OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

Databases
and
Software
Engineering

Technische
Universität
Braunschweig

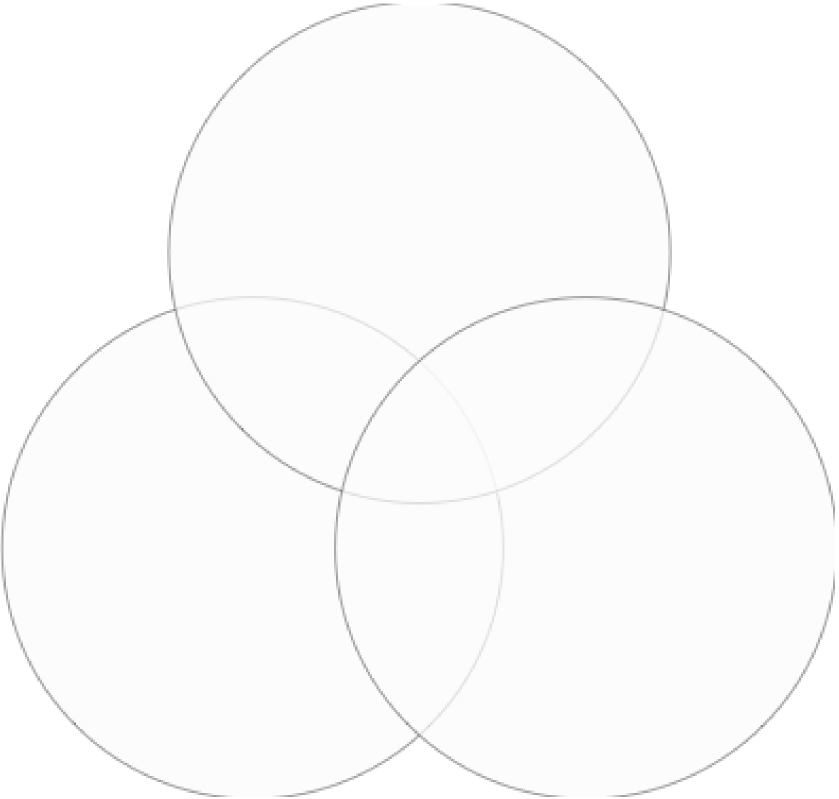
ISF

Proof Repositories for Correct-by-Construction Software Product Lines

Master Thesis | Elias Kuiter | January 8, 2021
Advisors: Ina Schaefer, Tabea Bordis, Tobias Runge, Gunter Saake

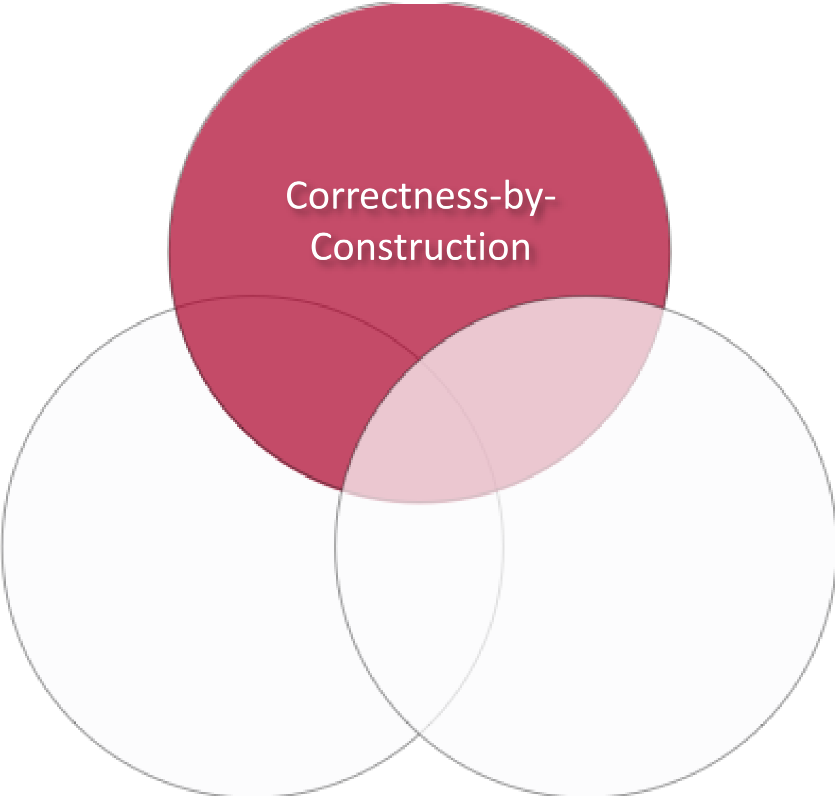
Introduction

Background & Problem



Introduction

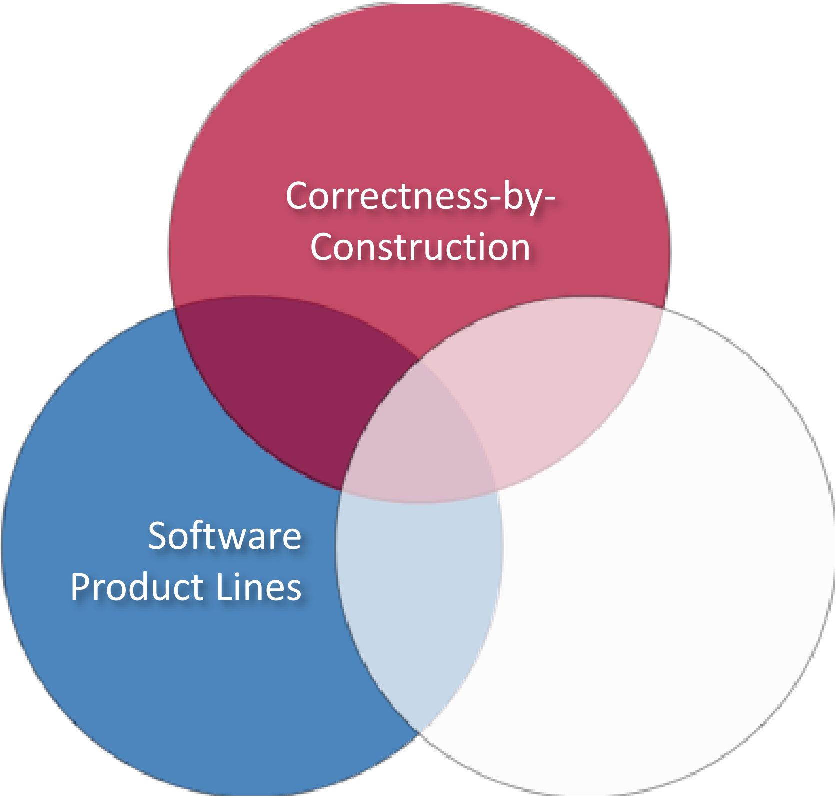
Background & Problem



Correctness-by-Construction

Introduction

Background & Problem

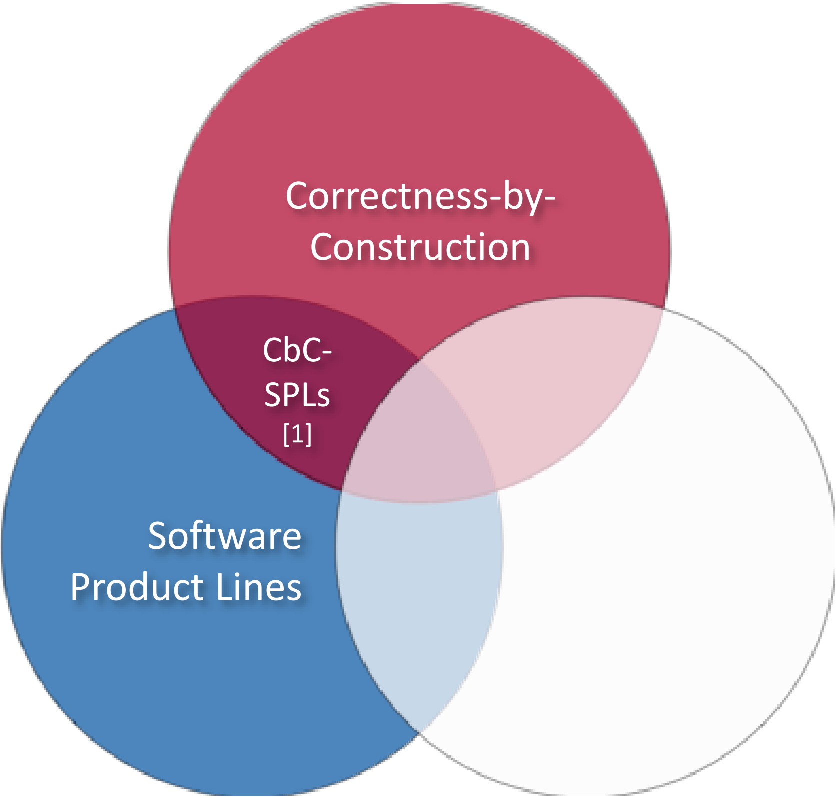


Correctness-by-Construction

Software Product Lines

Introduction

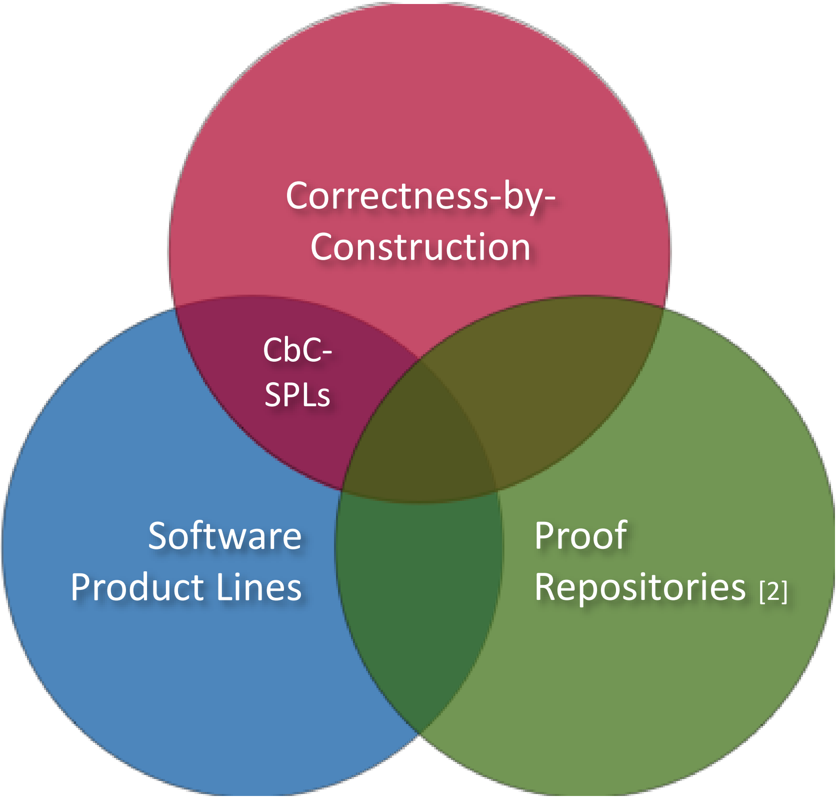
Background & Problem



[1] Tabea Bordis, Tobias Runge, and Ina Schaefer. 2020. Correctness-by-Construction for Feature-Oriented Software Product Lines. *GPCE Proc'*, ACM.

Introduction

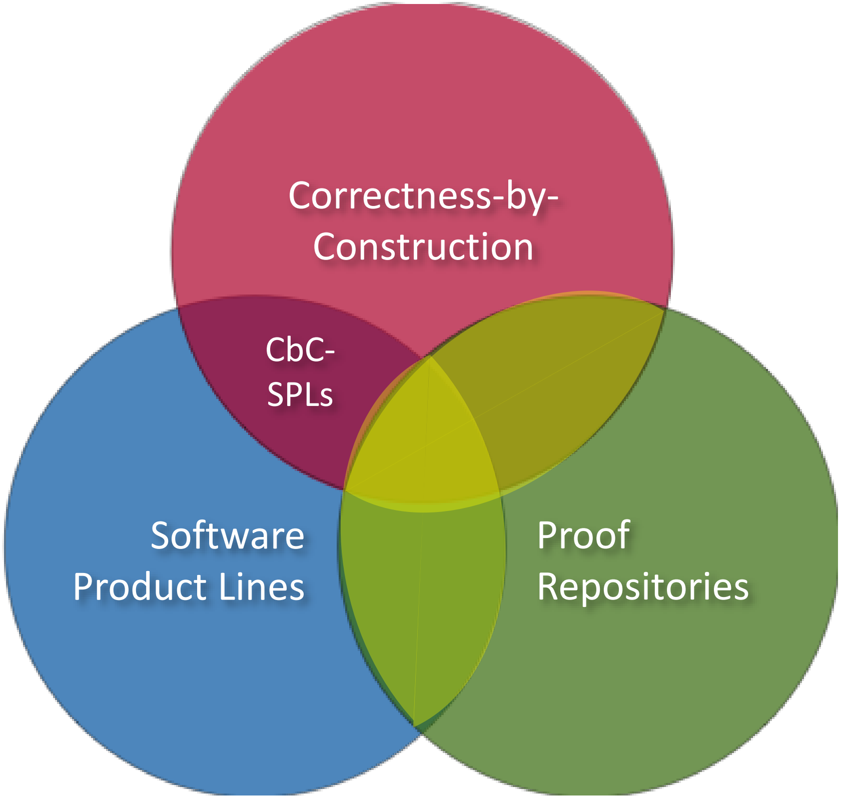
Background & Problem



[2] Richard Bubel et al. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. *Trans. on Found. for Mastering Change I*. Springer.

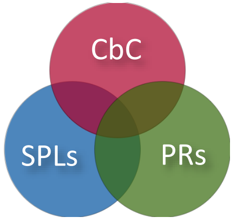
Introduction

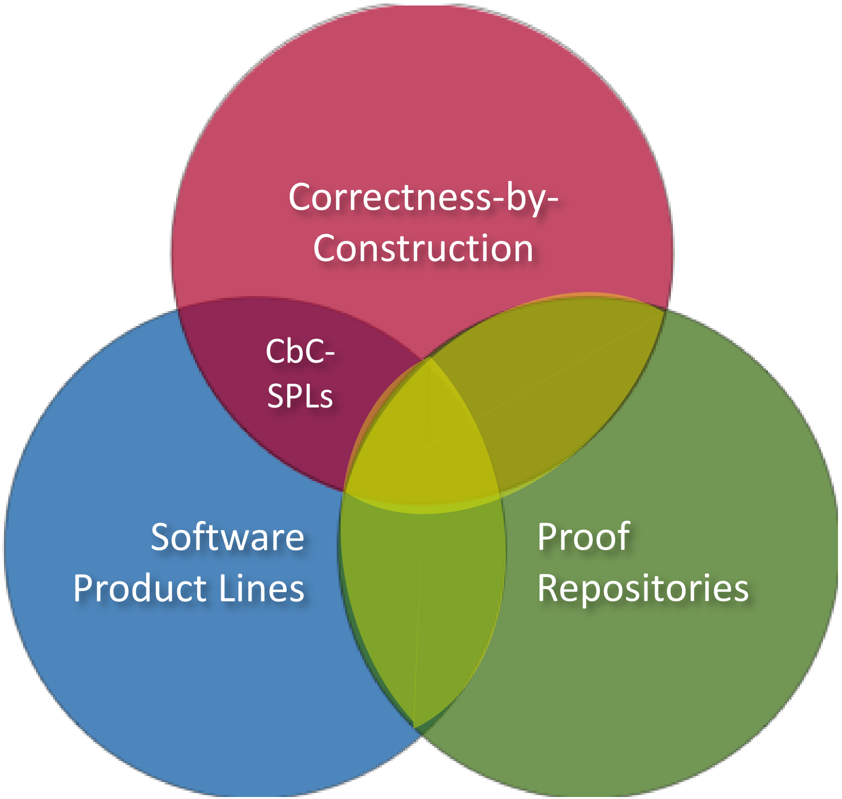
Goal & Contributions



Introduction

Goal & Contributions



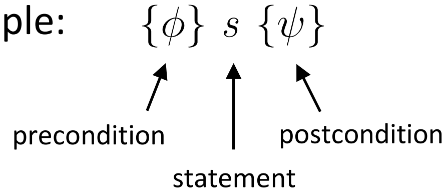


Correctness-by-Construction

CbC

- Programming methodology
- **Idea**: - Start with a (trivial) **correct program** and

  - Apply only correct transformations (***refinements***)
- Specification, code, and proof of correctness are constructed at the same time
- Follows the **design-by-contract** principle:



$$\{\phi\}\ s\ \{\psi\}$$

Correctness-by-Construction: Example

$\phi :=$ "A is an array"

$\psi :=$ "A' contains x and A"

① $\{\phi\} \ \text{\textvisiblespace}\ \{\psi\}$

Correctness-by-Construction: Example

CbC

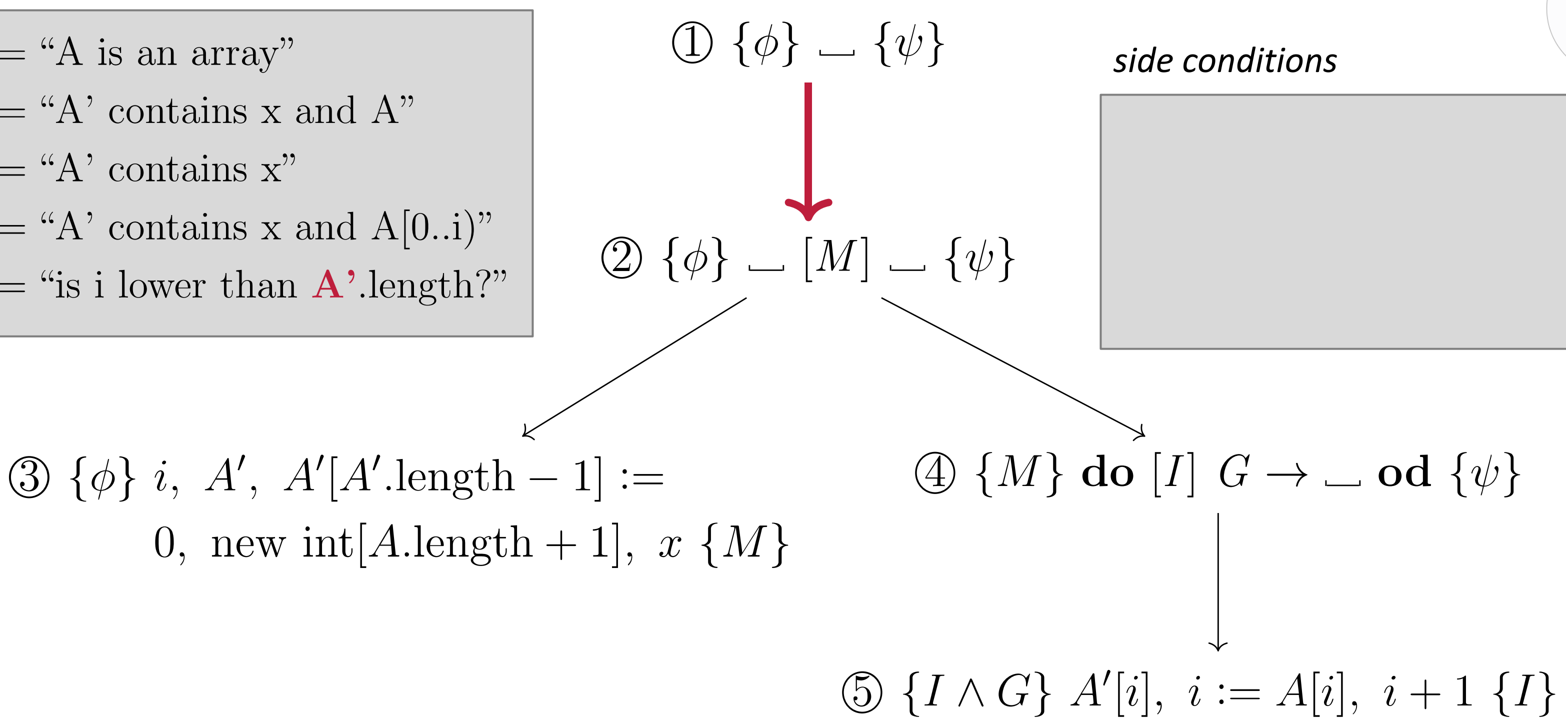
ϕ := “A is an array”

ψ := “A’ contains x and A”

M := “A’ contains x”

I := “A’ contains x and A[0..i)”

G := “is i lower than **A’**.length?”



① $\{\phi\} \textvisiblespace \{\psi\}$

② $\{\phi\} \textvisiblespace [M] \textvisiblespace \{\psi\}$

③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length} - 1] := 0,\ \text{new int}[A.\text{length} + 1],\ x\ \{M\}$

④ $\{M\}\ \textbf{do}\ [I]\ G \rightarrow \textvisiblespace\ \textbf{od}\ \{\psi\}$

⑤ $\{I \land G\}\ A'[i],\ i := A[i],\ i + 1\ \{I\}$

side conditions

Correctness-by-Construction: Example

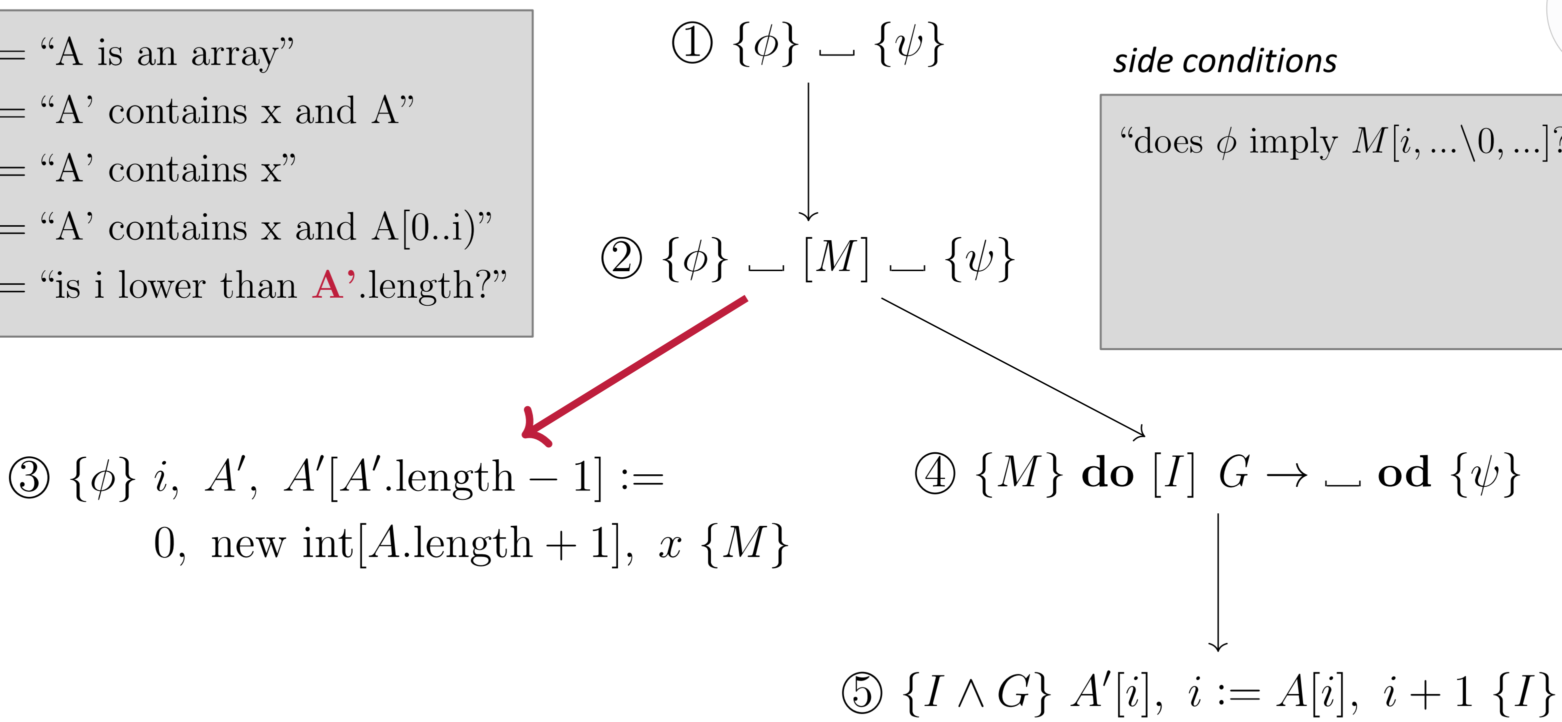
ϕ := “A is an array”

ψ := “A’ contains x and A”

M := “A’ contains x”

I := “A’ contains x and A[0..i)”

G := “is i lower than **A’**.length?”



① $\{\phi\}\ \_\ \{\psi\}$

② $\{\phi\}\ \_\ [M]\ \_\ \{\psi\}$

③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length}-1] := 0,\ \text{new int}[A.\text{length}+1],\ x\ \{M\}$

④ $\{M\}\ \textbf{do}\ [I]\ G \rightarrow \_\ \textbf{od}\ \{\psi\}$

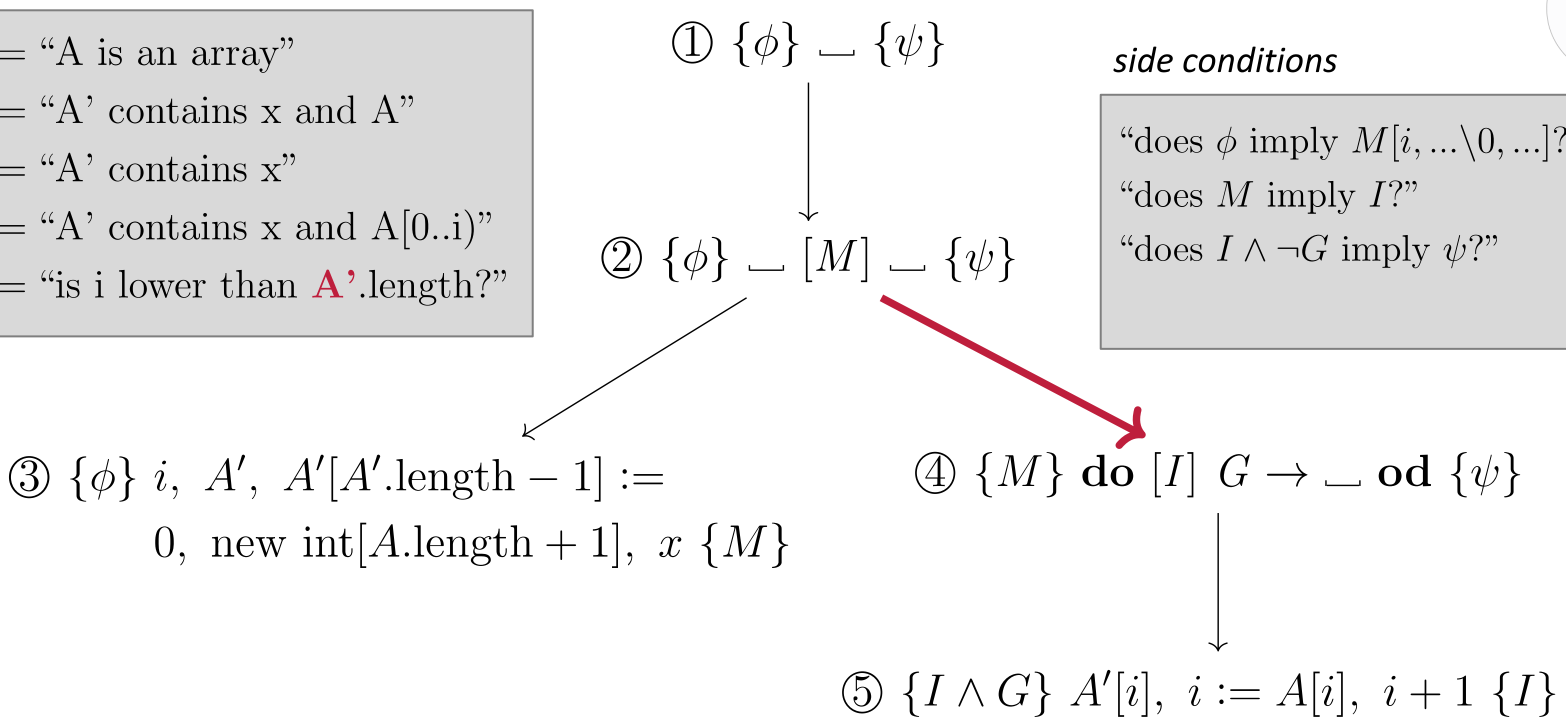
⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i+1\ \{I\}$

side conditions

“does ϕ imply $M[i,\ldots\backslash 0,\ldots]$?”

Correctness-by-Construction: Example

$\phi :=$ “A is an array”
$\psi :=$ “A’ contains x and A”
$M :=$ “A’ contains x”
$I :=$ “A’ contains x and A[0..i)”
$G :=$ “is i lower than **A’**.length?”



① $\{\phi\} \text{\textvisiblespace} \{\psi\}$

② $\{\phi\} \text{\textvisiblespace} [M] \text{\textvisiblespace} \{\psi\}$

③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length}-1] := 0,\ \text{new int}[A.\text{length}+1],\ x\ \{M\}$

④ $\{M\}\ \mathbf{do}\ [I]\ G \rightarrow \text{\textvisiblespace}\ \mathbf{od}\ \{\psi\}$

⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i+1\ \{I\}$

side conditions

“does ϕ imply $M[i, ...\backslash 0, ...]$?”
“does M imply I?”
“does $I \wedge \neg G$ imply ψ?”

CbC

Correctness-by-Construction: Example

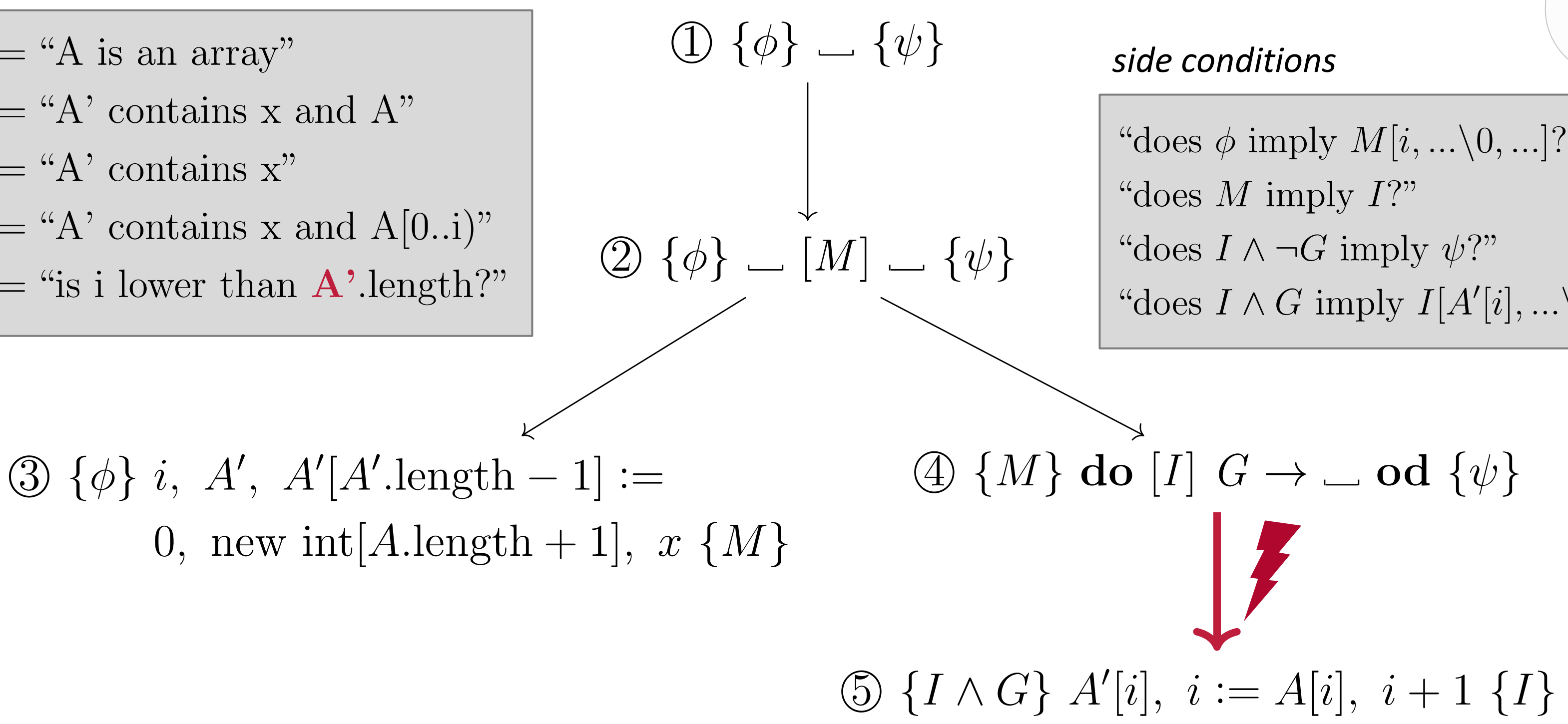
ϕ := "A is an array"

ψ := "A' contains x and A"

M := "A' contains x"

I := "A' contains x and A[0..i)"

G := "is i lower than **A'**.length?"



① $\{\phi\} \textvisiblespace \{\psi\}$

② $\{\phi\} \textvisiblespace [M] \textvisiblespace \{\psi\}$

③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length} - 1] := 0,\ \text{new int}[A.\text{length} + 1],\ x\ \{M\}$

④ $\{M\}\ \textbf{do}\ [I]\ G \rightarrow \textvisiblespace\ \textbf{od}\ \{\psi\}$

⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i + 1\ \{I\}$

side conditions

"does ϕ imply $M[i, ...\backslash 0, ...]$?"

"does M imply I?"

"does $I \wedge \neg G$ imply ψ?"

"does $I \wedge G$ imply $I[A'[i], ...\backslash A[i], ...]$?"

Software Product Lines

Feature-Oriented Programming

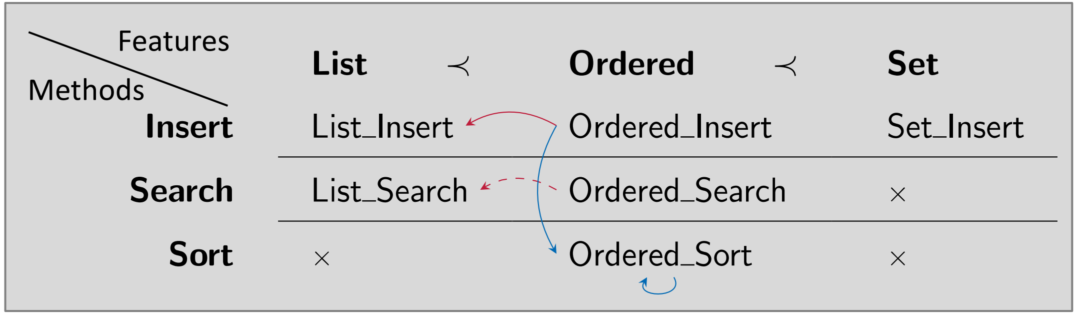
- **FOP**: Composition-based approach for building software product lines
- **Idea**: - Create total order of features
  - Feature modules add new methods or call parent methods with *`original()`*
- Variability is encoded in non-deterministic *`original()`* calls

Software Product Lines

Feature-Oriented Programming

- **FOP**: Composition-based approach for building software product lines
- **Idea**: - Create total order of features

  - Feature modules add new methods or call parent methods with *`original()`*
- Variability is encoded in non-deterministic *`original()`* calls



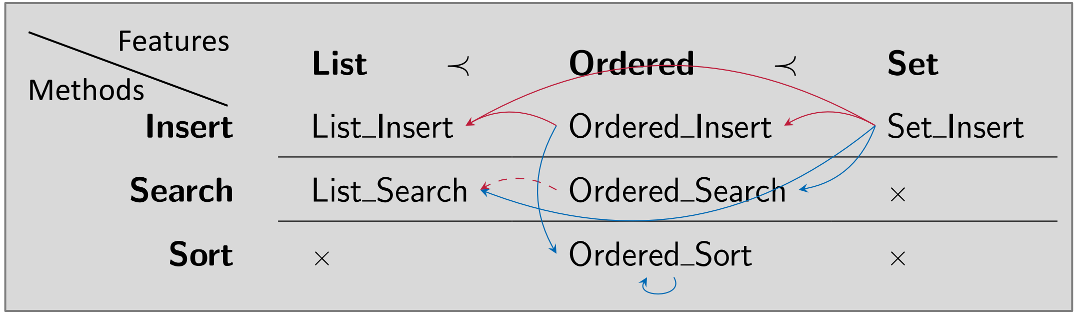
Features / Methods	**List**	≺	**Ordered**	≺	**Set**
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

Software Product Lines

Feature-Oriented Programming

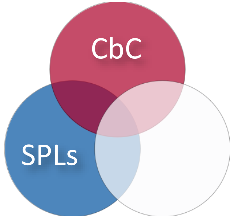
- **FOP**: Composition-based approach for building software product lines
- **Idea**: - Create total order of features

  - Feature modules add new methods or call parent methods with *`original()`*
- Variability is encoded in non-deterministic *`original()`* calls



Features / Methods	**List**	≺	**Ordered**	≺	**Set**
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

Software Product Lines



Correct-by-Construction SPLs [1]

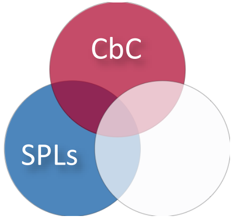
- An FOP-inspired extension of traditional CbC that ...
  - introduces a new refinement rule for **method calls**:

$$\begin{tikzcd}\{\phi\}\ \text{\textvisiblespace}\ \{\psi\} \arrow[d, "\text{(assuming that } \Phi \text{ and } \Psi \text{ fit with the contract of } m\text{)}"] \\ \{\phi\}\ b := m(a_1, \ldots, a_n)\ \{\psi\}\end{tikzcd}$$

  - introduces **original calls** (*m* = `original`) to invoke the parent method
  - allows `original` to occur in contracts (**contract composition**):

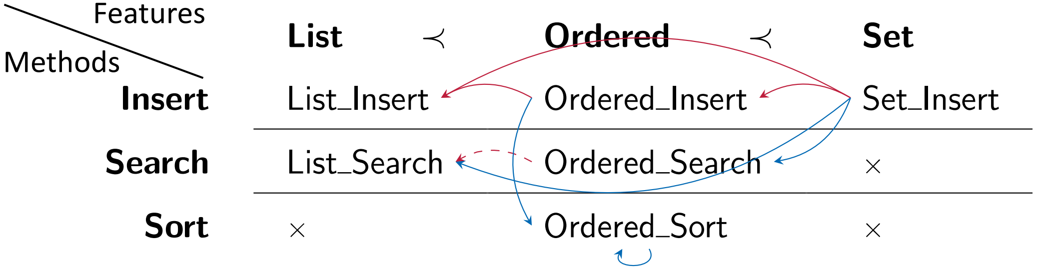
$$\{original_{pre}(A, x) \wedge isSorted(A)\}\ \text{\textvisiblespace}\ \{original_{post}(A, x)\}$$

[1] Tabea Bordis, Tobias Runge, and Ina Schaefer. 2020. Correctness-by-Construction for Feature-Oriented Software Product Lines. *GPCE Proc'*, ACM.



Software Product Lines

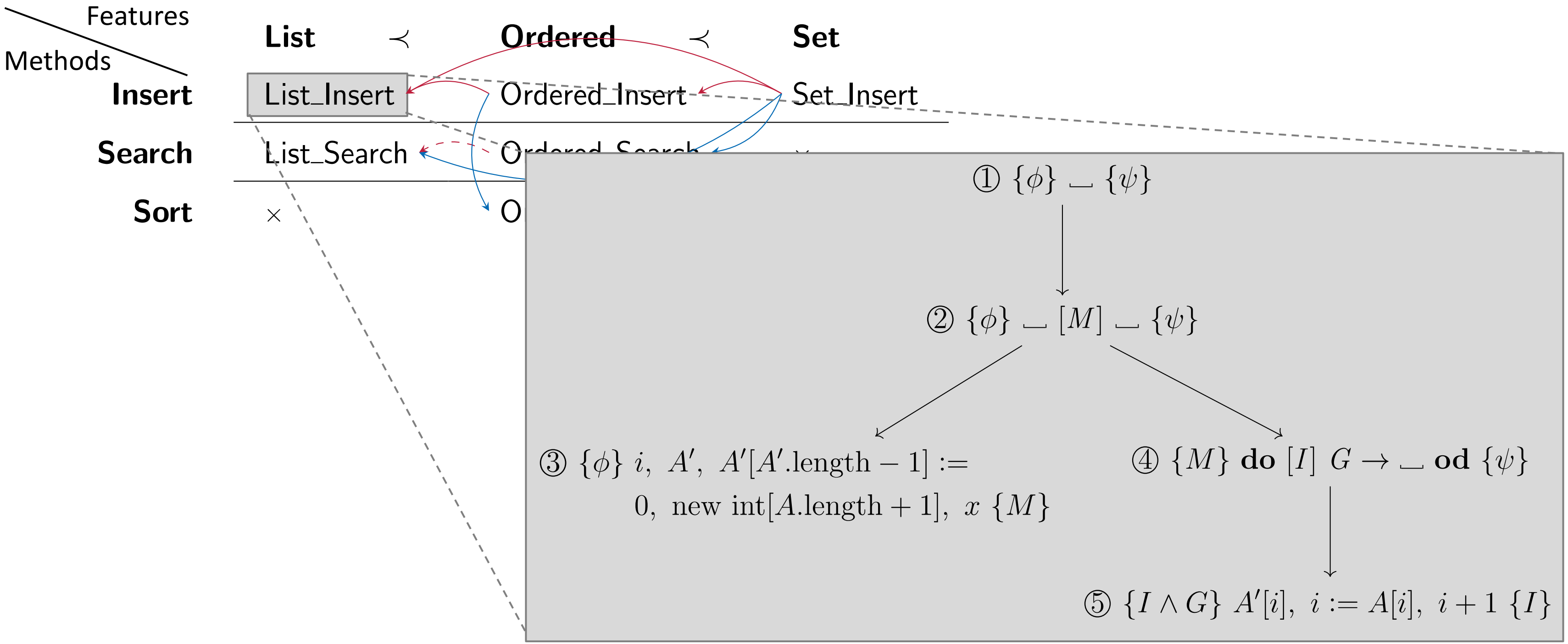
Correct-by-Construction SPLs: Example



Features / Methods	**List** ≺	**Ordered** ≺	**Set**
Insert	List_Insert	Ordered_Insert	Set_Insert
Search	List_Search	Ordered_Search	×
Sort	×	Ordered_Sort	×

Software Product Lines

Correct-by-Construction SPLs: Example



Features / Methods	**List**	≺	**Ordered**	≺	**Set**
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		O		

① $\{\phi\}\ \_\ \{\psi\}$

② $\{\phi\}\ \_\ [M]\ \_\ \{\psi\}$

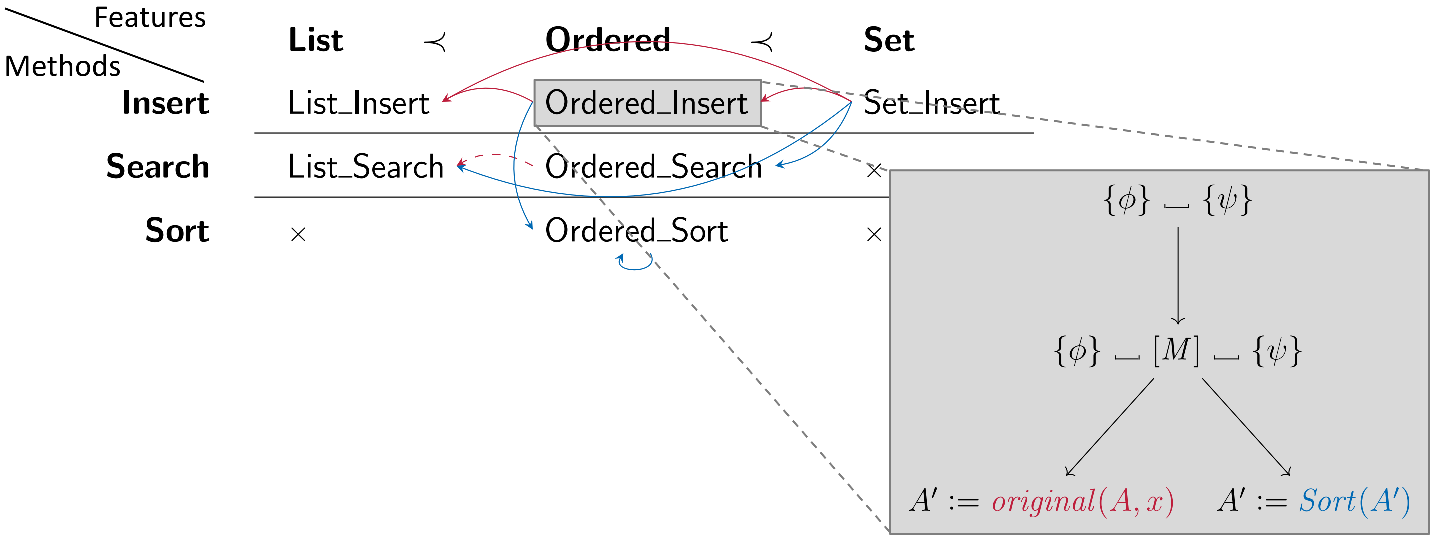
③ $\{\phi\}\ i,\ A',\ A'[A'.\mathrm{length}-1] := 0,\ \mathrm{new\ int}[A.\mathrm{length}+1],\ x\ \{M\}$

④ $\{M\}\ \mathbf{do}\ [I]\ G \rightarrow \_\ \mathbf{od}\ \{\psi\}$

⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i+1\ \{I\}$

Software Product Lines

Correct-by-Construction SPLs: Example



Features Methods	List	≺	Ordered	≺	Set
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

$$\{\phi\} \textvisiblespace \{\psi\}$$

$$\{\phi\} \textvisiblespace [M] \textvisiblespace \{\psi\}$$

$$A' := original(A, x) \qquad A' := Sort(A')$$

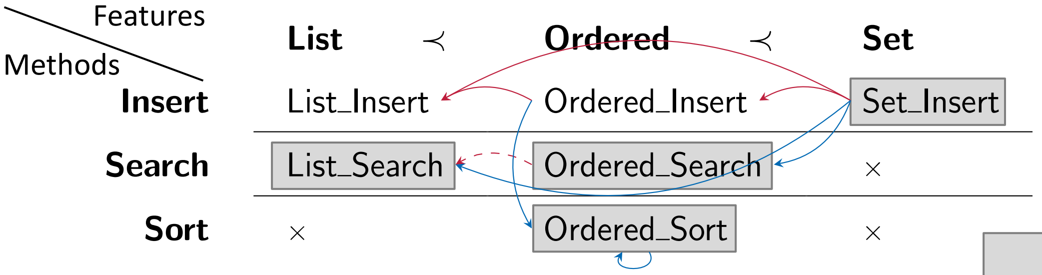
$$\phi := original_{pre}(A, x) \land isSorted(A)$$

$$\psi := original_{post}(A, x) \land isSorted(A')$$

$$M := original_{post}(A, x)$$

Software Product Lines

Correct-by-Construction SPLs: Example

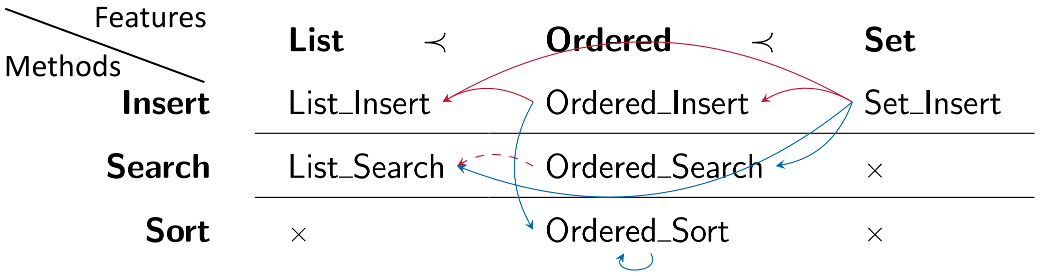


Features / Methods	List	≺	Ordered	≺	Set
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

. . .

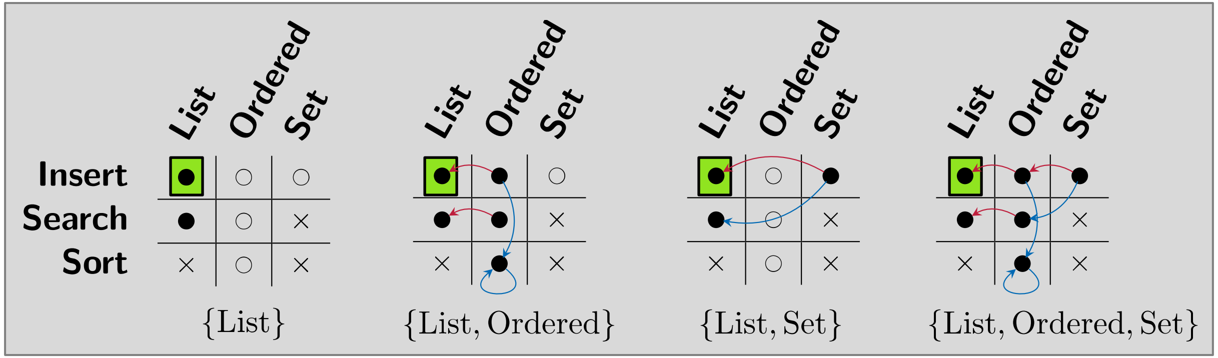
Software Product Lines

Correct-by-Construction SPLs: Example



Features / Methods	List	≺	Ordered	≺	Set
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

product derivation



	List	Ordered	Set
Insert	●	○	○
Search	●	○	×
Sort	×	○	×

{List}

	List	Ordered	Set
Insert	●	●	○
Search	●	●	×
Sort	×	●	×

{List, Ordered}

	List	Ordered	Set
Insert	●	○	●
Search	●	○	×
Sort	×	○	×

{List, Set}

	List	Ordered	Set
Insert	●	●	●
Search	●	●	×
Sort	×	●	×

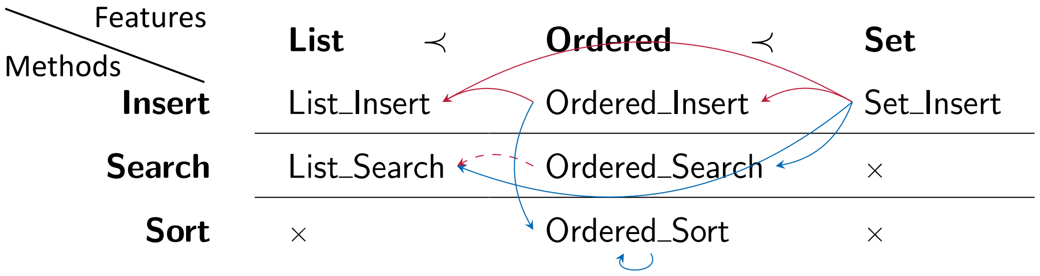
{List, Ordered, Set}

Potential for reuse

1. avoid obvious re-verification

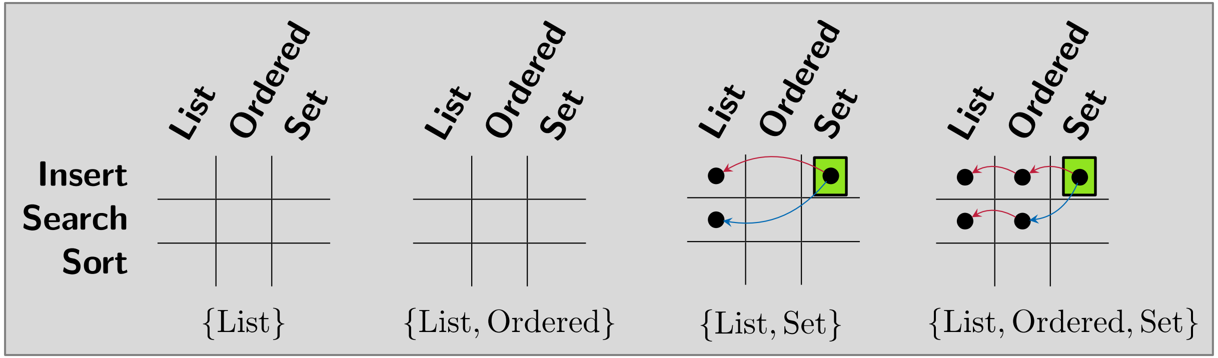
Software Product Lines

Correct-by-Construction SPLs: Example



Methods \ Features	List	≺	Ordered	≺	Set
Insert	List_Insert		Ordered_Insert		Set_Insert
Search	List_Search		Ordered_Search		×
Sort	×		Ordered_Sort		×

product derivation



Potential for reuse

1. avoid obvious re-verification
2. leverage overlaps

Proof Repositories [2]

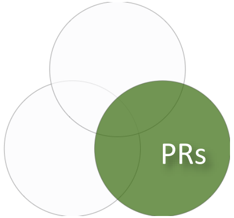
- A mathematical framework for **proof reuse** in compositional verification
- **Proof repository**: „Database“ (🛢) of conducted proofs (at method-level)
- Intended to improve performance for verification-in-the-large
- **Idea**: - Separate *method calls* from *called methods* with **abstract contracts** (▮)

```
void f() { ... g(); ... }

void g() { ... }
```

---

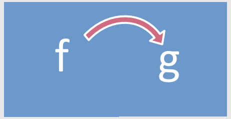
[2] Richard Bubel et al. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. *Trans. on Found. for Mastering Change I*. Springer.

Proof Repositories [2]

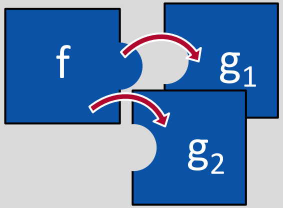


- A mathematical framework for **proof reuse** in compositional verification
- **Proof repository**: „Database“ (⛁) of conducted proofs (at method-level)
- Intended to improve performance for verification-in-the-large
- **Idea**: - Separate *method calls* from *called methods* with **abstract contracts** (🧩)

```
void f() { ... g(); ... }

void g() { ... }
```



```
void f() { ... ␣(); ... }

void g_variant1() { ... }

void g_variant2() { ... }
```



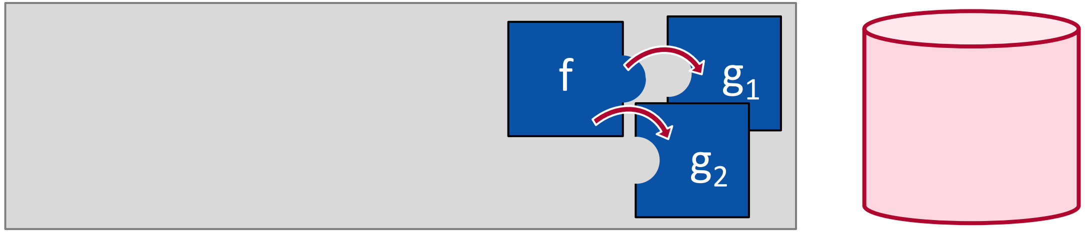
---

[2] Richard Bubel et al. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. *Trans. on Found. for Mastering Change I*. Springer.

Proof Repositories [2]

- A mathematical framework for **proof reuse** in compositional verification
- **Proof repository**: „Database“ (🛢) of conducted proofs (at method-level)
- Intended to improve performance for verification-in-the-large
- **Idea**: - Separate *method calls* from *called methods* with **abstract contracts** (🧩)

  - Then, conduct and reuse *partial proofs*



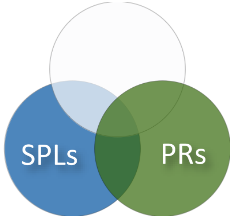
instead of:

$$f \longrightarrow g_1$$

$$f \longrightarrow g_2$$

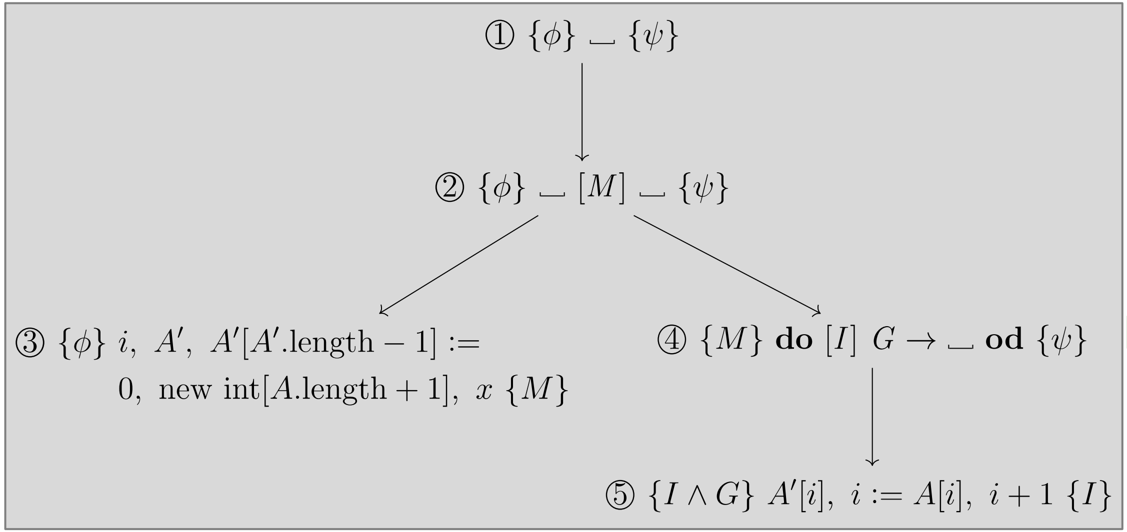
[2] Richard Bubel et al. 2016. Proof Repositories for Compositional Verification of Evolving Software Systems. *Trans. on Found. for Mastering Change I*. Springer.

Reducing Methods to Proof Repositories



Coarse-Grained Transformation

- **Goal**: Express CbC trees with the proof repository framework
- **First solution**: Translate CbC trees into whole methods



① $\{\phi\}\ \_\ \{\psi\}$

② $\{\phi\}\ \_\ [M]\ \_\ \{\psi\}$

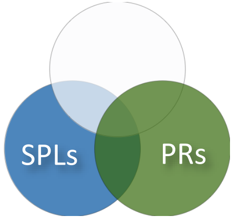
③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length}-1] := 0,\ \text{new int}[A.\text{length}+1],\ x\ \{M\}$

④ $\{M\}\ \textbf{do}\ [I]\ G \rightarrow \_\ \textbf{od}\ \{\psi\}$

⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i+1\ \{I\}$

```
class List_Insert {
  int[] A, A';
  int x, i;

  /*@ requires φ; ensures ψ; @*/
  int[] main(int[] A, int x) {
    i = 0, ...;
    /*@ loop_invariant I; @*/
    while (i < A.length)
      A'[i] = A[i], ...;
    return A';
  }
}
```

Reducing Methods to Proof Repositories

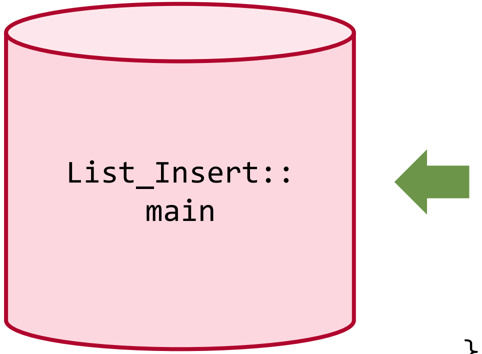


Coarse-Grained Transformation

- **Goal**: Express CbC trees with the proof repository framework
- **First solution**: Translate CbC trees into whole methods

Pro
- simple
- suitable for classical FOP

Con
- requires finished methods
- no reuse for evolution
- hampers debugging

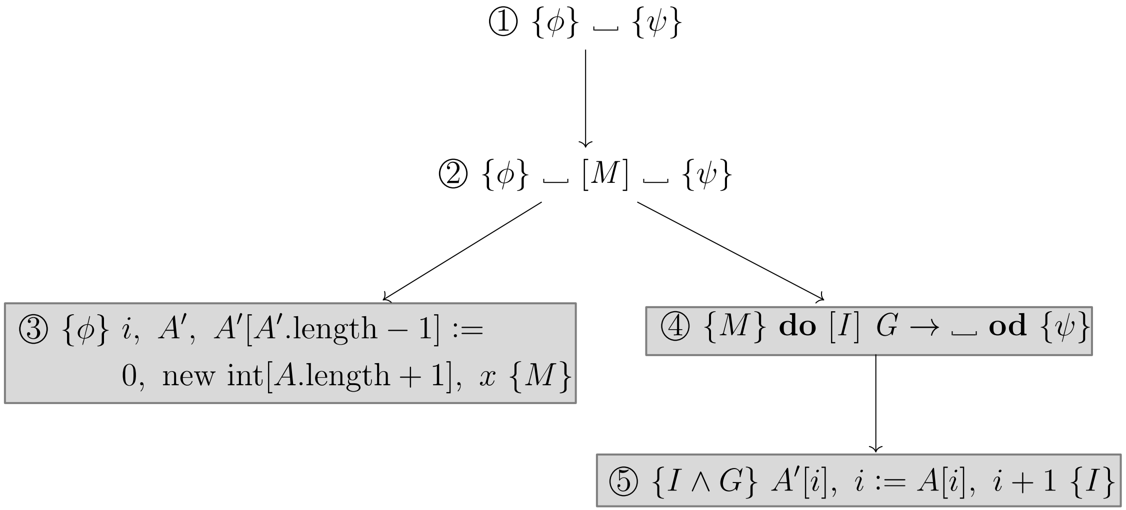


```
class List_Insert {
  int[] A, A';
  int x, i;

  /*@ requires ϕ; ensures ψ; @*/
  int[] main(int[] A, int x) {
    i = 0, ...;
    /*@ loop_invariant I; @*/
    while (i < A.length)
      A'[i] = A[i], ...;
    return A';
  }
}
```

Reducing Methods to Proof Repositories

Fine-Grained Transformation

- **Goal**: Express CbC trees with the proof repository framework
- **Second solution**: Translate into many small methods



① $\{\phi\} \textvisiblespace \{\psi\}$

② $\{\phi\} \textvisiblespace [M] \textvisiblespace \{\psi\}$

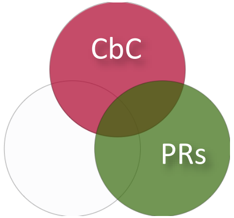
③ $\{\phi\}\ i,\ A',\ A'[A'.\text{length}-1] := 0,\ \text{new int}[A.\text{length}+1],\ x\ \{M\}$

④ $\{M\}\ \textbf{do}\ [I]\ G \to \textvisiblespace\ \textbf{od}\ \{\psi\}$

⑤ $\{I \wedge G\}\ A'[i],\ i := A[i],\ i+1\ \{I\}$

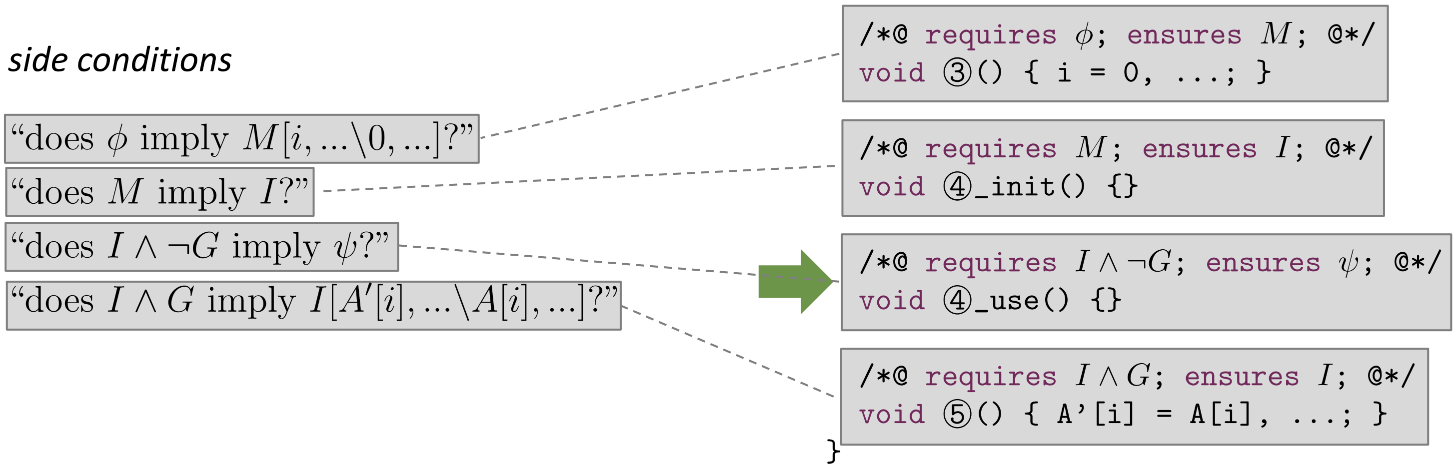
```
class List_Insert {
  int[] A, A';
  int x, i;

  /*@ requires ϕ; ensures M; @*/
  void ③() { i = 0, ...; }

  /*@ requires M; ensures I; @*/
  void ④_init() {}

  /*@ requires I ∧ ¬G; ensures ψ; @*/
  void ④_use() {}

  /*@ requires I ∧ G; ensures I; @*/
  void ⑤() { A'[i] = A[i], ...; }
}
```

Reducing Methods to Proof Repositories



Fine-Grained Transformation

- **Goal**: Express CbC trees with the proof repository framework
- **Second solution**: Translate into many small methods



side conditions

“does ϕ imply $M[i, ...\backslash 0, ...]$?”

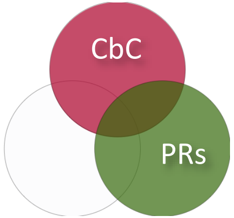
“does M imply I?”

“does $I \wedge \neg G$ imply ψ?”

“does $I \wedge G$ imply $I[A'[i], ...\backslash A[i], ...]$?”

```
class List_Insert {
  int[] A, A’;
  int x, i;

  /*@ requires φ; ensures M; @*/
  void ③() { i = 0, ...; }

  /*@ requires M; ensures I; @*/
  void ④_init() {}

  /*@ requires I ∧ ¬G; ensures ψ; @*/
  void ④_use() {}

  /*@ requires I ∧ G; ensures I; @*/
  void ⑤() { A’[i] = A[i], ...; }
}
```

Reducing Methods to Proof Repositories

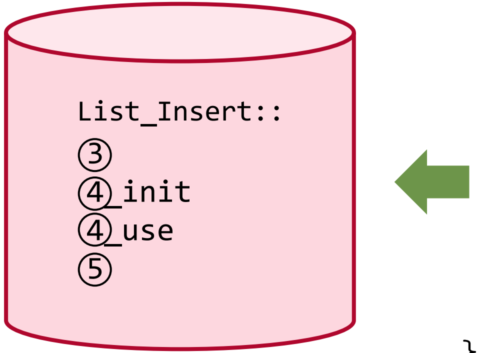


Fine-Grained Transformation

- **Goal**: Express CbC trees with the proof repository framework
- **Second solution**: Translate into many small methods

Pro
- suitable for CbC
- allows evolution/debugging
- correctness-preserving

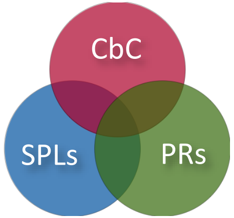
Con
- implementation effort



List_Insert::
③
④_init
④_use
⑤

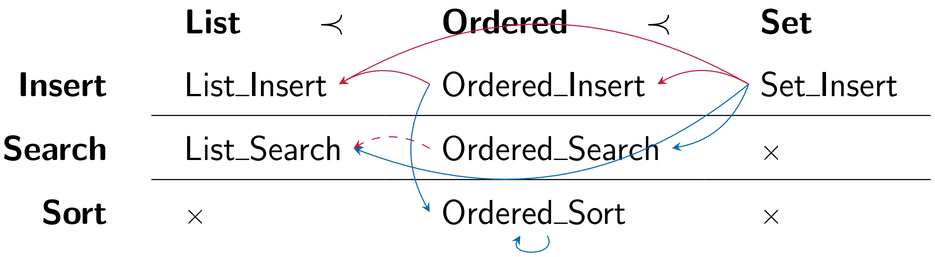
```
class List_Insert {
  int[] A, A’;
  int x, i;

  /*@ requires ϕ; ensures M; @*/
  void ③() { i = 0, ...; }

  /*@ requires M; ensures I; @*/
  void ④_init() {}

  /*@ requires I ∧ ¬G; ensures ψ; @*/
  void ④_use() {}

  /*@ requires I ∧ G; ensures I; @*/
  void ⑤() { A’[i] = A[i], ...; }
}
```

Reducing CbC-SPLs to Proof Repositories



CbC-SPL Transformation

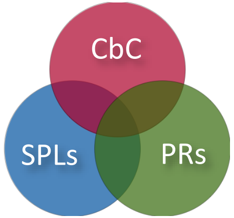
- **Goal**: Express entire CbC-SPLs with the proof repository framework
- **Solution**: Bind calls to actual methods using *abstract contracts*



	List $\prec$	Ordered $\prec$	Set
Insert	List_Insert	Ordered_Insert	Set_Insert
Search	List_Search	Ordered_Search	×
Sort	×	Ordered_Sort	×

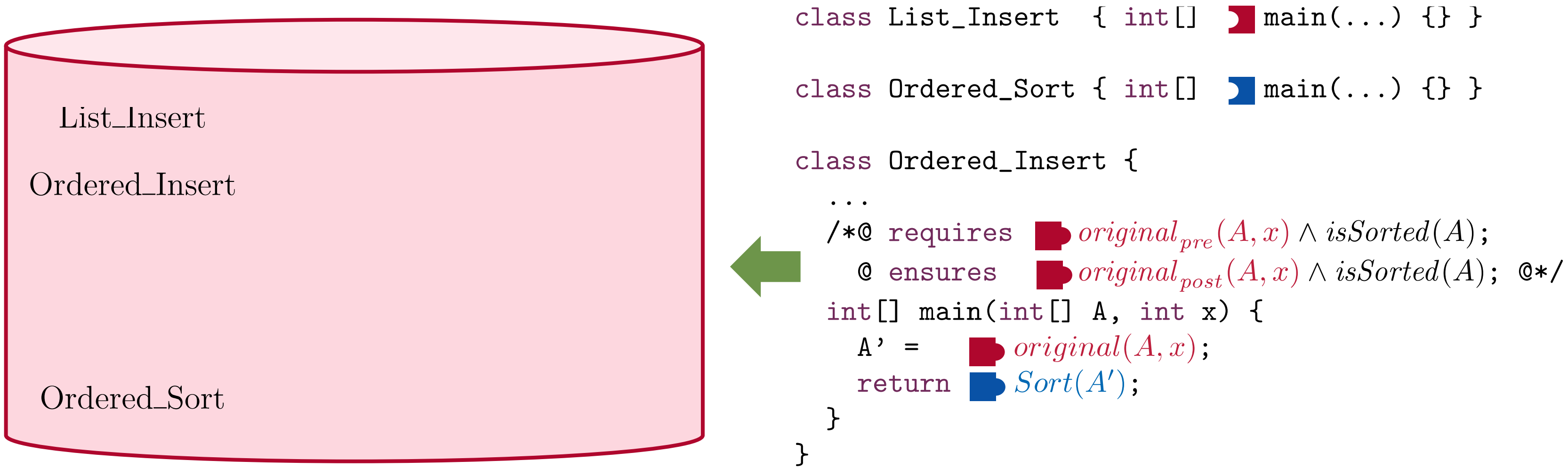
```
class List_Insert   { int[] 🟥 main(...) {} }

class Ordered_Sort { int[] 🟦 main(...) {} }

class Ordered_Insert {
  ...
  /*@ requires 🟥 $original_{pre}(A, x) \wedge isSorted(A)$;
    @ ensures  🟥 $original_{post}(A, x) \wedge isSorted(A)$; @*/
  int[] main(int[] A, int x) {
    A' =    🟥 $original(A, x)$;
    return 🟦 $Sort(A')$;
  }
}
```

Reducing CbC-SPLs to Proof Repositories



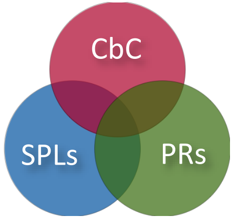
CbC-SPL Transformation

- **Goal**: Express entire CbC-SPLs with the proof repository framework
- **Solution**: Bind calls to actual methods using *abstract contracts*

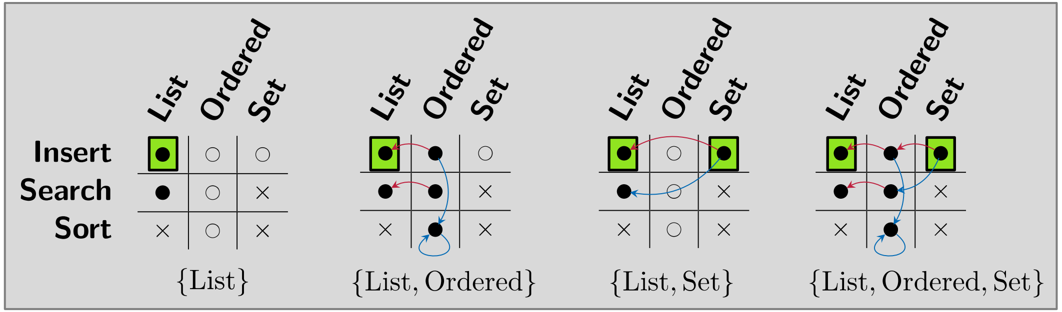


```
class List_Insert   { int[]  🟥 main(...) {} }

class Ordered_Sort { int[]  🟦 main(...) {} }

class Ordered_Insert {
  ...
  /*@ requires  🟥 original_pre(A, x) ∧ isSorted(A);
    @ ensures   🟥 original_post(A, x) ∧ isSorted(A);  @*/
  int[] main(int[] A, int x) {
    A’ =   🟥 original(A, x);
    return 🟦 Sort(A′);
  }
}
```

Reducing CbC-SPLs to Proof Repositories



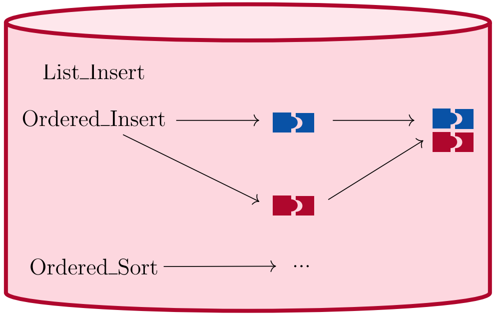
Proof Reuse



Potential for reuse

1. avoid obvious re-verification
2. leverage overlaps

... with proof repositories:



Solution

1. *structural reuse* (SR)
2. *partial proof reuse* (PPR)

Implementation

KeYPR

- **KeY** for **P**roof **R**epositories
- implementation of proof repositories for CbC-SPLs developed with Java/JML
- CbC-SPLs are written in a Lisp-based **DSL**
- uses **KeY with abstract contracts** [3] for conducting proofs
- implements four **query strategies**

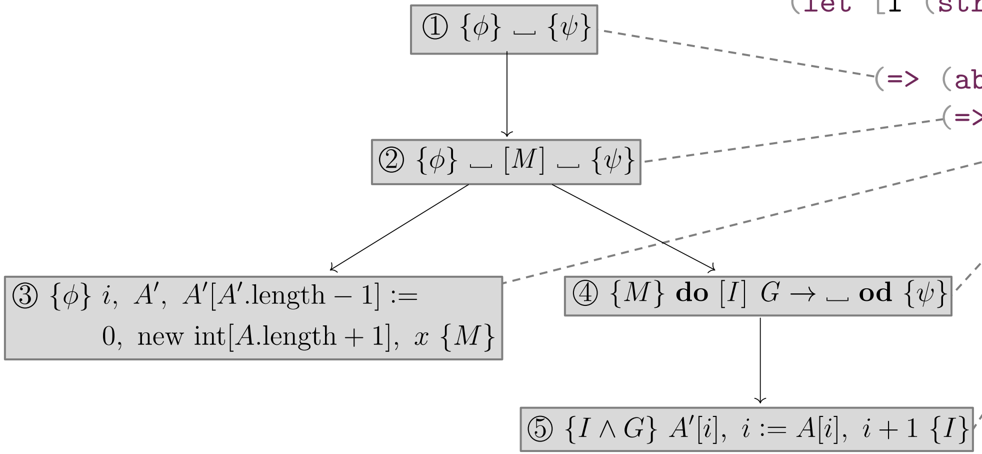
---

[3] Maria Pelevina. 2014. Realization and Extension of Abstract Operation Contracts for Program Logic. Bachelor's Thesis. Technische Universität Darmstadt.

Implementation

KeYPR: DSL

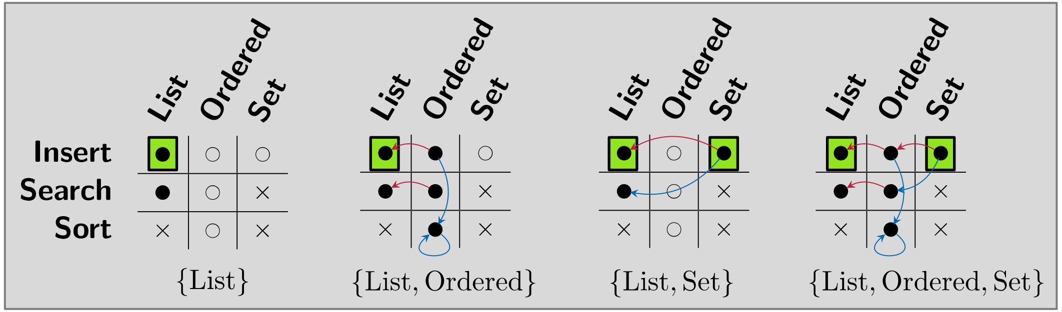
```
(CbC-SPL
 ["List" "Ordered" "Set"]                         ; features
 [["List"] ... ["List" "Ordered" "Set"]]          ; configurations
 [[:strategy-property "NON_LIN_ARITH = DEF_OPS"]] ; KeY options
 [(M "int[] ::List_Insert(int[] A, int x)"        ; method signature
     (T "A.length > 0"                            ; precondition
        "app(A’, x) && appAll(A’, A)"             ; postcondition
        (let [I (str "A’.length == A.length + 1"  ; loop invariant
                     "&& A’[A’.length - 1] == x && appIn(A’,0,i,A)")]
             (=> (abstract-statement)             ; refinements
                 (=> (composition (str I " && i == 0"))
                     (=> (assignment ["i" "A’" "A’[A’.length-1]"]
                                     ["0" "new int[A.length + 1]" "x"]))
                     (=> (repetition I "A.length - i" "i < A’.length")
                         (=> (assignment ["A’[i]" "i"]
                                         ["A[i]" "i + 1"]))))))))])
```

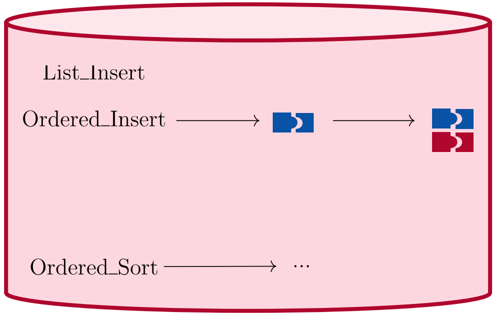


Implementation

KeYPR: Query Strategies

SR (🛢)	PPR (🧩)	Classification





Evaluation

Research Questions

$\mathbf{RQ_1}$ Is it feasible to create CbC-SPLs and guarantee their correctness?

$\mathbf{RQ_2}$ Does the choice of parameters influence the required verification effort?

$\mathbf{RQ_3}$ Does our analysis reduce verification effort compared to previous approaches?

We measure *proof nodes* and *verification time*.

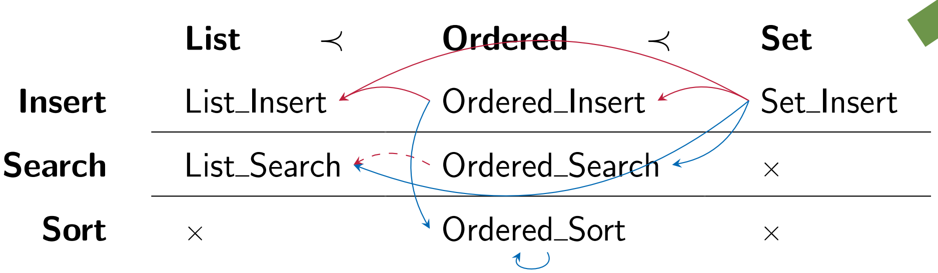
Proofs are cancelled after 10000 nodes or 5 minutes.

Evaluation

Research Questions

RQ$_1$ Is it feasible to create CbC-SPLs and guarantee their correctness?

```
(CbC-SPL
 ["List" "Ordered" "Set"]                          ; features
 [["List"] ... ["List" "Ordered" "Set"]]           ; configurations
 [[:strategy-property "NON_LIN_ARITH = DEF_OPS"]] ; KeY options
 [(M "int[] ::List_Insert(int[] A, int x)"         ; method signature
     (T "A.length > 0"                             ; precondition
        "app(A’, x) && appAll(A’, A)"              ; postcondition
        (let [I (str "A’.length == A.length + 1"   ; loop invariant
                     "&& A’[A’.length - 1] == x && appIn(A’,0,i,A)")]
             (=> (abstract-statement)              ; refinements
                 (=> (composition (str I " && i == 0"))
                     (=> (assignment ["i" "A’" "A’[A’.length-1]"]
                                     ["0" "new int[A.length + 1]" "x"]))
                     (=> (repetition I "A.length - i" "i < A’.length")
                         (=> (assignment ["A’[i]" "i"]
                                         ["A[i]" "i + 1"]))))))))])
```



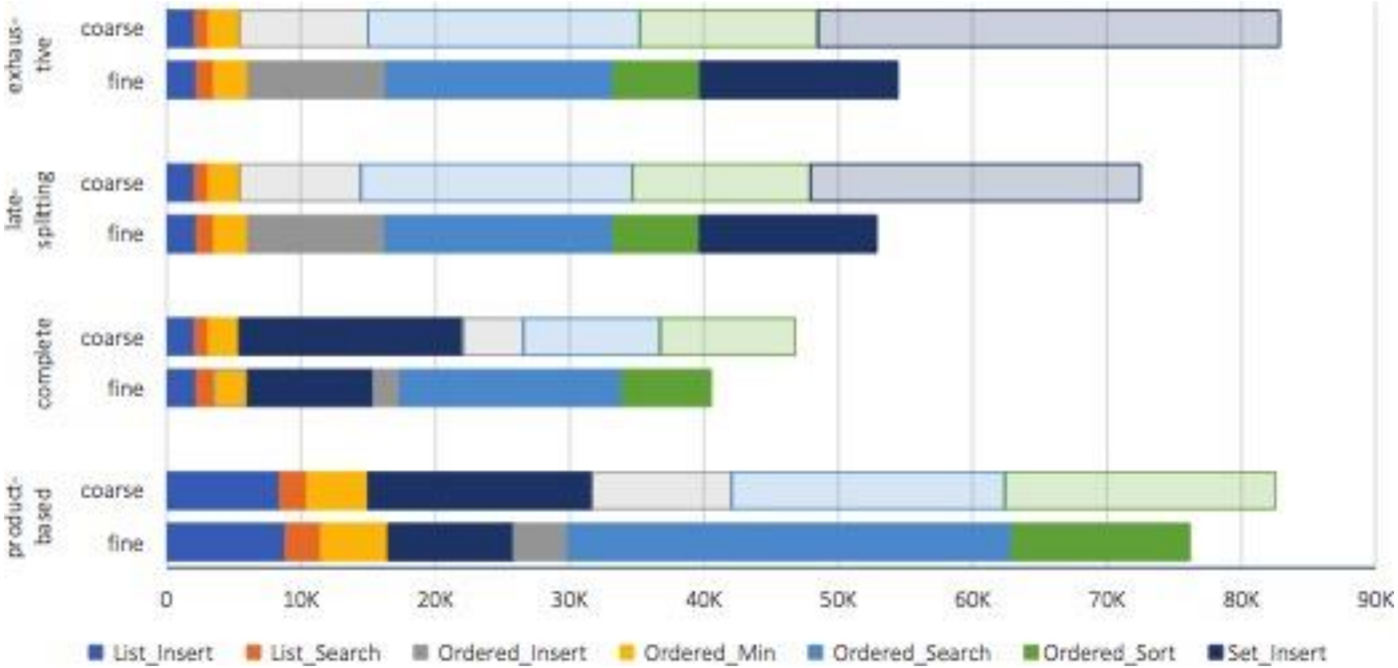
	List ≺	Ordered ≺	Set
Insert	List_Insert	Ordered_Insert	Set_Insert
Search	List_Search	Ordered_Search	×
Sort	×	Ordered_Sort	×

Yes, with minor changes.

Evaluation

Research Questions

$\mathbf{RQ_2}$ Does the choice of parameters influence the required verification effort?



Evaluation

Research Questions

RQ$_2$ Does the choice of parameters influence the required verification effort?

Fine-Grained ≫ Coarse-Grained Transformation

Complete ≫ Late-Splitting ≫ Exhaustive ≫ Product-Based

Evaluation

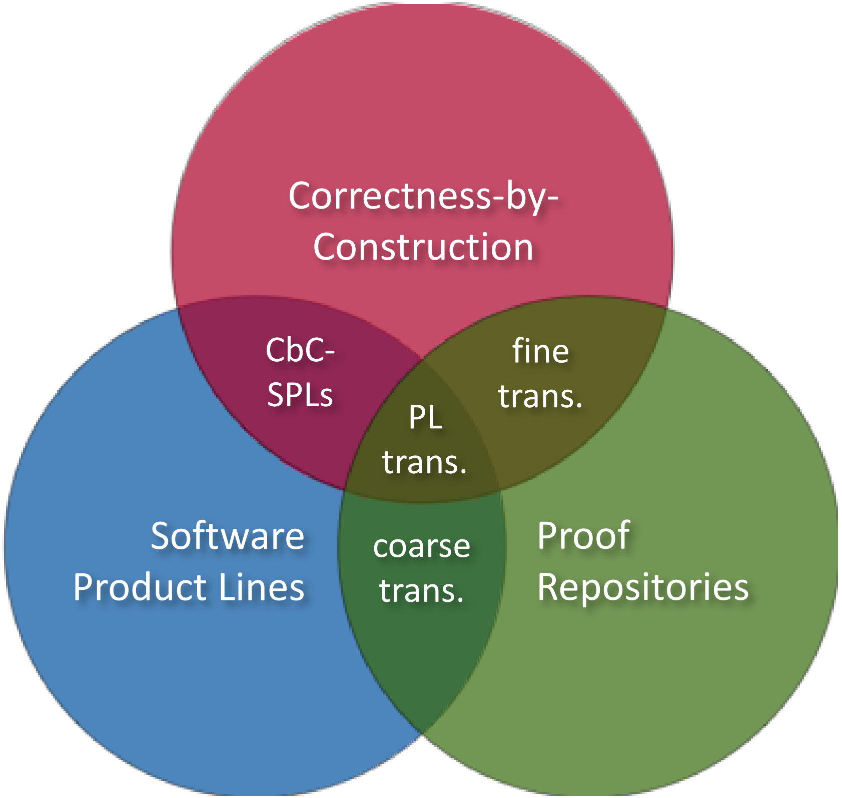
Research Questions

RQ$_3$ Does our analysis reduce verification effort compared to previous approaches?

- **Product-based** analysis	**tailored to CbC:**
- **unoptimized**: Apel/Benduhn/Bolle in *FeatureHouse*	emulated as fine *product-based*
- **optimized**: Bordis/Runge/Kodetzki in *VarCorC*	emulated as fine *complete*
- **Family-based** analysis (Thüm et al. 2012)	N.A.
- **Feature-family-based** analysis (*KeYPR*)	*fine late-splitting*

Optimized Product-Based $\gg$ Feature-Family-Based $\gg$ Unoptimized Product-Based

Conclusion



Correctness-by-Construction

CbC-SPLs

fine trans.

PL trans.

Software Product Lines

coarse trans.

Proof Repositories