Delta-based Verification of Software Product Families

Marco Scaletta, Reiner Hähnle,
Dominic Steinhöfel, Richard Bubel

TECHNISCHE UNIVERSITÄT DARMSTADT

Verification

of

Software Product Lines

implemented with

Delta Oriented Programming (ABS)

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     Unit m(Int x) {
5        this.f = this.f + x;
6     }
7
8     Unit m1(Int x) {
9        this.m(x);
10       this.f = this.f + 1;
11    }
12 }
```

Variants:

{ C }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     Unit m(Int x) {
5        this.f = this.f + x;
6     }
7
8     Unit m1(Int x) {
9        this.m(x);
10       this.f = this.f + 1;
11    }
12 }
```

```
1  delta D1 when Feature1{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 2;
6        }
7     }
8  }
```

Variants:

{ C, C.D1 }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     Unit m(Int x) {
5        this.f = this.f + x;
6     }
7
8     Unit m1(Int x) {
9        this.m(x);
10       this.f = this.f + 1;
11    }
12 }
```

```
1  delta D1 when Feature1{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 2;
6        }
7     }
8  }
```

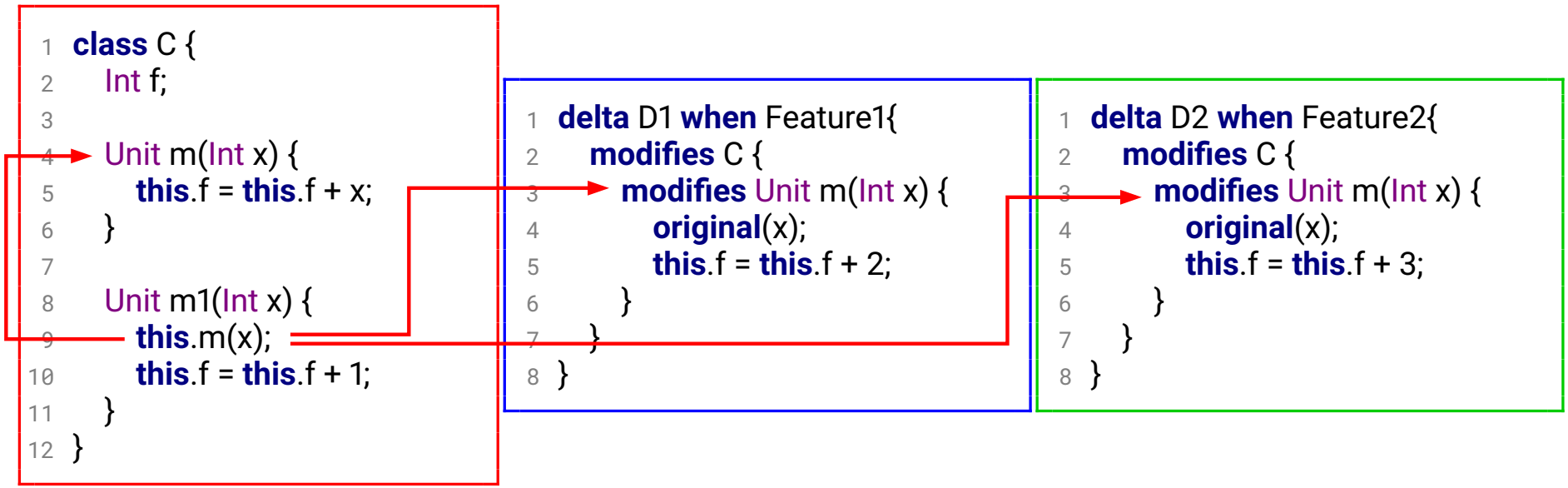
```
1  delta D2 when Feature2{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 3;
6        }
7     }
8  }
```

Variants:

{ C, C.D1, C.D2, C.D1.D2 }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT



```
class C {
    Int f;

    Unit m(Int x) {
        this.f = this.f + x;
    }

    Unit m1(Int x) {
        this.m(x);
        this.f = this.f + 1;
    }
}
```

```
delta D1 when Feature1{
    modifies C {
        modifies Unit m(Int x) {
            original(x);
            this.f = this.f + 2;
        }
    }
}
```

```
delta D2 when Feature2{
    modifies C {
        modifies Unit m(Int x) {
            original(x);
            this.f = this.f + 3;
        }
    }
}
```

Variants:

{ C, C.D1, C.D2, C.D1.D2 }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     Unit m(Int x) {
5        this.f = this.f + x;
6     }
7
8     Unit m1(Int x) {
9        this.m(x);
10       this.f = this.f + 1;
11    }
12 }
```

```
1  delta D1 when Feature1{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 2;
6        }
7     }
8  }
```

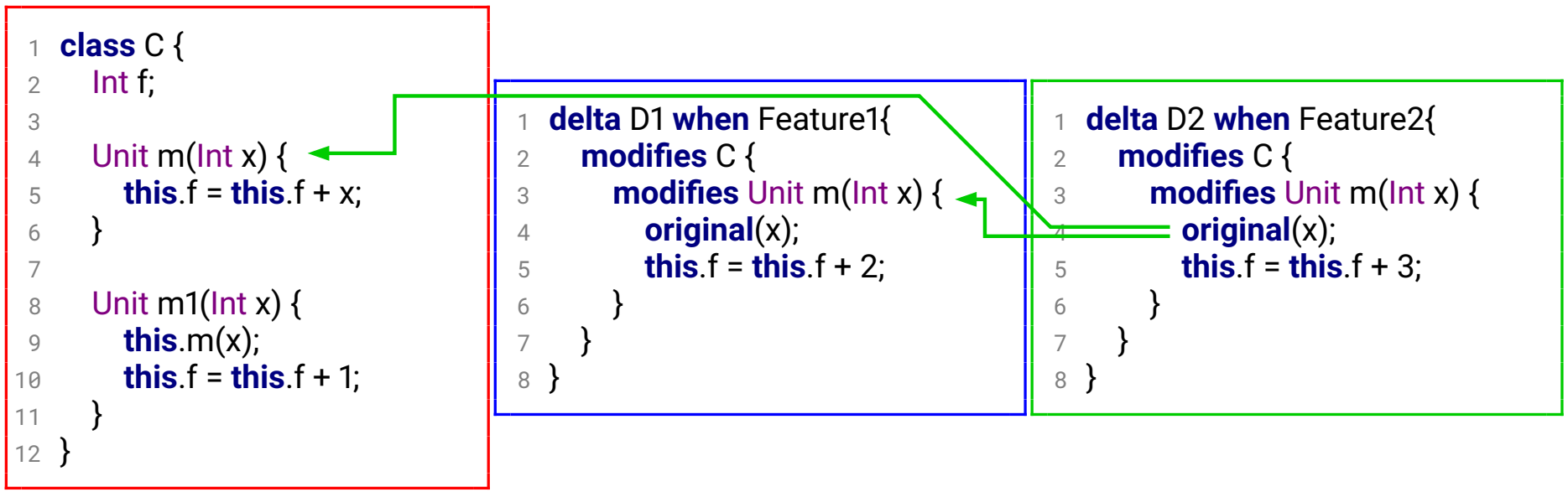
```
1  delta D2 when Feature2{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 3;
6        }
7     }
8  }
```

Variants:

{ C, C.D1, C.D2, C.D1.D2 }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT



```
class C {
   Int f;

   Unit m(Int x) {
      this.f = this.f + x;
   }

   Unit m1(Int x) {
      this.m(x);
      this.f = this.f + 1;
   }
}
```

```
delta D1 when Feature1{
   modifies C {
      modifies Unit m(Int x) {
         original(x);
         this.f = this.f + 2;
      }
   }
}
```

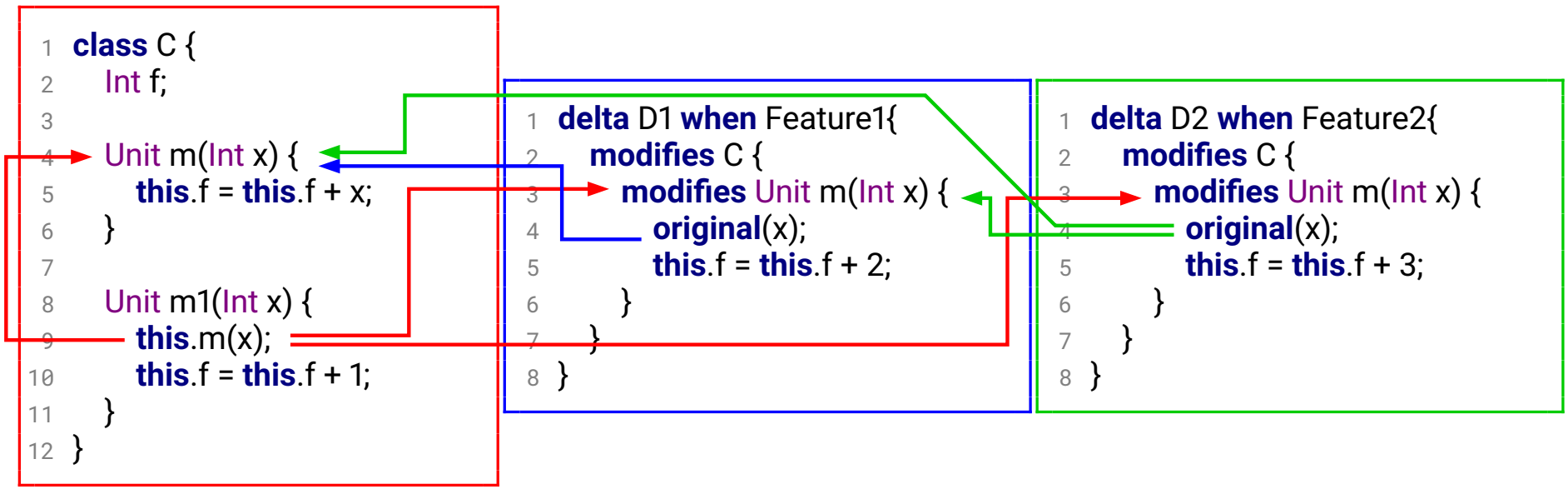
```
delta D2 when Feature2{
   modifies C {
      modifies Unit m(Int x) {
         original(x);
         this.f = this.f + 3;
      }
   }
}
```

Variants:

{ C, C.D1, C.D2, C.D1.D2 }

Delta Oriented Programming

TECHNISCHE UNIVERSITÄT DARMSTADT



Variants:

{ C, C.D1, C.D2, C.D1.D2 }

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

`(m.pre, m.post, m.frame)`

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True; ⇐ precondition
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

`(m.pre, m.post, m.frame)`

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True; ⇐ precondition
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, m.post, m.frame)
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x; ⇐ postcondition
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, m.post, m.frame)
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x; ⇐ postcondition
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, this.f = \old(this.f) + x, m.frame)
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f; ⇐ frame
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, this.f = \old(this.f) + x, m.frame)
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f; ⇐ frame
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, this.f = \old(this.f) + x, {this.f})
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, this.f = \old(this.f) + x, {this.f})
```

Contract-based Approach

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1   class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7
8     Unit m(Int x) {
9       this.f = this.f + x;
10    }
11  }
12
```

Behavior of `m` specified by

```
(True, this.f = \old(this.f) + x, {this.f})
```

!! We focus on normal behaviors !!

Compositional Verification

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7     Unit m(Int x) {
8        this.f = this.f + x;
9     }
10
11    //@requires True;
12    //@ensures this.f = \old(this.f) + x  + 1;
13    //@assignable this.f;
14    Unit m1(Int x) {
15       this.m(x);
16       this.f = this.f + 1;
17    }
18 }
```

Compositional Verification

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@requires True;
5     //@ensures this.f = \old(this.f) + x;
6     //@assignable this.f;
7     Unit m(Int x) {
8        this.f = this.f + x;
9     }
10
11    //@requires True;
12    //@ensures this.f = \old(this.f) + x  + 1;
13    //@assignable this.f;
14    Unit m1(Int x) {
15       this.m(x);
16       this.f = this.f + 1;
17    }
18 }
```

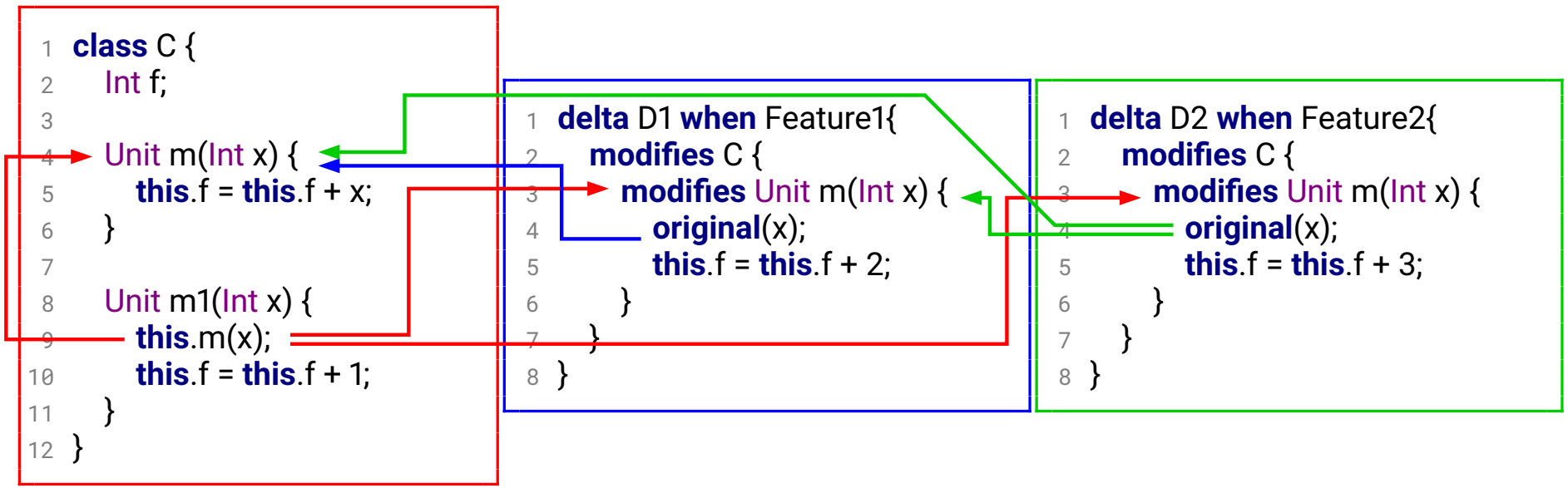
Compositional Verification

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3
4      //@requires True;
5      //@ensures this.f = \old(this.f) + x;
6      //@assignable this.f;
7      Unit m(Int x) {
8          this.f = this.f + x;
9      }
10
11     //@requires True;
12     //@ensures this.f = \old(this.f) + x + 1;
13     //@assignable this.f;
14     Unit m1(Int x) {
15         this.m(x);
16         this.f = this.f + 1;
17     }
18 }
```

Compositional Verification ??

TECHNISCHE UNIVERSITÄT DARMSTADT



```
class C {
    Int f;

    Unit m(Int x) {
        this.f = this.f + x;
    }

    Unit m1(Int x) {
        this.m(x);
        this.f = this.f + 1;
    }
}
```

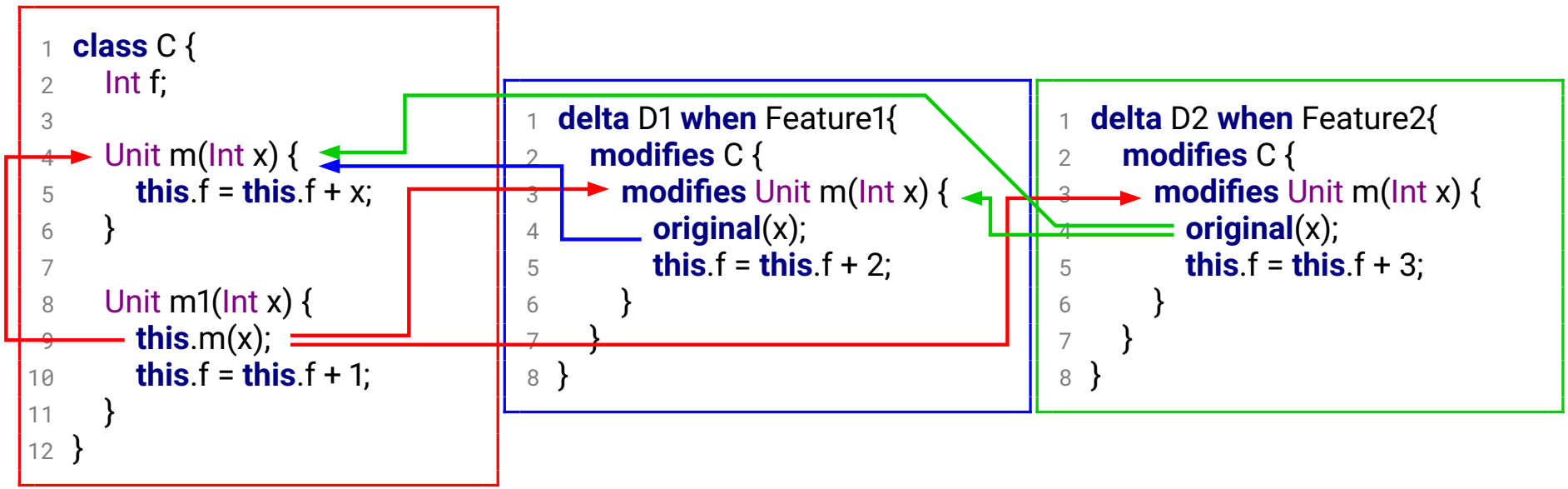
```
delta D1 when Feature1{
    modifies C {
        modifies Unit m(Int x) {
            original(x);
            this.f = this.f + 2;
        }
    }
}
```

```
delta D2 when Feature2{
    modifies C {
        modifies Unit m(Int x) {
            original(x);
            this.f = this.f + 3;
        }
    }
}
```

Straightforward solution: **product-based** verification

Compositional Verification ??

TECHNISCHE UNIVERSITÄT DARMSTADT



```
1  class C {
2     Int f;
3
4     Unit m(Int x) {
5        this.f = this.f + x;
6     }
7
8     Unit m1(Int x) {
9        this.m(x);
10       this.f = this.f + 1;
11    }
12 }
```

```
1  delta D1 when Feature1{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 2;
6        }
7     }
8  }
```

```
1  delta D2 when Feature2{
2     modifies C {
3        modifies Unit m(Int x) {
4           original(x);
5           this.f = this.f + 3;
6        }
7     }
8  }
```

Straightforward solution: **product-based** verification

⇓

Expensive: **exponential complexity** $O(2^N * M)$

(N nr. of deltas, M nr. of methods)

Liskov Substitution Principle for DOP

Hähnle and Schaefer (2012)

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`C.m.contract` ***is-more-general-than*** `D1.C.m.contract`

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`C.m.contract` $\succeq$ `D1.C.m.contract`

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`(C.m.preq, C.m.post, C.m.frame)` $\succeq$ `(D1.C.m.preq, D1.C.m.post, D1.C.m.frame)`

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`(m.preq, m.post, m.frame)` $\succeq$ `(m.preq, m.post, m.frame)`

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`(m.preq, m.post, m.frame)` $\succeq$ `(m.preq, m.post, m.frame)`

if and only if

m.preq $\Rightarrow$ m.preq

m.post $\Leftarrow$ m.post

m.frame $\supseteq$ m.frame

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`(m.preq, m.post, m.frame)` $\succeq$ `(m.preq, m.post, m.frame)`

if and only if

$$\text{True} \Rightarrow \text{True}$$

$$\text{m.post} \Leftarrow \text{m.post}$$

$$\text{m.frame} \supseteq \text{m.frame}$$

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

(m.preq, m.post, m.frame) $\succeq$ (m.preq, m.post, m.frame)

if and only if

True $\Rightarrow$ True

f >= \old(f) + x $\Leftarrow$ f >= \old(f) + x + 2

m.frame $\supseteq$ m.frame

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

(m.preq, m.post, m.frame) $\succeq$ (m.preq, m.post, m.frame)

if and only if

True $\Rightarrow$ True

f >= \old(f) + x $\Leftarrow$ f >= \old(f) + x + 2

{f} $\supseteq$ {f}

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

`C.m.contract` $\succeq$ `D1.C.m.contract`

then

`D1.m` satisfies `D1.m.contract` $\Longrightarrow$ `D1.m` satisfies `C.m.contract`

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

A *correct* method `m` **fulfills** every contract more general than its own

More-general-than Relation ($\succeq$)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2      Int f;
3      //@requires True;
4      //@ensures f >= \old(f) + x;
5      //@assignable f;
6      Unit m(Int x) {
7          this.f = this.f + x;
8      }
9
10     //...
11 }
```

```
1  delta D1 when Feature1{
2      modifies C {
3          //@requires True;
4          //@ensures f >= \old(f) + x  + 2;
5          //@assignable f;
6          modifies Unit m(Int x) {
7              original(x);
8              this.f = this.f + 2;
9          }
10     }
11 }
```

A *correct* method `m` **fulfills** every contract more general than its own

When **calling `m`** we can consider its **most general contract**

Liskov Substitution Principle for DOP

Hähnle and Schaefer (2012)

TECHNISCHE UNIVERSITÄT DARMSTADT

For each version of `m`
its contract is **more-specific-than each previous one**

Liskov Substitution Principle for DOP

Hähnle and Schaefer (2012)

TECHNISCHE UNIVERSITÄT DARMSTADT

For each version of `m`
its contract is **more-specific-than each previous one**

+

Each method is **Liskov-verified**:

*for each method call we consider its **most general contract***

Liskov Substitution Principle for DOP

Hähnle and Schaefer (2012)

TECHNISCHE UNIVERSITÄT DARMSTADT

For each version of `m`
its contract is **more-specific-than each previous one**

+

Each method is **Liskov-verified**:

*for each method call we consider its **most general contract***

⇓

Every variant **is correct**

Liskov Substitution Principle for DOP

Hähnle and Schaefer (2012)

TECHNISCHE UNIVERSITÄT DARMSTADT

For each version of `m`
its contract is **more-specific-than each previous one**

+

Each method is **Liskov-verified**:

*for each method call we consider its **most general contract***

⇓

Every variant **is correct**

+

No variant is **generated**: polynomial complexity **O(N*M)**

(N=nr. of deltas, M=nr. of methods)

Liskov Substitution Principle for DOP: PROBLEM

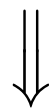
TECHNISCHE UNIVERSITÄT DARMSTADT

Program behaviors can **only** become **more specific**

Liskov Substitution Principle for DOP: PROBLEM

TECHNISCHE UNIVERSITÄT DARMSTADT

Program behaviors can **only** become **more specific**

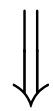


All the versions of a method have
the **same** (most general) **behavior**

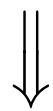
Liskov Substitution Principle for DOP: PROBLEM

TECHNISCHE UNIVERSITÄT DARMSTADT

Program behaviors can **only** become **more specific**



All the versions of a method have
the **same** (most general) **behavior**



Deltas **do not modify** method **behaviors**

Idea

TECHNISCHE UNIVERSITÄT DARMSTADT

Goal: Overcome Liskov's restrictions *without increasing costs*

Idea

TECHNISCHE UNIVERSITÄT DARMSTADT

Goal: Overcome Liskov's restrictions *without increasing costs*

How?

Idea

TECHNISCHE UNIVERSITÄT DARMSTADT

Goal: Overcome Liskov's restrictions *without increasing costs*

How?

An **original call** in `m` refers to a **set of versions** of `m`

Idea

TECHNISCHE UNIVERSITÄT DARMSTADT

Goal: Overcome Liskov's restrictions *without increasing costs*

How?

An **original call** in `m` refers to a **set of versions** of `m`

+

Represent this set abstractly:

original calls as *abstract programs*

Idea

TECHNISCHE UNIVERSITÄT DARMSTADT

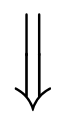
Goal: Overcome Liskov's restrictions *without increasing costs*

How?

An **original call** in `m` refers to a **set of versions** of `m`

+

Represent this set abstractly:
original calls as *abstract programs*



Verifying each delta in isolation

(*delta-based verification*)

How to Deal with Abstract Programs?

TECHNISCHE UNIVERSITÄT DARMSTADT

σ_1

$$x \doteq 0$$
$$y \doteq 2$$

How to Deal with Abstract Programs?

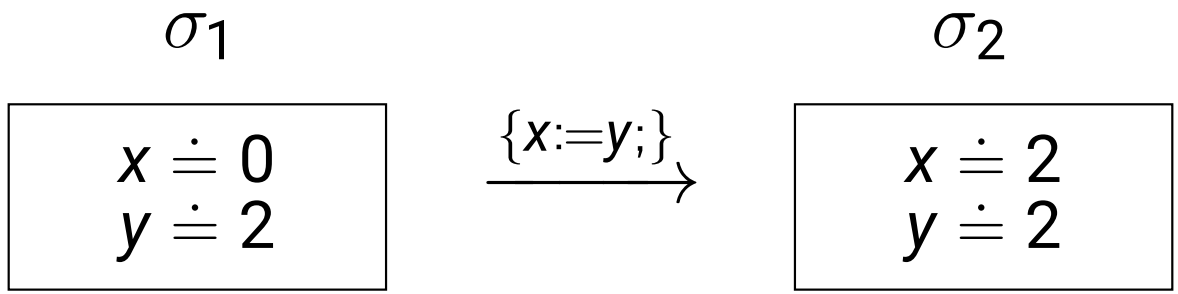
TECHNISCHE UNIVERSITÄT DARMSTADT

$$\sigma_1$$

$$\boxed{\begin{array}{c} x \doteq 0 \\ y \doteq 2 \end{array}} \xrightarrow{\{x := y;\}}$$

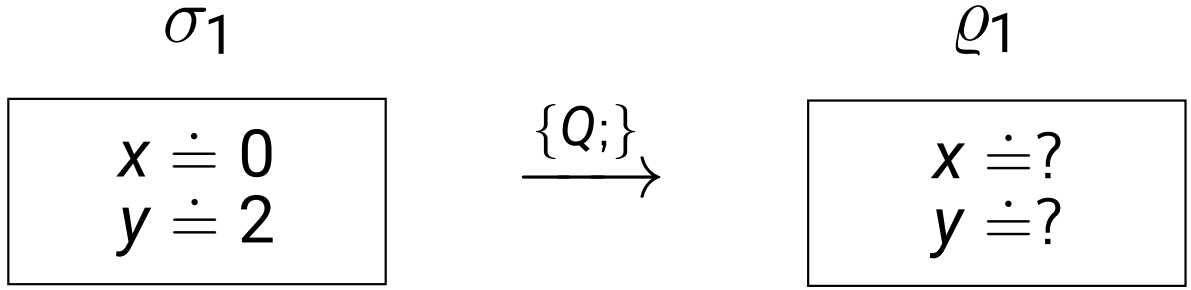
How to Deal with Abstract Programs?

TECHNISCHE UNIVERSITÄT DARMSTADT



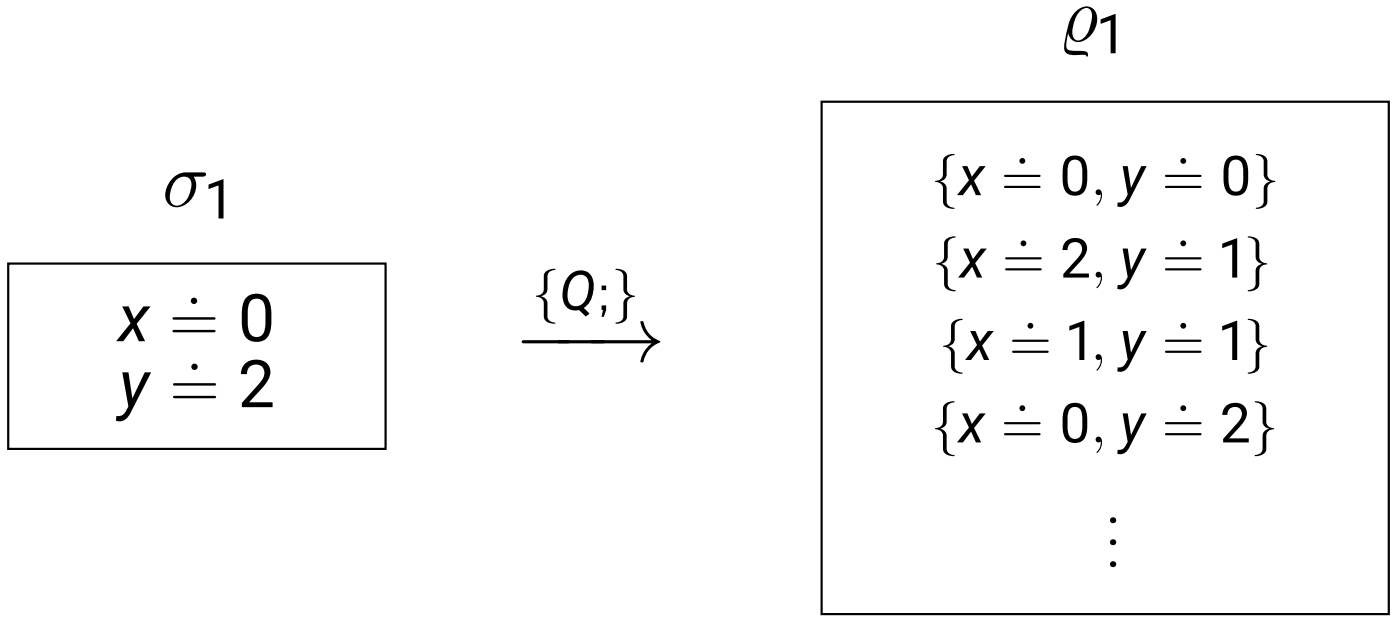
How to Deal with Abstract Programs?

TECHNISCHE UNIVERSITÄT DARMSTADT



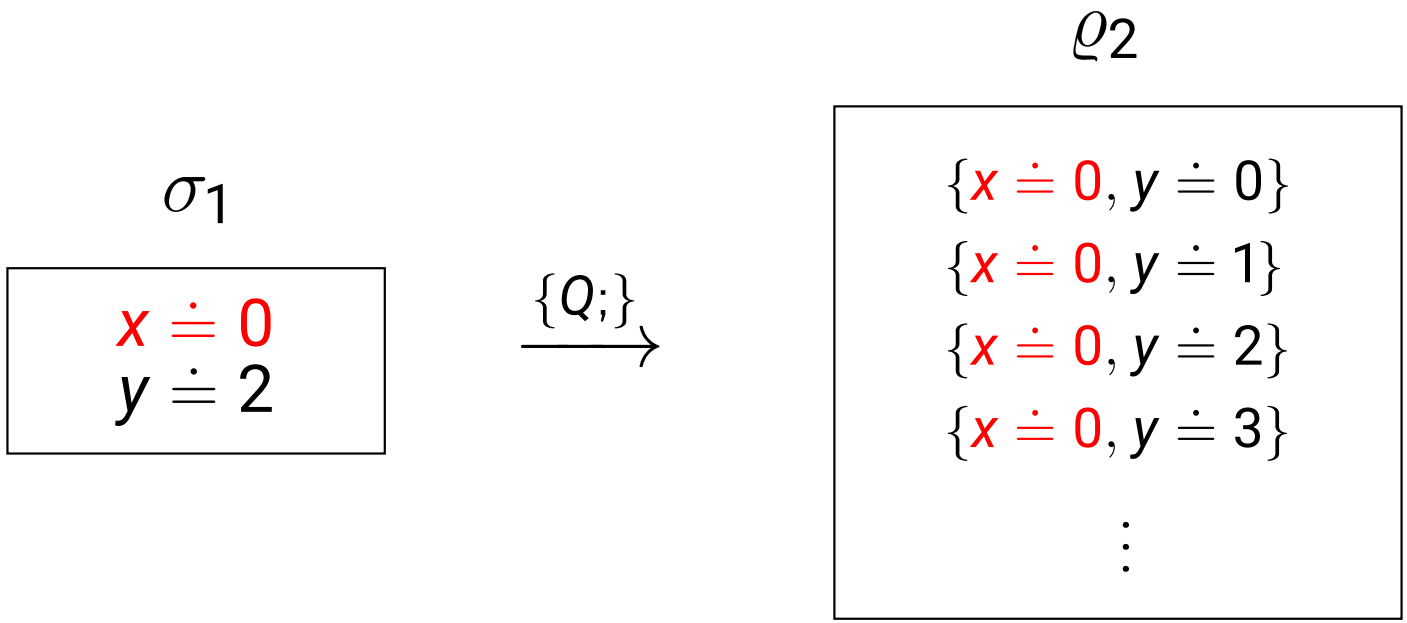
How to Deal with Abstract Programs?

TECHNISCHE UNIVERSITÄT DARMSTADT



How to Deal with Abstract Programs?

TECHNISCHE UNIVERSITÄT DARMSTADT

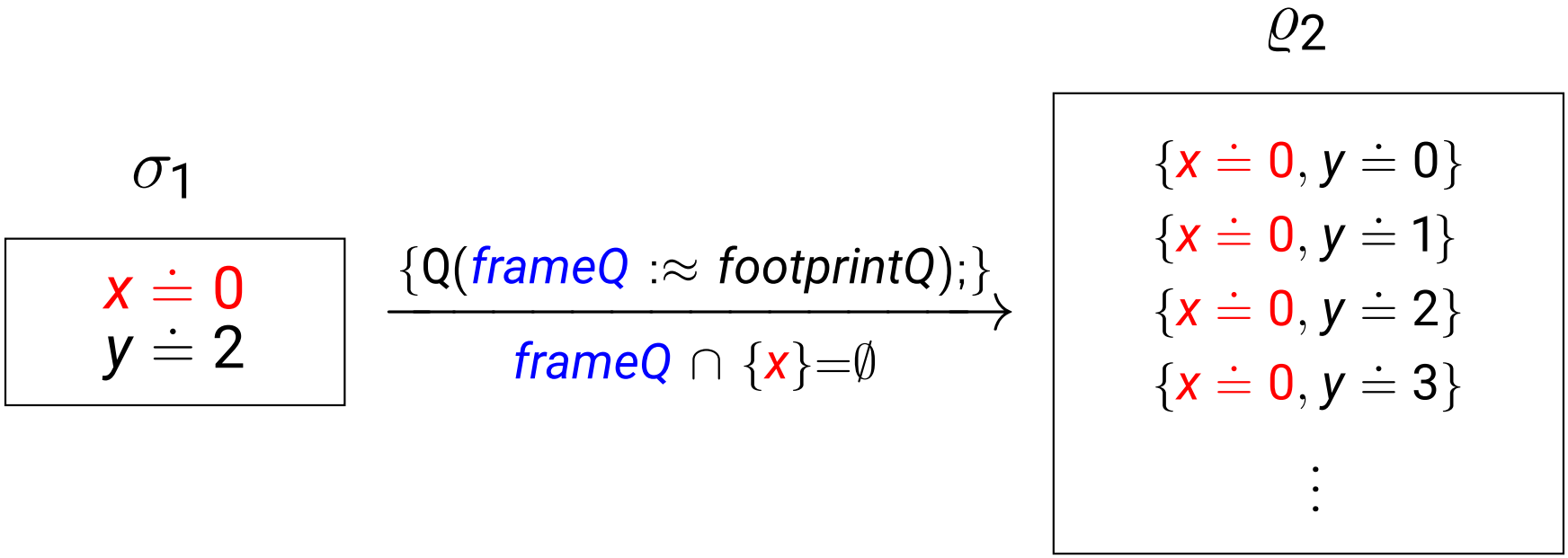


If Q cannot assign x

Abstract Execution

Steinhöfel and Hähnle (2019)

TECHNISCHE UNIVERSITÄT DARMSTADT

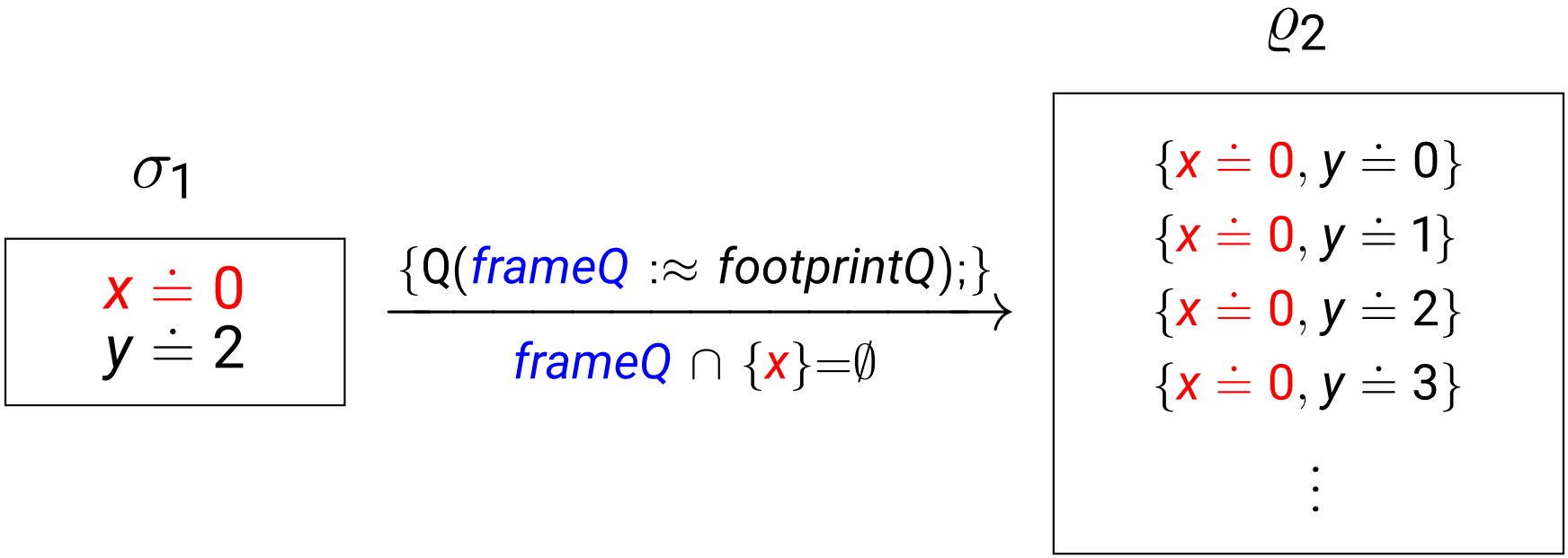


Q is modelled as an **Abstract Statement**

Abstract Execution

Steinhöfel and Hähnle (2019)

TECHNISCHE UNIVERSITÄT DARMSTADT



Proving Universal Behavioral Properties

$x \doteq$ `\old(x)` after executing Q(*frameQ* :≈ *footprintQ*)

Modelling Original Calls as Abstract Statements

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 delta D {
2     modifies C {
3         modifies Unit m(Int x) {
4             label : original(x);
5             //...
6         }
7     }
8 }
```

Modelling Original Calls as Abstract Statements

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 delta D {
2     modifies C {
3         modifies Unit m(Int x) {
4             label : original(x);
5             //...
6         }
7     }
8 }
```

$\texttt{Original}(\mathit{frameO} :\approx \mathit{footprintO})$
$x \notin \mathit{frameO}$

Modelling Original Calls as Abstract Statements

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1 delta D {
2     modifies C {
3         modifies Unit m(Int x) {
4             label: original(x);
5             //...
6         }
7     }
8 }
```

`Original(`*frameO* $:\approx$ *footprintO*`)`
$x \notin$ frameO

`\old (field, label)`

Verifying Deltas with Abstract Execution

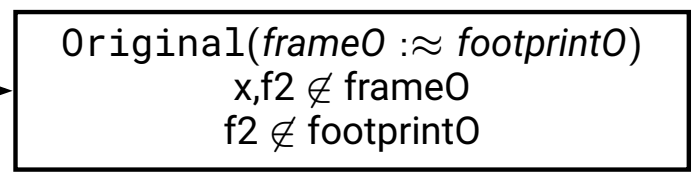
TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```

Verifying Deltas with Abstract Execution

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```



Verifying Deltas with Abstract Execution

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```

Original(*frame0* :≈ *footprint0*)

x,f2 ∉ frame0

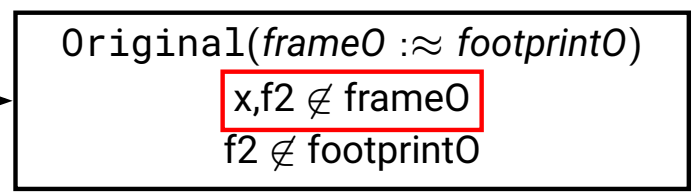
f2 ∉ footprint0

Beyond Liskov!

Verifying Deltas with Abstract Execution

TECHNISCHE UNIVERSITÄT DARMSTADT

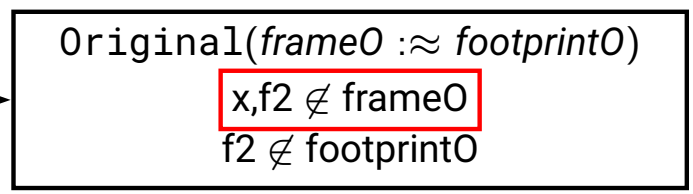
```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```



Verifying Deltas with Abstract Execution

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```



This contract **cannot be invalidate** by delta applications

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; }
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() {
9        m1();
10       f = f + 2;
11    }
12
13    //@ensures this.f == \old(this.f) + 5;
14    Unit m3() {
15       m2();
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; }
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() {
6           original();
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ NOT MODIFIED
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() {
9        m1(); ⇐ NOT MODIFIED
10       f = f + 2;
11    }
12
13    //@ensures this.f == \old(this.f) + 5;
14    Unit m3() {
15       m2();
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; }
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() {
6           original();
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ NOT MODIFIED
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() { ⇐ LISKOV
9        m1(); ⇐ NOT MODIFIED
10       f = f + 2;
11    }
12
13    //@ensures this.f == \old(this.f) + 5;
14    Unit m3() {
15       m2();⇐ LISKOV
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; }
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() { ⇐ LISKOV
6           original();⇐ LISKOV
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ NOT MODIFIED
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() { ⇐ LISKOV
9        m1(); ⇐ NOT MODIFIED
10       f = f + 2;
11    }
12
13    //@ensures ??;
14    Unit m3() {
15       m4(); ⇐ NON LISKOV
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; } ⇐ NON LISKOV
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() { ⇐ LISKOV
6           original();⇐ LISKOV
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }⇐ NON LISKOV
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Behavior Preserving Method)

A *Behavior Preserving Method*:

- does **not call non Behavior Preserving methods**
- is either
  - non **modified**
  - **modified** and **fulfilling Liskov**

Definition (Behavior Preserving Sequence)

Sequence of statements **without calls to non Behavior Preserving methods**

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Behavior Preserving Method)

A *Behavior Preserving Method*:

- does **not call non Behavior Preserving methods**
- is either
  - non **modified**
  - **modified** and **fulfilling Liskov**

Definition (Behavior Preserving Sequence)

Sequence of statements **without calls to non Behavior Preserving methods**

$\Rightarrow$ Its behavior **cannot be modified** by delta application

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ NOT MODIFIED
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() { ⇐ LISKOV
9        m1(); ⇐ NOT MODIFIED
10       f = f + 2;
11    }
12
13    //@ensures ??;
14    Unit m3() {
15       m4(); ⇐ NON LISKOV
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; } ⇐ NON LISKOV
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() { ⇐ LISKOV
6           original();⇐ LISKOV
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }⇐ NON LISKOV
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ BP
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() { ⇐ BP
9        m1(); ⇐ BP
10       f = f + 2;
11    }
12
13    //@ensures ??;
14    Unit m3() {⇐ NON BP
15       m4(); ⇐ NON BP
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; } ⇐ NON BP
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() { ⇐ BP
6           original();⇐ BP
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }⇐ NON BP
12
13    }
14 }
```

Behavior Preserving Method

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f;
3
4     //@ensures this.f == \old(this.f) + 1;
5     Unit m1(){ f = f + 1; } ⇐ BP
6
7     //@ensures this.f >= \old(this.f) + 3;
8     Unit m2() { ⇐ BP
9        m1(); ⇐ BP ⇐ BP-SEQUENCE
10       f = f + 2;
11    }
12
13    //@ensures ??;
14    Unit m3() {⇐ NON BP
15       m4(); ⇐ NON BP
16       f = f + 2;
17    }
18
19    //@ensures this.f == −1;
20    Unit m4(){ f = −1; } ⇐ NON BP
21 }
```

```
1  delta D {
2     modifies C {
3
4        //@ensures this.f >= \old(this.f) + 7;
5        modifies Unit m2() { ⇐ BP
6           original();⇐ BP
7           f = f + 4;
8        }
9
10       //@ensures this.f == 1;
11       modifies Unit m4() { f = 1; }⇐ NON BP
12
13    }
14 }
```

Shallow Single Original Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```

Shallow Single Original Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1; ⇐ BP-SEQUENCE
13          l : original(x);
14          f1 = f1 + 1; ⇐ BP-SEQUENCE
15       }
16    }
17 }
```

Shallow Single Original Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1; ⇐ BP-SEQUENCE
13          l : original(x); ⇐ original call
14          f1 = f1 + 1; ⇐ BP-SEQUENCE
15       }
16    }
17 }
```

Shallow Single Original Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1; ⇐ BP-SEQUENCE
13          l : original(x); ⇐ original call
14          f1 = f1 + 1; ⇐ BP-SEQUENCE
15       }
16    }
17 }
```

Definition

The body of $\delta.\mathtt{m}$ is

$$\{ S_1 \; ; \; \textbf{original}(\overline{p}) \; ; \; S_2 \; ; \}$$

with S_1,S_2 BP-sequences

Refactoring to Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Refactorable

```
1  modifies Unit m(Int x) {
2     if (x > 0) {
3        field1 += 1;
4        m1(); ⇐ BP
5        original(x);
6     } else {
7        original(x);
8        field2 += 1;
9     }
10 }
```

Refactoring to Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Refactorable

```
1  modifies Unit m(Int x) {
2    if (x > 0) {
3      field1 += 1;
4      m1(); ⇐ BP
5      original(x);
6    } else {
7      original(x);
8      field2 += 1;
9    }
10 }
```

Refactoring to Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Refactorable

```
1  modifies Unit m(Int x) {
2     if (x > 0) {
3        field1 += 1;
4        m1(); ⇐ BP
5        original(x);
6     } else {
7        original(x);
8        field2 += 1;
9     }
10 }
```

$\Longrightarrow$

Refactored

```
1  modifies Unit m(Int x) {
2     if (x > 0) {
3        field1 += 1;
4        m1(); ⇐ BP
5     }
6     original(x);
7     if (x<=0) {
8        field2 += 1;
9     }
10 }
```

Refactoring to Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Refactorable

```
1  modifies Unit m(Int x) {
2    if (x > 0) {
3      field1 += 1;
4      m1(); ⇐ BP
5      original(x);
6    } else {
7      original(x);
8      field2 += 1;
9    }
10 }
```

$\Longrightarrow$

Refactored

```
1  modifies Unit m(Int x) {
2    if (x > 0) {
3      field1 += 1;   ⇐ BP-SEQUENCE S1
4      m1(); ⇐ BP
5    }
6    original(x);     ⇐ original call
7    if (x<=0) {
8      field2 += 1;   ⇐ BP-SEQUENCE S2
9    }
10 }
```

Refactoring to Normal Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Refactorable

```
1  modifies Unit m(Int x) {
2     if (x > 0) {
3        field1 += 1;
4        m1(); ⇐ BP
5        original(x);
6     } else {
7        original(x);
8        field2 += 1;
9     }
10 }
```

$\Longrightarrow$

Refactored

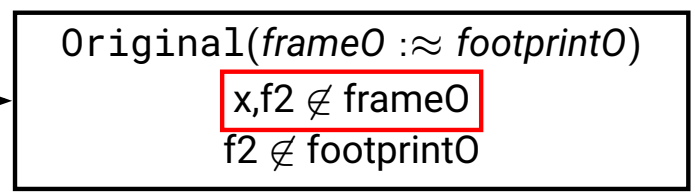
```
1  modifies Unit m(Int x) {
2     if (x > 0) {
3        field1 += 1;   ⇐ BP-SEQUENCE S1
4        m1(); ⇐ BP
5     }
6     original(x);      ⇐ original call
7     if (x<=0) {
8        field2 += 1;   ⇐ BP-SEQUENCE S2
9     }
10 }
```

Using **Abstract Execution** to prove **correctness of refactoring**

Change Old Behavior Principle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```



Change Old Behavior Principle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```

Definition

The body of δ `.m` is either:

a) in normal form
$\{ S_1 \; ; \; \textbf{original}(\overline{p}) \; ; \; S_2 \; ; \}$
where $\mathbf{S_1}$ **assigns new fields**, *or*

b) a BP-sequence.

`Original`(*frame0* :≈ *footprint0*)
x,f2 ∉ frame0
f2 ∉ footprint0

Change Old Behavior Principle

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  class C {
2     Int f1;
3     Unit m(Int x){/*...*/}
4  }
5
6  delta D {
7     modifies C {
8        adds Int f2;
9        //@ensures f2 == \old(f2) + 1
10       //@      && f1 == \old(f1, l) + 1;
11       modifies Unit m(Int x) {
12          f2 = f2 + 1;
13          l : original(x);
14          f1 = f1 + 1;
15       }
16    }
17 }
```

Definition

The body of δ `.m` is either:

a) in normal form
$\{ S_1 \; ; \; \textbf{original}(\overline{p}) \; ; \; S_2 \; ; \}$
where $\mathbf{S_1}$ **assigns new fields**, *or*

b) a BP-sequence.

`Original`(*frameO* :≈ *footprintO*)
x,f2 ∉ frameO
f2 ∉ footprintO

The behavior of $\delta.m$ **cannot be affected** by delta application

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *m is modified*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *m is modified*
     - *fulfills Change Old Behavior Principle, and is verified,* ***or***

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *m is modified*
     - *fulfills Change Old Behavior Principle, and is verified,* ***or***
     - *is a BP-method, and is Liskov-verified*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *m is modified*
     - *fulfills Change Old Behavior Principle, and is verified,* ***or***
     - *is a BP-method, and is Liskov-verified*
   - 2.2 *m is not modified (added), its body is a BP-sequence and is verified*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *`m` is modified*
     - *fulfills Change Old Behavior Principle, and is verified, **or***
     - *is a BP-method, and is Liskov-verified*
   - 2.2 *`m` is not modified (added), its body is a BP-sequence and is verified*
3. *No contract can call a non BP-method.*

Theorem

TECHNISCHE UNIVERSITÄT DARMSTADT

The following theorem guarantees the correctness of this approach.

Theorem

Assume the following:

1. *Each method in the core is a BP-sequence, and verified.*
2. *For each δ.m one of the following holds:*
   - 2.1 *`m` is modified*
     - *fulfills Change Old Behavior Principle, and is verified, **or***
     - *is a BP-method, and is Liskov-verified*
   - 2.2 *`m` is not modified (added), its body is a BP-sequence and is verified*
3. *No contract can call a non BP-method.*

Then each variant is correct.

Normal Form and **Abstract Execution** enable **delta-based verification**

Normal Form and **Abstract Execution** enable **delta-based verification**

+

original is the only callable non BP-method

Normal Form and **Abstract Execution** enable **delta-based verification**

+

original is the only callable non BP-method

⇓

Deltas **cannot affect** the behavior of **non modified methods**

Delta-based Verification of Software Product Families

TECHNISCHE UNIVERSITÄT DARMSTADT

Normal Form and **Abstract Execution** enable **delta-based verification**

+

original is the only callable non BP-method

⇓

Deltas **cannot affect** the behavior of **non modified methods**

⇓

Once a contract is proved it **cannot be invalidate** by delta applications

Delta-based Verification of Software Product Families

TECHNISCHE UNIVERSITÄT DARMSTADT

Normal Form and **Abstract Execution** enable **delta-based verification**

+

original is the only callable non BP-method

⇓

Deltas **cannot affect** the behavior of **non modified methods**

⇓

Once a contract is proved it **cannot be invalidate** by delta applications

⇓

Each contract has to be **proven once**: **polynomial complexity O(N*M)**
(N=nr. of deltas, M=nr. of methods)

Case Studies

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Expressiveness and Performance?

Expressiveness

TECHNISCHE UNIVERSITÄT DARMSTADT

Specification and verification of

- **BankAccountSPL** Thüm et al. (2012)
- **MinePumpPL** Apel et al. (2013)

Expressiveness

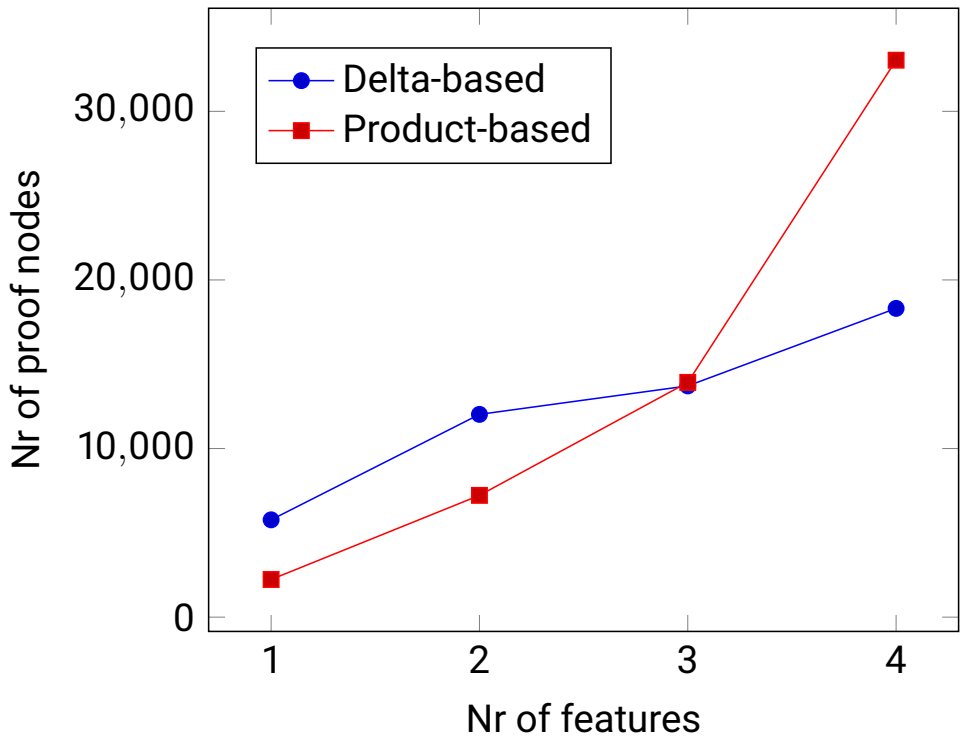
TECHNISCHE UNIVERSITÄT DARMSTADT

Specification and verification of

- **BankAccountSPL** Thüm et al. (2012)
- **MinePumpPL** Apel et al. (2013) Performance?

Performance

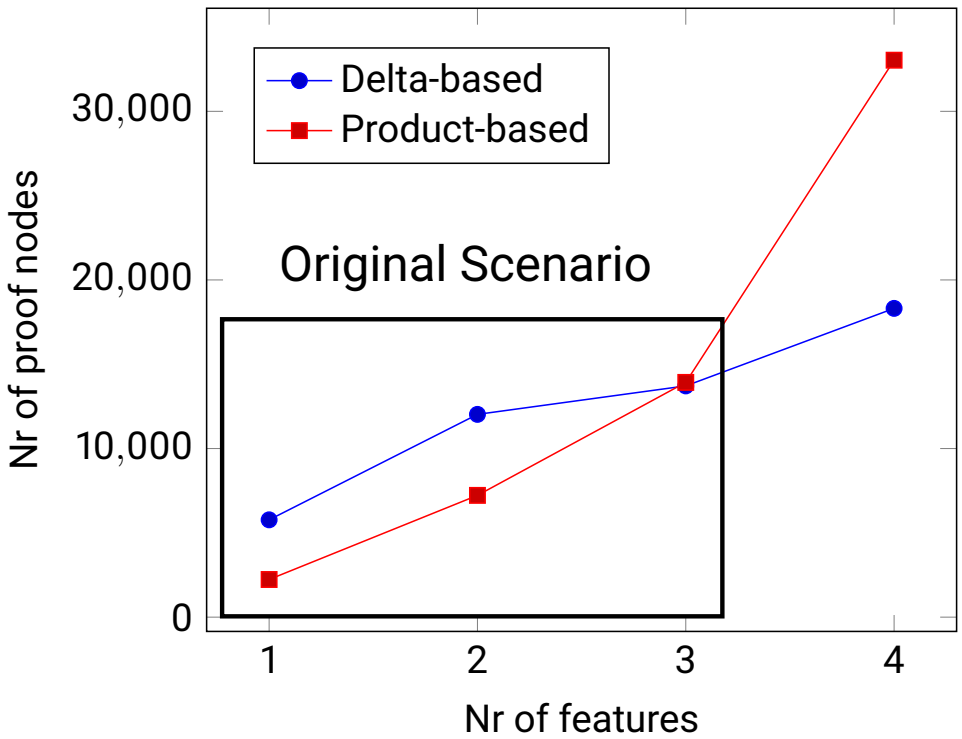
TECHNISCHE UNIVERSITÄT DARMSTADT



For a subset of **MinePumpPL**

Performance

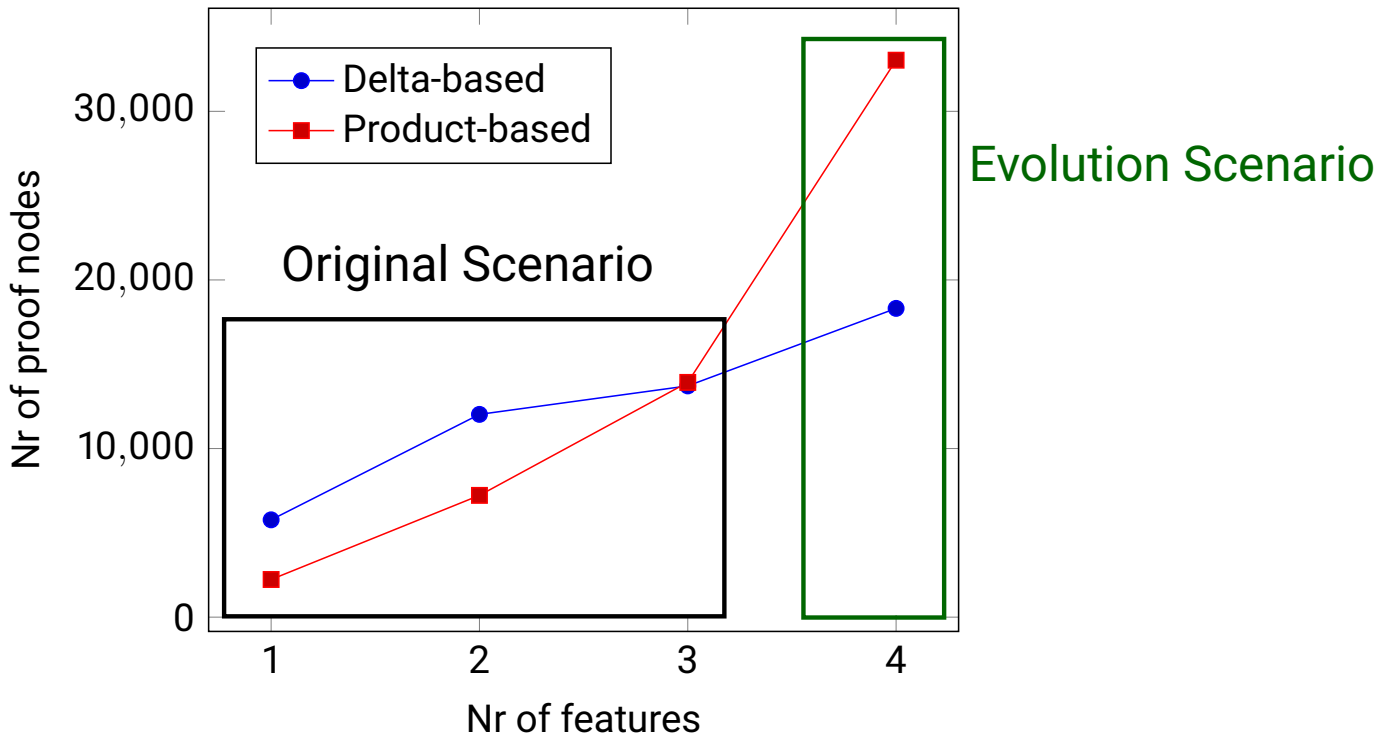
TECHNISCHE UNIVERSITÄT DARMSTADT



For a subset of **MinePumpPL**

Performance

TECHNISCHE UNIVERSITÄT DARMSTADT



For a subset of **MinePumpPL**

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

Achievement

Overcome Liskov's restrictions *without increasing costs*

How?

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

Achievement

Overcome Liskov's restrictions *without increasing costs*

How?

Delta-based Verification:
Normal Form $\Rightarrow$ Abstract Execution

Expressiveness and Performance?

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

Achievement

Overcome Liskov's restrictions *without increasing costs*

How?

Delta-based Verification:
Normal Form $\Rightarrow$ **Abstract Execution**

Expressiveness and Performance?

$\Rightarrow$ **Expressive** enough to specify existing product lines
$\Rightarrow$ **Outperforms** product-based analysis, **suitable for evolving SPLs**

References I

TECHNISCHE UNIVERSITÄT DARMSTADT

S. Apel, A. v. Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. Intl. Conf. on Software Engineering*, ICSE '13, page 482–491, Los Alamitos, CA, 2013. IEEE Press. ISBN 9781467330763. doi: 10.1109/ICSE.2013.6606594.

R. Hähnle and I. Schaefer. A liskov principle for delta-oriented programming. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *LNCS*, pages 32–46, Berlin, Heidelberg, 2012. Springer.

D. Steinhöfel and R. Hähnle. Abstract execution. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *Formal Methods – The Next 30 Years*, volume 11800 of *LNCS*, pages 319–336, Cham, 2019. Springer.

References II

T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. *SIGPLAN Not.*, 48(3):11–20, Sept. 2012. ISSN 0362-1340. doi: 10.1145/2480361.2371404. URL https://doi.org/10.1145/2480361.2371404.

Achieving Normal Form with AE

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  //before refactoring
2  modifies Unit m(p) {
3
4     if ( p1 == 0 ) {
5        field1 = 1;
6        m1(p1); //BP−method
7        original(p);
8        field2 +=1;
9     } else {
10       field1 = 2;
11       original(p);
12       m2(p2); //BP−method
13       field2 +=2;
14    }
15 }
```

⇝

```
1  //after refactoring
2  modifies Unit m(p) {
3     Bool cond = p1 == 0;
4     if ( cond ) {
5        field1 = 1;
6        m1(p1); //BP−method
7     } else {
8        field1 = 2;
9     }
10    original(p);
11    if (cond) {
12       field2 +=1;
13    } else {
14       m2(p2); //BP−method
15       field2 +=2;
16    }
17 }
```

Achieving Normal Form with AE (cont'd)

TECHNISCHE UNIVERSITÄT DARMSTADT

Transformation rules:

1. Conditional Statement Condition Extraction (CSCE)
2. Unfold If Branch Prefix
3. Unfold Else Branch Prefix
4. Consolidate Duplicate Conditional Fragments (Extract Prefix)
5. Split Conditional Statement

```
if ( e^boolean(frameE :≈ footprintE) ) {
    Q1(frameQ1 :≈ footprintQ1);
} else {
    Q2(frameQ2 :≈ footprintQ2);
}
```

CSCE ⟶

```
/*@ ae_constraint
    @ x ∉ frameQ1 && x ∉ footprintQ1 &&
    @ x ∉ frameQ2 && x ∉ footprintQ2;
    @*/
x = e^boolean(frameE :≈ footprintE);
if (x) {
    Q1(frameQ1 :≈ footprintQ1);
} else {
    Q2(frameQ2 :≈ footprintQ2);
}
```