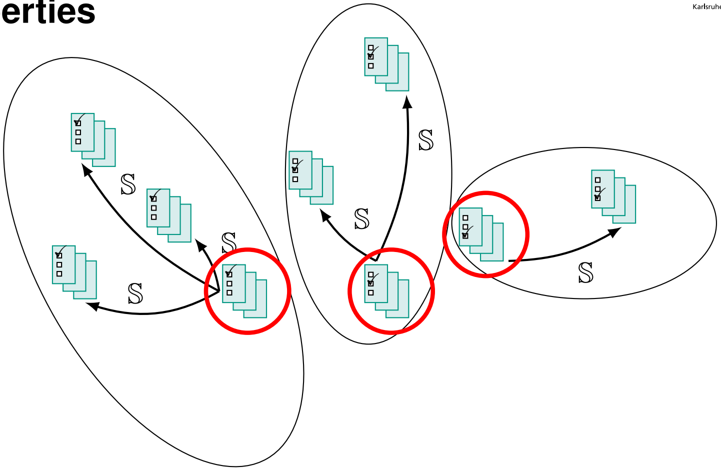
Symmetric profiles (for a symmetry property $\mathbb{S}$)

Exploiting Symmetries on Functional Properties



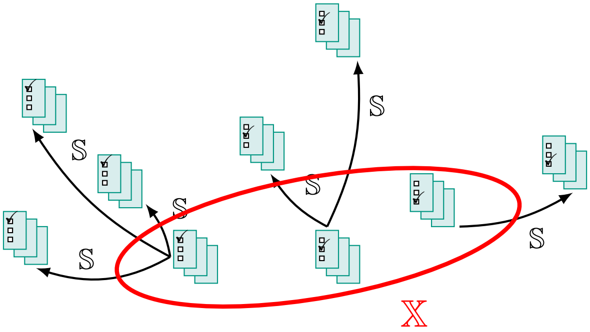
Symmetric profiles (for a symmetry property $\mathbb{S}$)
are reachable via symmetry (profile-) operations.

Exploiting Symmetries on Functional Properties



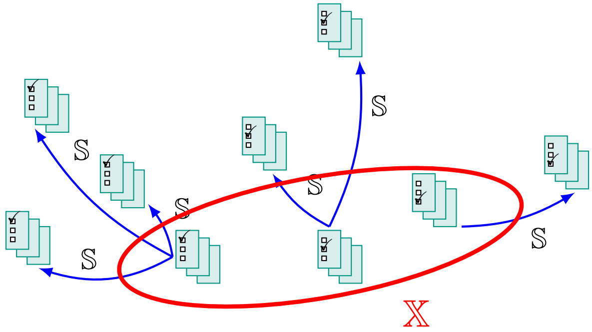
Symmetric profiles (for a symmetry property $\mathbb{S}$)
are reachable via symmetry (profile-) operations from minimal elements.

Exploiting Symmetries on Functional Properties



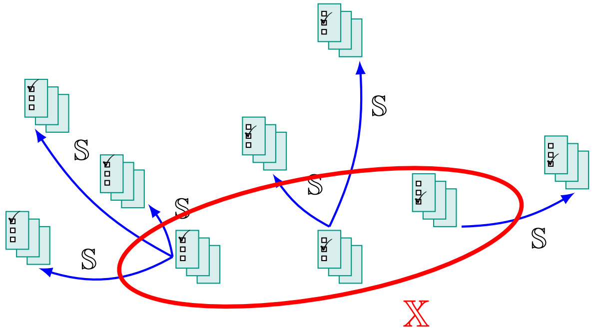
These minimal elements form a set $\mathbb{X}$,

Exploiting Symmetries on Functional Properties



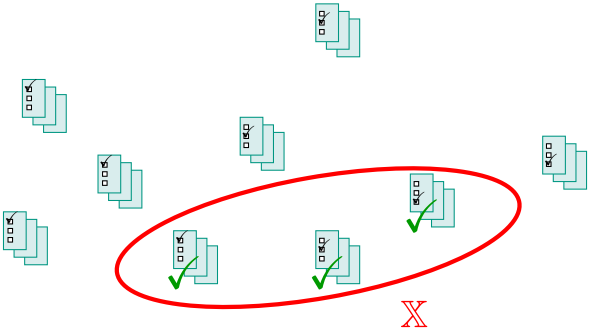
These minimal elements form a set $\mathbb{X}$, via which *all possible profiles* are reachable.

Exploiting Symmetries on Functional Properties



Hence, if $\mathbb{S}$-operations preserve the desired property $\boldsymbol{P}$,

Exploiting Symmetries on Functional Properties



Hence, if $\mathbb{S}$-operations preserve the desired property ***P***,
verifying ***P*** only for elements in $\mathbb{X}$ is sufficient.

Verification of Functional Properties

- Verification Task: Does voting rule $\boldsymbol{V}$ satisfy property $\boldsymbol{P}$?
- Conjecture: $\boldsymbol{V}$ satisfies symmetry property $\mathbb{S}$.

Verification of Functional Properties

- Verification Task: Does voting rule $\boldsymbol{V}$ satisfy property $\boldsymbol{P}$?
- Conjecture: $\boldsymbol{V}$ satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for $\boldsymbol{V}$ using relational techniques
2. Verify $\boldsymbol{V}$ satisfies property $\boldsymbol{P}$ only for subset $\mathbb{X}$
3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property $\boldsymbol{P}$

Verification of Functional Properties

- Verification Task: Does voting rule ***V*** satisfy property ***P***?
- Conjecture: ***V*** satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for ***V*** using relational techniques
2. Verify ***V*** satisfies property ***P*** only for subset $\mathbb{X}$

program verification

3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property ***P***

Verification of Functional Properties

- Verification Task: Does voting rule ***V*** satisfy property ***P*** ?
- Conjecture: ***V*** satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for ***V*** using relational techniques
2. Verify ***V*** satisfies property ***P*** only for subset $\mathbb{X}$

program verification

3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property ***P***

Verification of Functional Properties

- Verification Task: Does voting rule $\boldsymbol{V}$ satisfy property $\boldsymbol{P}$?
- Conjecture: $\boldsymbol{V}$ satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for $\boldsymbol{V}$ using relational techniques
2. Verify $\boldsymbol{V}$ satisfies property $\boldsymbol{P}$ only for subset $\mathbb{X}$

} program verification

3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property $\boldsymbol{P}$

} independent of V

Verification of Functional Properties

- Verification Task: Does voting rule $\boldsymbol{V}$ satisfy property $\boldsymbol{P}$?
- Conjecture: $\boldsymbol{V}$ satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for $\boldsymbol{V}$ using relational techniques
2. Verify $\boldsymbol{V}$ satisfies property $\boldsymbol{P}$ only for subset $\mathbb{X}$

} program verification

3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property $\boldsymbol{P}$

} independent of V

Example

$\boldsymbol{V}$: Plurality rule

$\boldsymbol{P}$: Majority criterion

$\mathbb{S}$: Anonymity property

$\mathbb{X}$: ?

Verification of Functional Properties

- Verification Task: Does voting rule $\boldsymbol{V}$ satisfy property $\boldsymbol{P}$?
- Conjecture: $\boldsymbol{V}$ satisfies symmetry property $\mathbb{S}$.

General Theorem for Verification

1. Verify $\mathbb{S}$ for $\boldsymbol{V}$ using relational techniques
2. Verify $\boldsymbol{V}$ satisfies property $\boldsymbol{P}$ only for subset $\mathbb{X}$

program verification

3. Prove that $\mathbb{X}$ spans *all possible profiles*
4. Prove that $\mathbb{S}$-operations preserve property $\boldsymbol{P}$

independent of V

Example

$\boldsymbol{V}$: Plurality rule

$\boldsymbol{P}$: Majority criterion

$\mathbb{S}$: Anonymity property

$\mathbb{X}$: All sorted (by chosen candidate) profiles

Exploiting Symmetries for Verification

How do we fix the set $\mathbb{X}$ for use in verification?

Exploiting Symmetries for Verification

How do we fix the set $\mathbb{X}$ for use in verification?

Answer: Use *symmetry-breaking predicates* (SBP).

Exploiting Symmetries for Verification

How do we fix the set $\mathbb{X}$ for use in verification?

Answer: Use *symmetry-breaking predicates* (SBP).

- Predicates which are only valid for elements in $\mathbb{X}$
- Means to reduce search space
- Used as precondition for input

Exploiting Symmetries for Verification

How do we fix the set $\mathbb{X}$ for use in verification?

Answer: Use *symmetry-breaking predicates* (SBP).

- Predicates which are only valid for elements in $\mathbb{X}$
- Means to reduce search space
- Used as precondition for input

Example for anonymity property and plurality rule

- Profiles denoted as $(b_1, \ldots, b_N)$ (N number of cast ballots)
- Each ballot denotes exactly one chosen candidate
- Predicate valid only for sorted ballot profiles:
  $\forall i \in \{2, \ldots, N\} : b_{i-1} \leq b_i$

Exploiting Symmetries for Verification

How do we fix the set $\mathbb{X}$ for use in verification?

Answer: Use *symmetry-breaking predicates* (SBP).

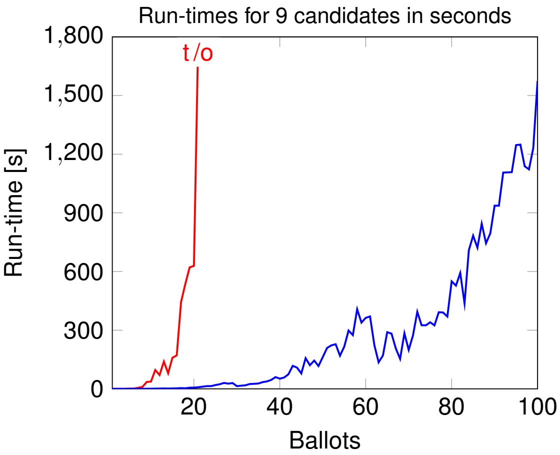
- Predicates which are only valid for elements in $\mathbb{X}$
- Means to reduce search space
- Used as precondition for input

Example for anonymity property and plurality rule

- Profiles denoted as $(b_1, \ldots, b_N)$ (N number of cast ballots)
- Each ballot denotes exactly one chosen candidate
- Predicate valid only for sorted ballot profiles:
  $\forall i \in \{2, \ldots, N\} : b_{i-1} \leq b_i$

Verification Using Symmetry Breaking

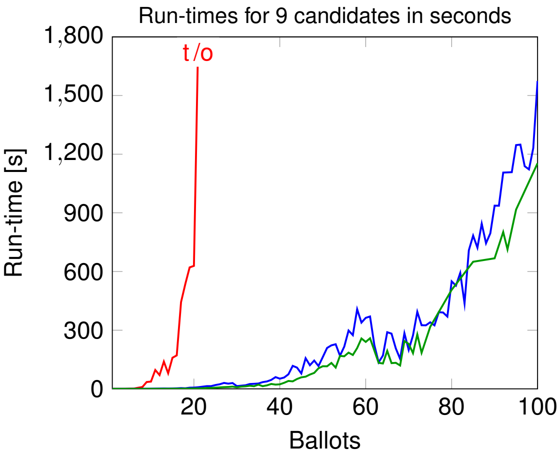
Example: Verification using bounded model checking (CBMC)



- Verified majority for plurality rule
- With and without SBP for anonymity
- Results: Significantly pushed the boundaries!
- Case study for multiple rules and properties
- Composition of symmetries: anonymity **plus** neutrality

Verification Using Symmetry Breaking

Example: Verification using bounded model checking (CBMC)



- Verified majority for plurality rule
- With and without SBP for anonymity
- Results: Significantly pushed the boundaries!
- Case study for multiple rules and properties
- Composition of symmetries: anonymity **plus** neutrality

Conclusion and Outlook

Karlsruhe Institute of Technology

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Conclusion and Outlook

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Conclusion and Outlook

Karlsruhe Institute of Technology

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Conclusion and Outlook

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Conclusion and Outlook

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Conclusion and Outlook

Results

- **General** approach for verification of axiomatic properties
- Coupling evaluations enable short and concise specifications
  $\Rightarrow$ Often critical point to make **verification feasible!**
- Exploiting (generalised) symmetries significantly pushes boundaries
- Feasibility demonstrated on a variety of well-known results

Future Work

- Generalisation of approach to further classes of properties
- Application on further and more complex examples

Questions and Answers

Thank you

for your attention!

Any questions?

Questions and Answers

Thank you

for your attention!

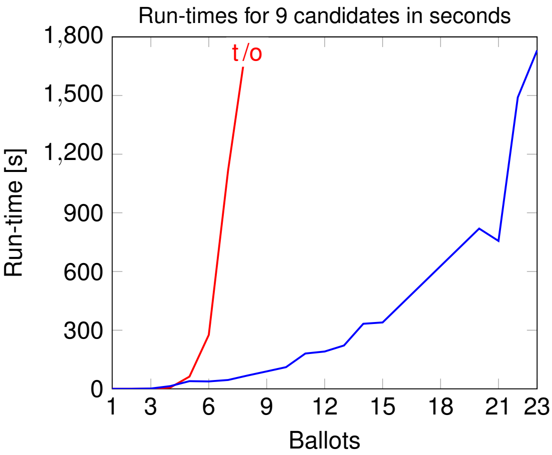
Any questions?

References

Peter C. Fishburn. *The Theory of Social Choice*. Princeton University Press, 1973 (cit. on pp. 18, 19).

Verification with Coupling Evaluations

Example: Verification using bounded model checking (CBMC)



- Verified anonymity for plurality rule
- Concise specifications useable for BMC ⇒ Guidance for SAT-solver
- Separate and coupling evaluations
- Results: Achieved higher bounds