Variational Correctness-by-Construction

VaMoS'20, Magdeburg

Tabea Bordis[1], Tobias Runge[1], Alexander Knüppel[1], Thomas Thüm[2] and Ina Schaefer[1]

[1]TU Braunschweig [2]University of Ulm

05. February 2020

Correctness-by-Construction

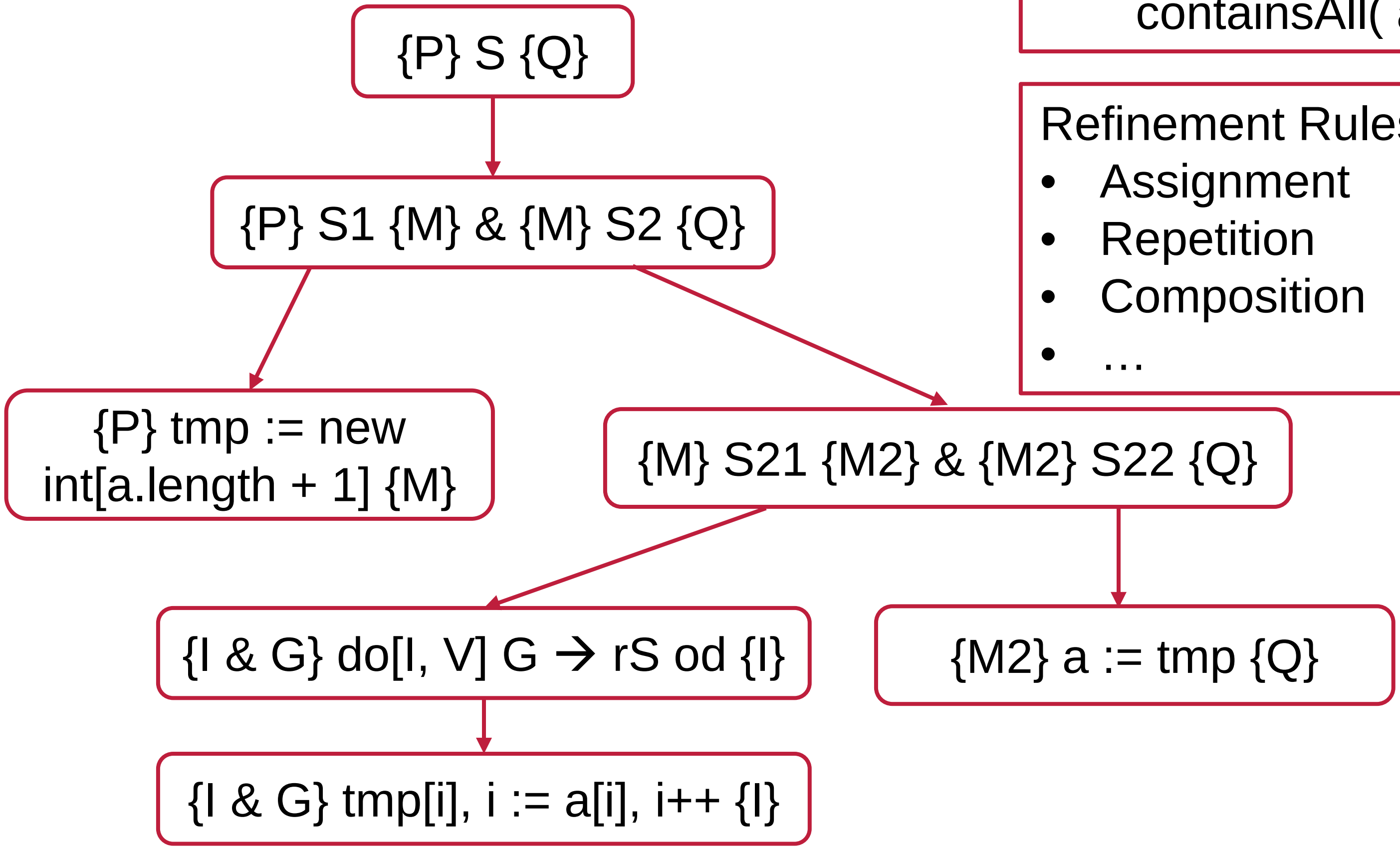
void push(int newTop)

int[] a

P = a != null
Q = contains(a, newTop) &
containsAll(a, 0, a.length, a^{old})

Refinement Rules

- Assignment
- Repetition
- Composition
- …



{P} S {Q}

{P} S1 {M} & {M} S2 {Q}

{P} tmp := new int[a.length + 1] {M}

{M} S21 {M2} & {M2} S22 {Q}

{I & G} do[I, V] G → rS od {I}

{M2} a := tmp {Q}

{I & G} tmp[i], i := a[i], i++ {I}

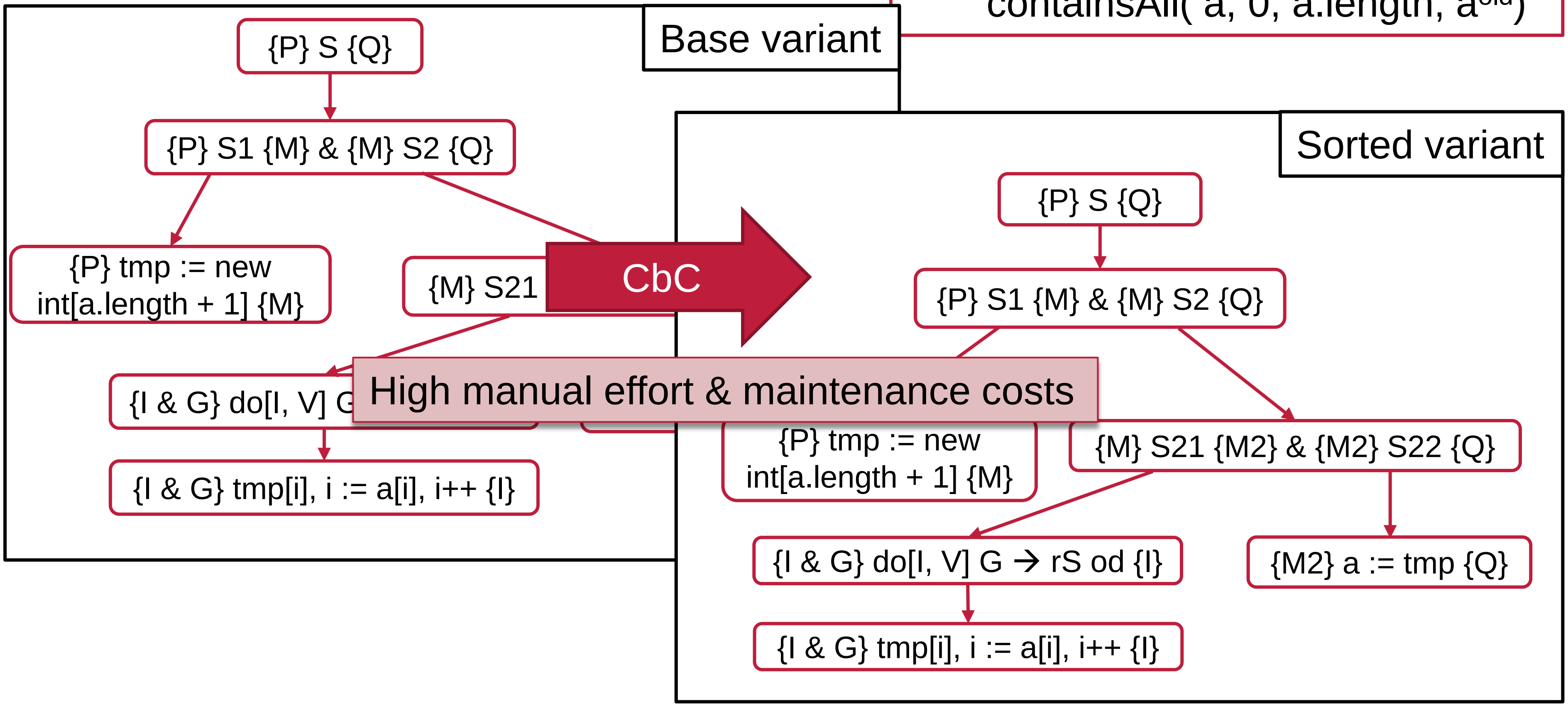
Variants with Correctness-by-Construction

void push(int newTop)

int[] a

P = a != null

Q = contains(a, newTop) &
containsAll(a, 0, a.length, a^{old})


Base variant
{P} S {Q}
{P} S1 {M} & {M} S2 {Q}
{P} tmp := new int[a.length + 1] {M}
{M} S21
{I & G} do[I, V] G
{I & G} tmp[i], i := a[i], i++ {I}
CbC
Sorted variant
{P} S {Q}
{P} S1 {M} & {M} S2 {Q}
{P} tmp := new int[a.length + 1] {M}
{M} S21 {M2} & {M2} S22 {Q}
{I & G} do[I, V] G → rS od {I}
{M2} a := tmp {Q}
{I & G} tmp[i], i := a[i], i++ {I}

High manual effort & maintenance costs

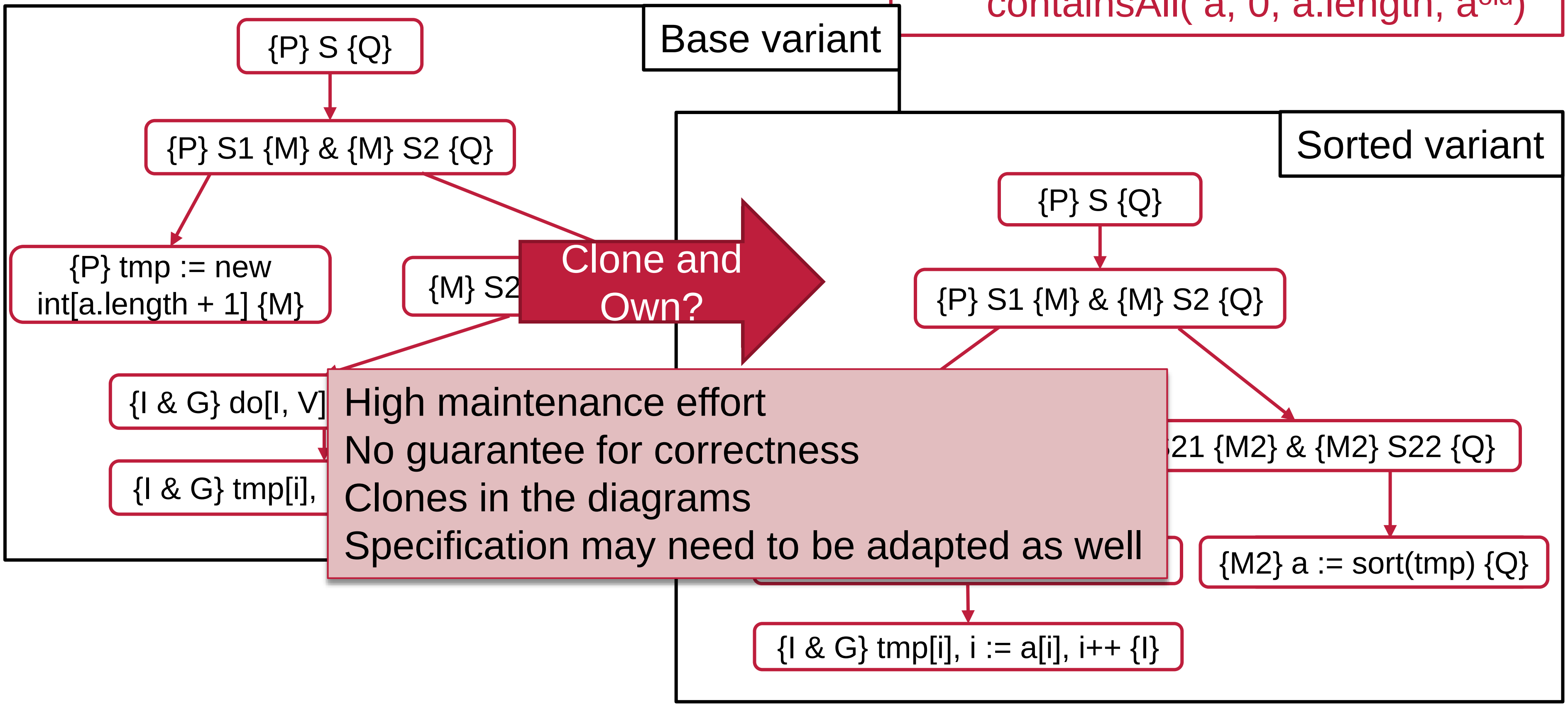
Variants with Correctness-by-Construction

void push(int newTop)

int[] a

P = a != null

Q = contains(a, newTop) &
containsAll(a, 0, a.length, a^{old})

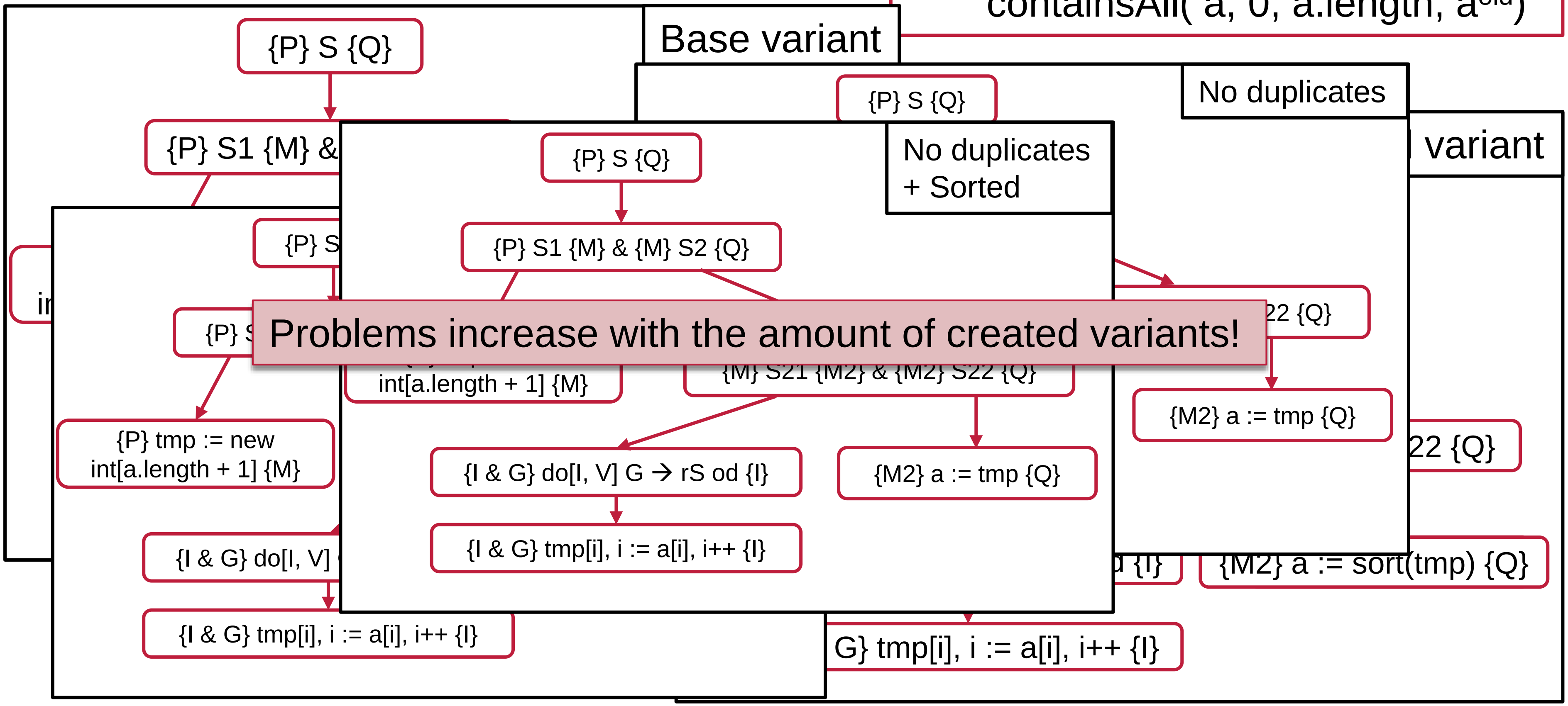


High maintenance effort
No guarantee for correctness
Clones in the diagrams
Specification may need to be adapted as well

Variants with Correctness-by-Construction

void push(int newTop) int[] a

P = a != null
Q = contains(a, newTop) &
containsAll(a, 0, a.length, a^{old})



Base variant

{P} S {Q}

{P} S1 {M} &

in

variant

22 {Q}

d {I}

{M2} a := sort(tmp) {Q}

G} tmp[i], i := a[i], i++ {I}

No duplicates

{P} S {Q}

22 {Q}

{M2} a := tmp {Q}

{P} S

{P} S

{P} tmp := new int[a.length + 1] {M}

{I & G} do[I, V]

{I & G} tmp[i], i := a[i], i++ {I}

No duplicates + Sorted

{P} S {Q}

{P} S1 {M} & {M} S2 {Q}

int[a.length + 1] {M}

{M} S21 {M2} & {M2} S22 {Q}

{I & G} do[I, V] G → rS od {I}

{M2} a := tmp {Q}

{I & G} tmp[i], i := a[i], i++ {I}

Problems increase with the amount of created variants!

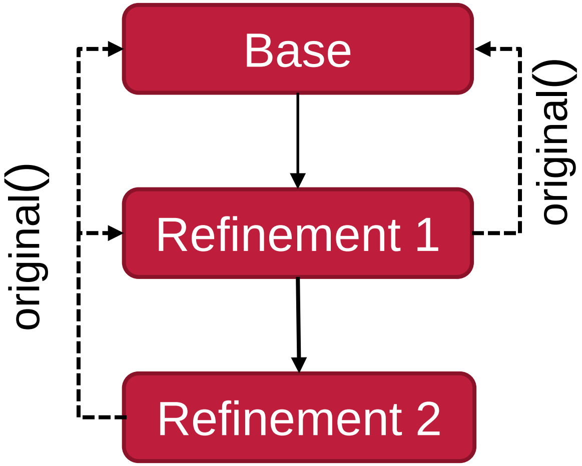
Variational Correctness-by-Construction

1. Variation Point Refinement Rule
   - Variability in the statements
   - Has to preserve the correctness for all valid replacements

2. Contract Composition
   - Variability in the pre- and postcondition
   - Match the conditions to the changed behavior

Variation Points

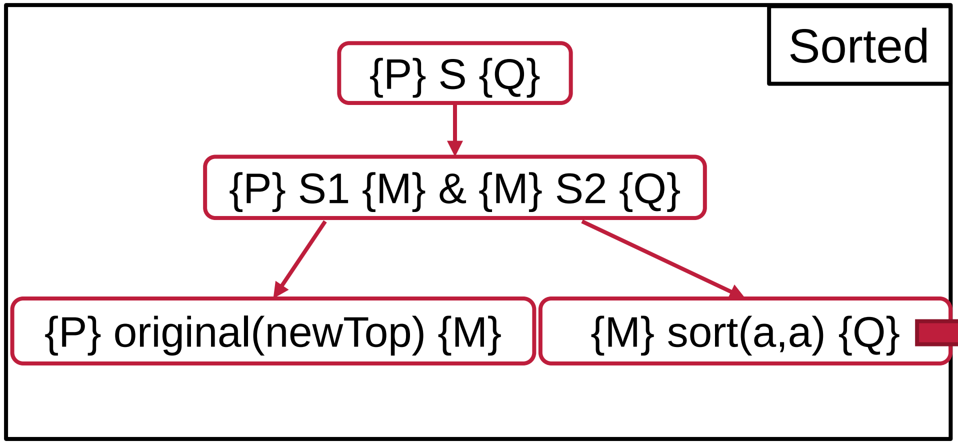
- Similar to feature-oriented programming
  - Refinement hierarchy for methods
  - Each refinement can call **original()** to refer to the implementation of the refinement above



Idea:

- Treat original-call like any method call
- Keep track of valid refinement chains for each method

Variant Sorted – Method Call



{P‘} **sort**(int[] data, return int[] res) {Q‘}
P‘ = true
Q‘ = containsAll(res, 0, res.length, data^{old})
& isSorted(res)

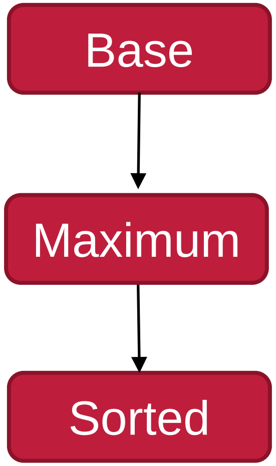
Q‘[data^{old}\a^{old}, res\a] *implies* Q

Method Call

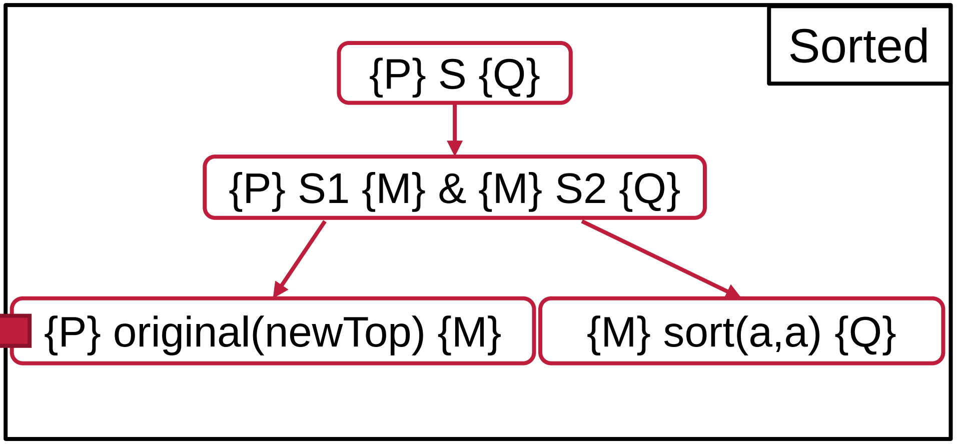
$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to* $\{\mathtt{P}\}\ M(a_1, \ldots, a_n,\ b)\{\mathtt{Q}\}$
with method $\{\mathtt{P}'\}\ M(parameter\ p_1, \ldots, p_n,\ return\ r)\ \{\mathtt{Q}'\}$
iff $\mathtt{P}$ *implies* $\mathtt{P}'[p_i \backslash a_i]$ *and* $\mathtt{Q}'\ [p_i^{old} \backslash a_i^{old}, r \backslash b]$ *implies* $\mathtt{Q}$

- Adapted from [Kourie/Watson, 2012]

Variant Sorted – Variation Point



Replacements:
- Base
- Maximum & Base



P‘= P_Max ● P_Base
Q‘= Q_Max ● Q_Base

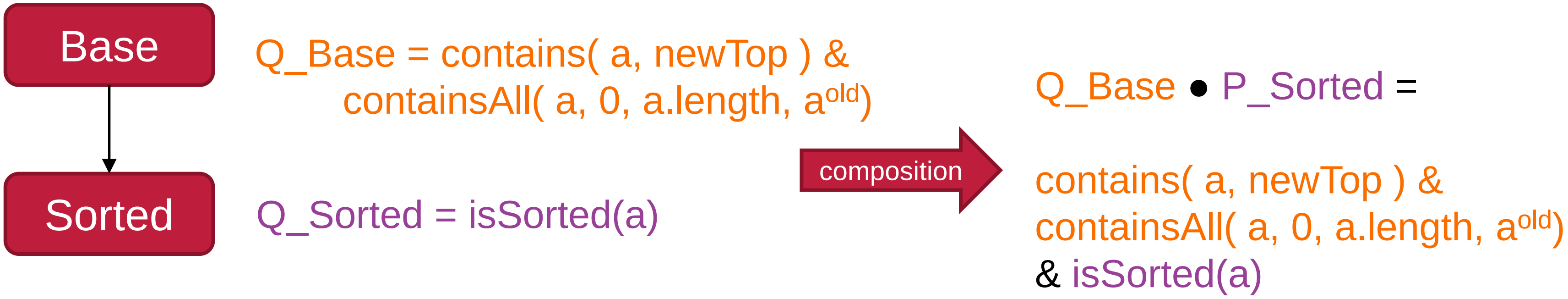
P implies P‘ and Q‘ implies Q

Variation Point

$\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\}$ *can be refined to* $\{\mathtt{P}\}$ **original**$(a_1, ..., a_n,\ b)\ \{\mathtt{Q}\}$
with x *composed methods* $R = \{\mathtt{P}'\}\ M(param\ p_1, .., p_n,\ return\ r)\ \{\mathtt{Q}'\}$
which are composed as $c_1 \bullet c_2 \bullet \ldots \bullet c_l$
with l *method refinements* $c_i = \{\mathtt{P}_i\}\ M(param\ p_1, ..., p_n,\ return\ r)\ \{\mathtt{Q}_i\}$
iff for all R: $\mathtt{P}$ *implies* $\mathtt{P}'[p_j \backslash a_j]$ *and* $\mathtt{Q}'[p_j^{old} \backslash a_j^{old}, r \backslash b]$ *implies* $\mathtt{Q}$

Composition Techniques

- Use contract composition techniques proposed by [Thüm et al, 2019]:
  - Contract Overriding
  - Conjunctive Contract Refinement
  - Explicit Contracting



Q_Base = contains(a, newTop) & containsAll(a, 0, a.length, a^{old})

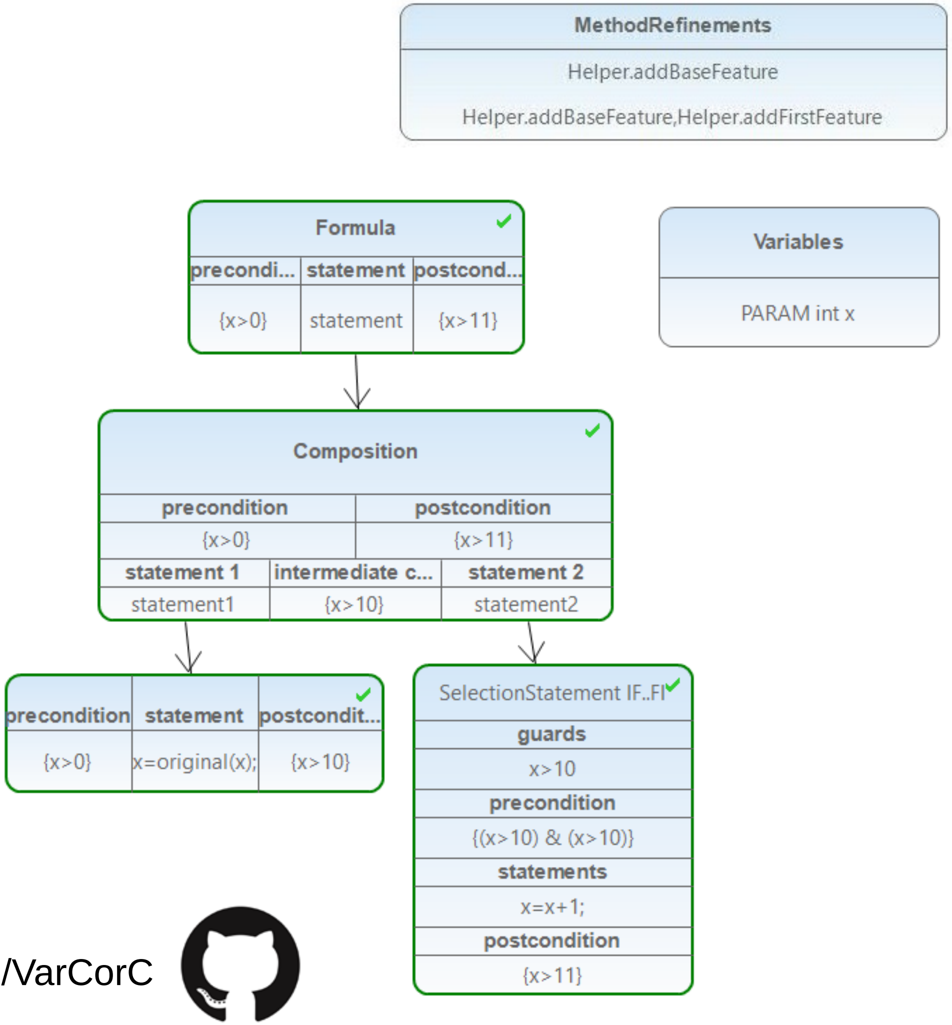
Q_Sorted = isSorted(a)

Q_Base ● P_Sorted =

contains(a, newTop) & containsAll(a, 0, a.length, a^{old}) & isSorted(a)

- Applied to compose **pre- and postcondition** of diagram and for the **variation point refinement rule**

VarCorC



MethodRefinements

Helper.addBaseFeature

Helper.addBaseFeature,Helper.addFirstFeature

Formula ✓

precondi...	statement	postcond...
{x>0}	statement	{x>11}

Variables

PARAM int x

Composition ✓

precondition	postcondition
{x>0}	{x>11}

statement 1	intermediate c...	statement 2
statement1	{x>10}	statement2

✓

precondition	statement	postcondit...
{x>0}	x=original(x);	{x>10}

SelectionStatement IF..FI ✓

guards

x>10

precondition

{(x>10) & (x>10)}

statements

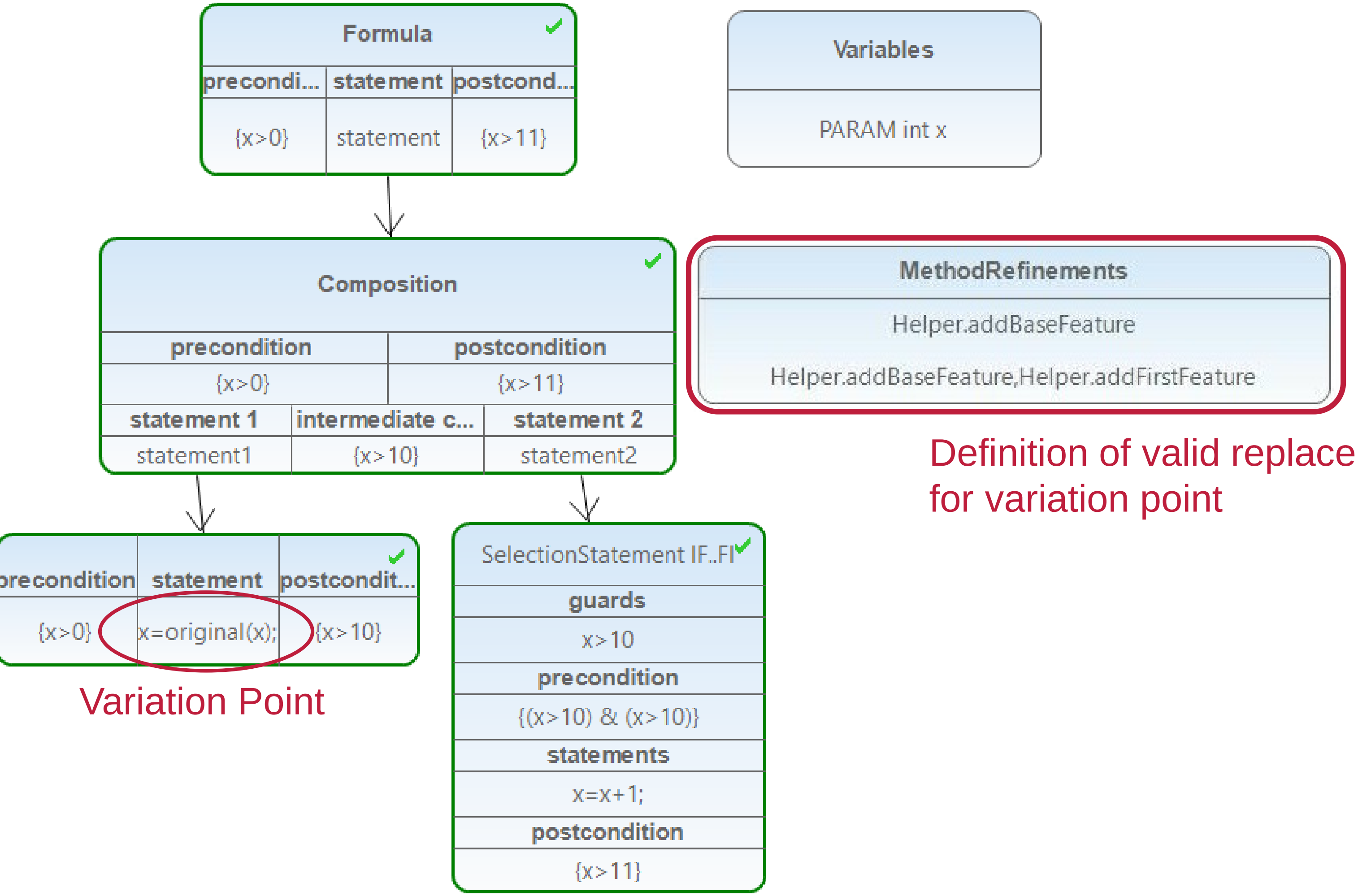
x=x+1;

postcondition

{x>11}

Available at https://github.com/TUBS-ISF/CorC/tree/VarCorC

Example



Formula ✓

precondi...	statement	postcond...
{x>0}	statement	{x>11}

Variables

PARAM int x

Composition ✓

precondition	postcondition
{x>0}	{x>11}

statement 1	intermediate c...	statement 2
statement1	{x>10}	statement2

MethodRefinements

Helper.addBaseFeature

Helper.addBaseFeature,Helper.addFirstFeature

Definition of valid replacements for variation point

precondition	statement	postcondit...
{x>0}	x=original(x);	{x>10}

✓

Variation Point

SelectionStatement IF..FI ✓

guards

x>10

precondition

{(x>10) & (x>10)}

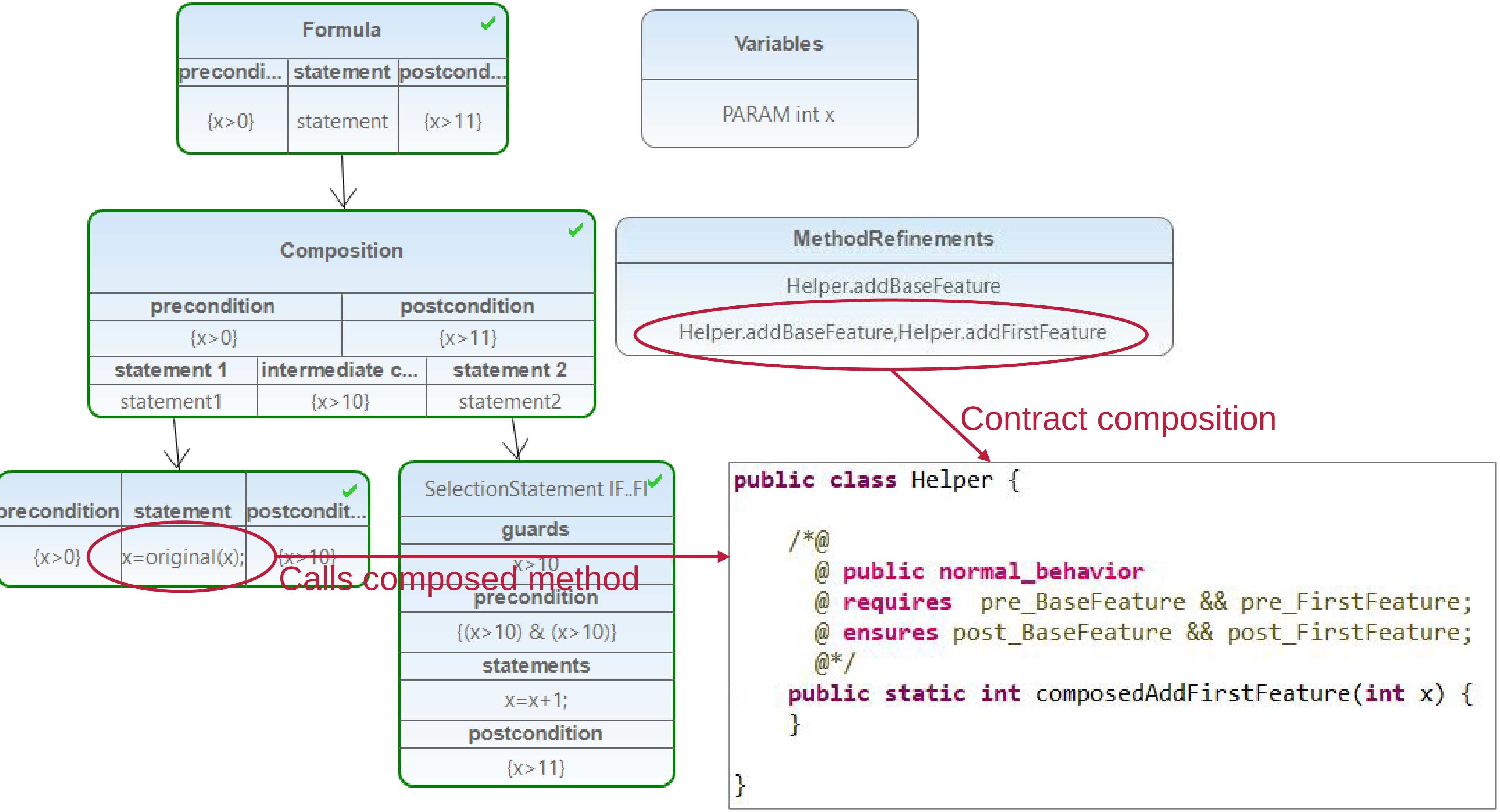
statements

x=x+1;

postcondition

{x>11}

Example



Formula ✓

precondi...	statement	postcond...
{x>0}	statement	{x>11}

Variables

PARAM int x

Composition ✓

precondition	postcondition
{x>0}	{x>11}

statement 1	intermediate c...	statement 2
statement1	{x>10}	statement2

MethodRefinements

Helper.addBaseFeature

Helper.addBaseFeature,Helper.addFirstFeature

Contract composition

✓

recondition	statement	postcondit...
{x>0}	x=original(x);	{x>10}

Calls composed method

SelectionStatement IF..FI ✓

guards

x>10

precondition

{(x>10) & (x>10)}

statements

x=x+1;

postcondition

{x>11}

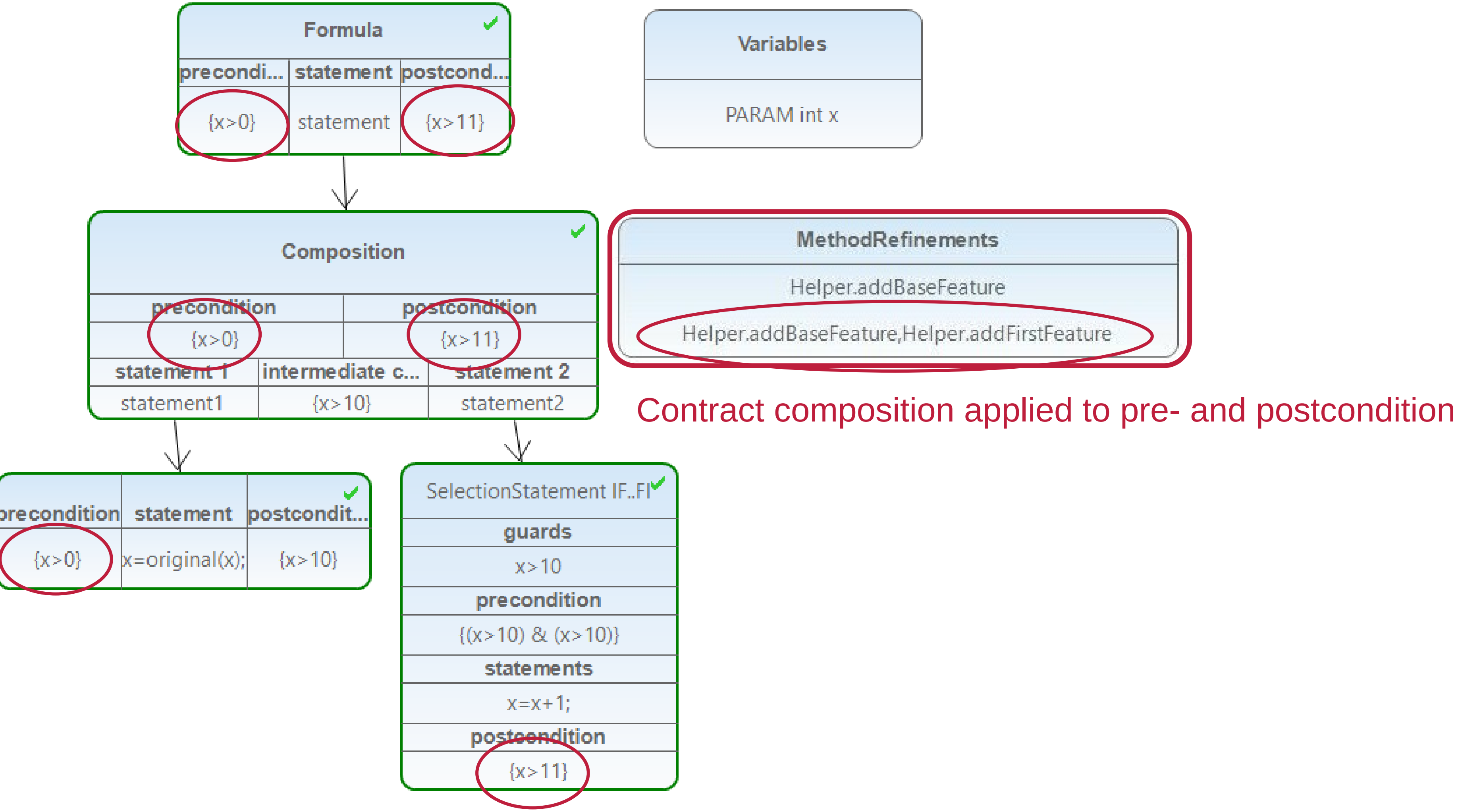
```
public class Helper {

    /*@
     @ public normal_behavior
     @ requires  pre_BaseFeature && pre_FirstFeature;
     @ ensures post_BaseFeature && post_FirstFeature;
     @*/
    public static int composedAddFirstFeature(int x) {
    }

}
```

Technische Universität Braunschweig

Example



Formula ✓

precondi...	statement	postcond...
{x>0}	statement	{x>11}

Variables

PARAM int x

Composition ✓

precondition	postcondition
{x>0}	{x>11}

statement 1	intermediate c...	statement 2
statement1	{x>10}	statement2

MethodRefinements

Helper.addBaseFeature

Helper.addBaseFeature,Helper.addFirstFeature

Contract composition applied to pre- and postcondition

recondition	statement	postcondit...
{x>0}	x=original(x);	{x>10}

SelectionStatement IF..FI ✓

guards

x>10

precondition

{(x>10) & (x>10)}

statements

x=x+1;

postcondition

{x>11}

Evaluation

RQ1: Is it possible to develop variational software using variational correctness-by-construction?

- **IntList**: 1 variational method
  - 3 refinements
- **BankAccount**: 4 variational methods
  - 2 or 3 refinements
- All possible feature configurations could be verified with the recreated CorC diagrams

Evaluation

RQ2**:** What are the **specification costs** compared to **post-hoc verification** with feature-oriented contracts in JML?

- **IntList**: 62% more conjuncted conditions with CbC
- **BankAccount**: 58% more conjuncted conditions with CbC

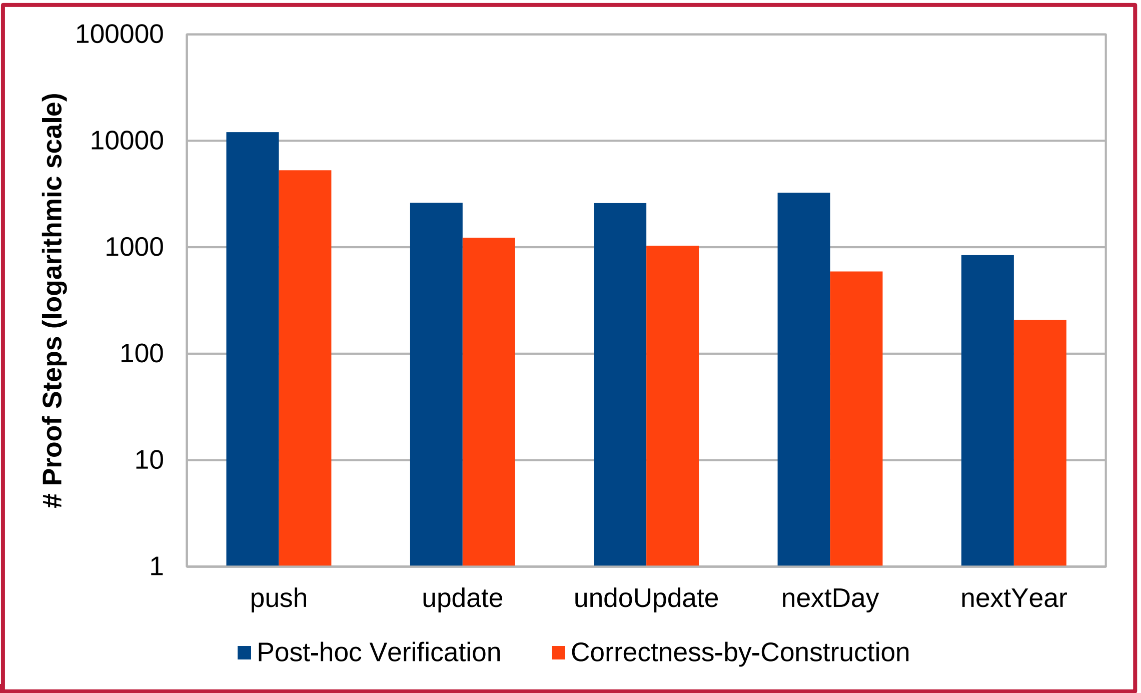
→ Pre- and postconditions almost identical amount of conjuncted conditions

→ 58% of the difference has been due to the intermediate conditions

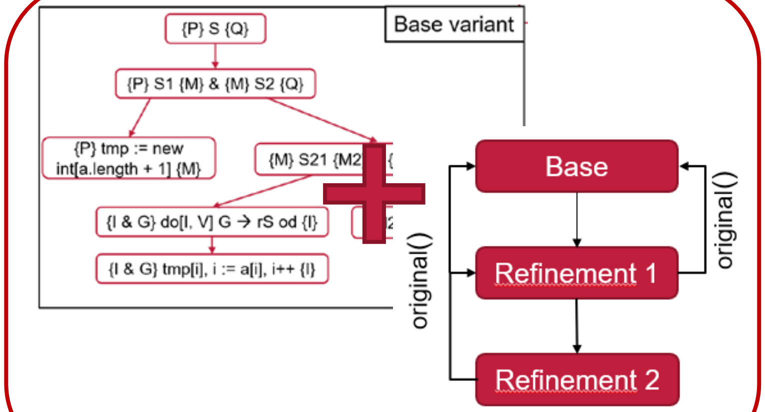
Evaluation

RQ3: What are the **verification costs** compared to **post-hoc verification** with feature-oriented contracts in JML?

- 53 – 81% less proof steps with correctness-by-construction

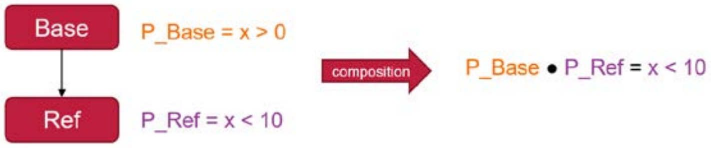


Conclusion

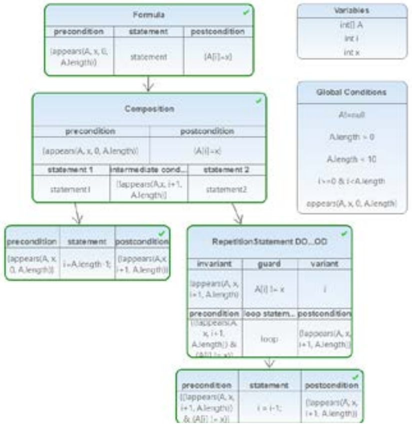


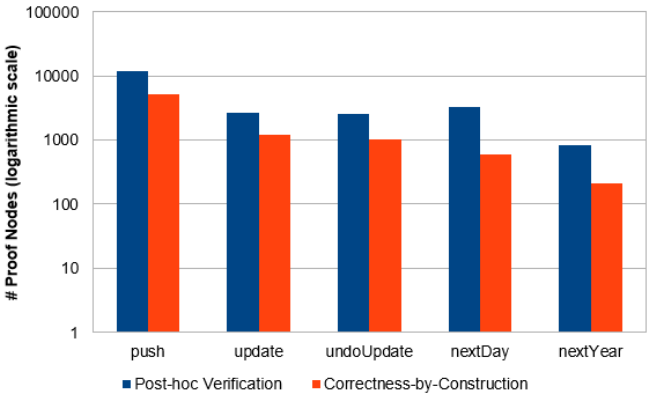
Variation Point

$\{P\}\ S\ \{Q\}$ *can be refined to* $\{P\}$ **original**$(a_1, ..., a_n,\ b)\ \{Q\}$
with x composed methods $R = \{P'\}\ M(param\ p_1, .., p_n,\ return\ r)\ \{Q'\}$
which are composed as $c_1 \bullet c_2 \bullet \ldots \bullet c_l$
with l method refinements $c_i = \{P_i\}\ M(param\ p_1, ..., p_n,\ return\ r)\ \{Q_i\}$
iff for all R: P *implies* $P'[p_j \backslash a_j]$ *and* $Q'[p_j^{old} \backslash a_j^{old}, r \backslash b]$ *implies* Q



VarCorC


Formula
precondition
statement
postcondition
{appears(A, x, 0, A.length)}
statement
{A[i]=x}
Variables
int[] A
int i
int x
Global Conditions
A!=null
A.length > 0
A.length < 10
i>=0 & i<A.length
appears(A, x, 0, A.length)
Composition
precondition
postcondition
{appears(A, x, 0, A.length)}
{A[i]=x}
statement 1
intermediate cond...
statement 2
statement1
{!appears(A,x, i+1, A.length)}
statement2
precondition
statement
postcondition
{appears(A, x, 0, A.length)}
i=A.length-1;
{!appears(A,x, i+1, A.length)}
RepetitionStatement DO...OD
invariant
guard
variant
!appears(A, x, i+1, A.length)
A[i] != x
i
precondition
loop statem...
postcondition
{!appears(A, x, i+1, A.length) & (A[i] != x)}
loop
{!appears(A, x, i+1, A.length)}
precondition
statement
postcondition
{(!appears(A, x, i+1, A.length)) & (A[i] != x)}
i = i-1;
{!appears(A, x, i+1, A.length)}


# Proof Nodes (logarithmic scale)
push
update
undoUpdate
nextDay
nextYear
■ Post-hoc Verification
■ Correctness-by-Construction