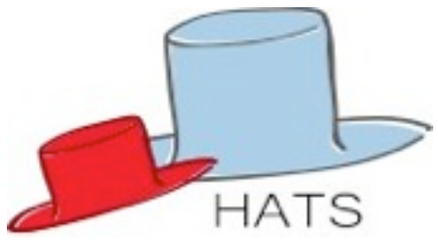
Reuse-based Verification of Software Product Lines

Ina Schaefer
Chalmers University
schaefer@chalmers.se



8th IFIP WG 2.11 Meeting
St. Andrews, U.K.
1-3 March 2010

HATS Project

HATS: Highly Adaptable & Trustworthy Software Using Formal Models

Proposal Data

- FP7 FET focused call Forever Yours
- Project started 1 March 2009, 48months runtime
- Integrated Project, academically driven
- 8 academic partners, 2 industrial research, 1 SME
- 7 countries
- 730 PM, EC contribution 5,27 M€ over 48 months
- Associated with FP7 Coordination Action: EternalS
  - Trustworthy Eternal Systems via Evolving Software, Data and Knowledge

HATS Consortium

Hähnle, Bubel	Chalmers Tekniska Högskola (Coordinator)	SE
Johnsen, Steffen	Universitetet i Oslo	NO
Dam, Gurov	Kungliga Tekniska Högskolan	SE
Puebla, Barthe	Universidad Politécnica de Madrid / IMDEA	ES
Poetzsch-Heffter	University of Kaiserslautern	DE
Sangiorgi, Lanese	Università di Bologna	IT
De Boer	Centrum voor Wiskunde en Informatica	NE
Østvold	Norsk Regnesentral	NO
Diakov	Fredhopper	NE
Carbon	Fraunhofer IESE	DE
Clarke, Piessens	Katholieke Universiteit Leuven	BE

HATS Approach

A tool-supported formal method for
building highly adaptable and trustworthy software

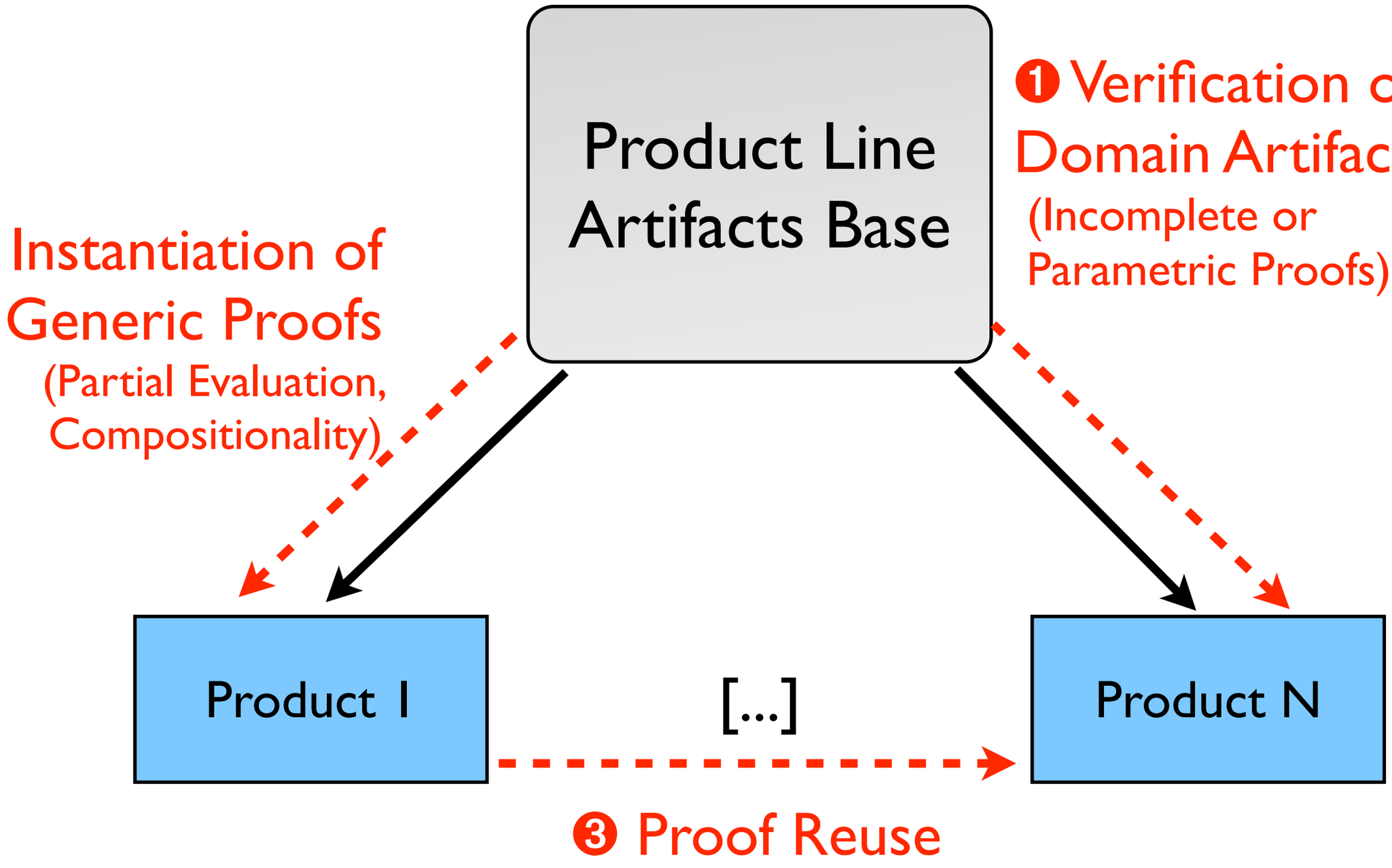
Ingredients

1. Abstract Behavioral Specification (ABS) language: Executable modeling language for adaptable software
2. Integrated framework and tool architecture around ABS
3. Tool suite for development and analysis:
   e.g., feature consistency, type checking, property verification, code generation, test case generation, specification mining

Verification of SPL

- Essential to ensure correctness of products because of high configurative variability.
- Formal verification by theorem proving and model checking.
- But, it is not feasible to verify each product in isolation.

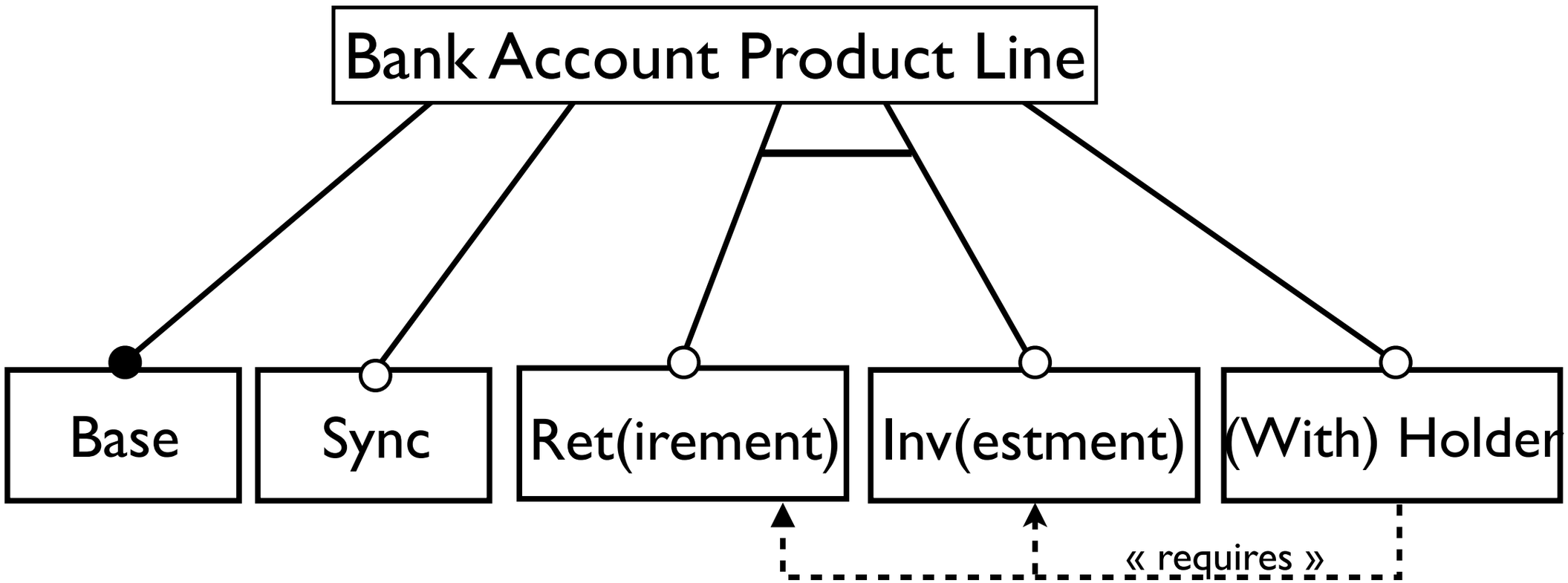
Reuse in Verification



Outline

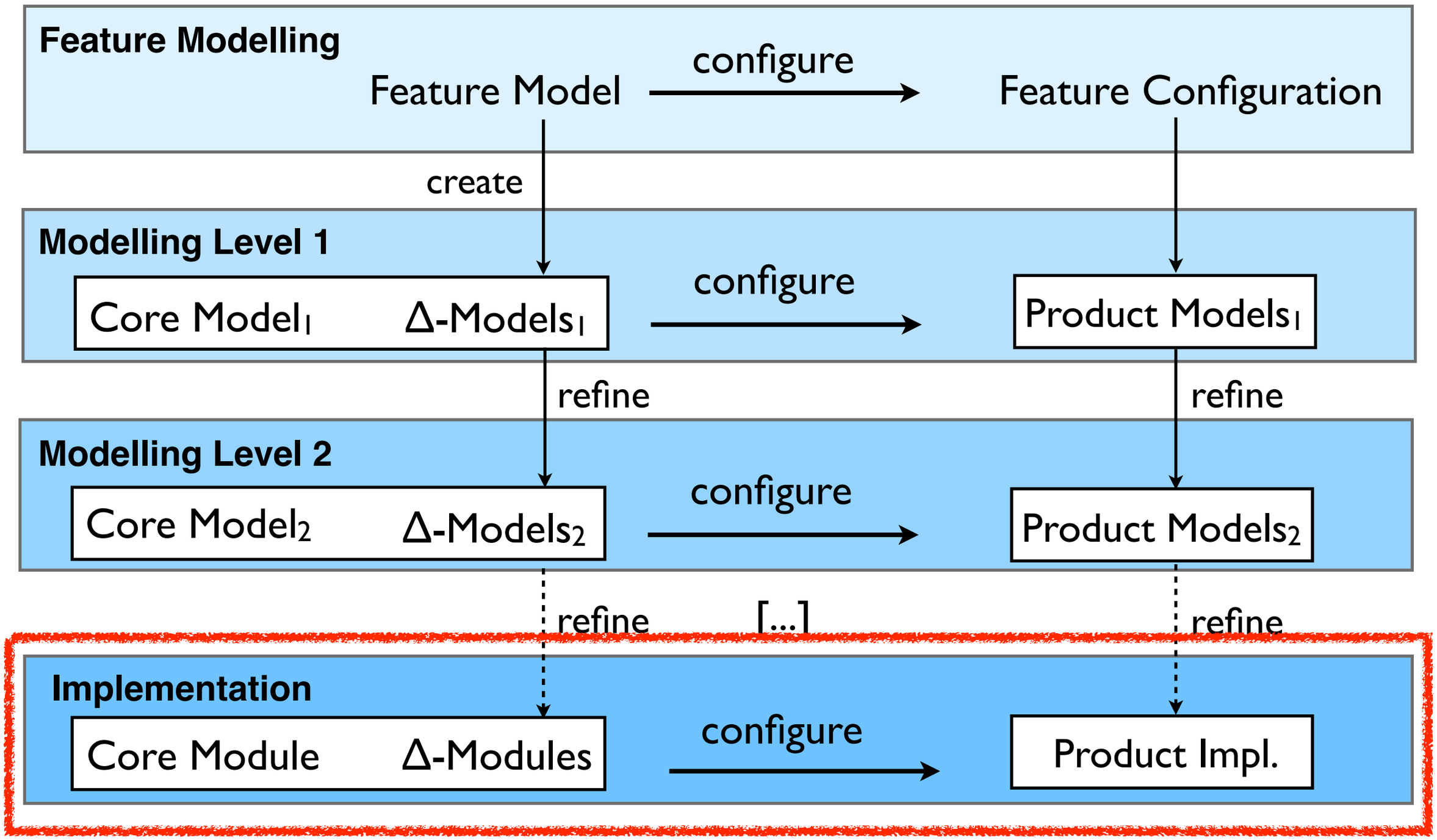
- Model-based Software Product Line Engineering
- Implementing SPL with FΔJ
- Proof Reuse for Verification of FΔJ SPL

Feature Model



Example taken from [Batory at. al., FOAL09]

Model-based Development



I. Schaefer: Variability Modelling for Model-driven Development of Software Product Lines. Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), Linz, January 2010.

FΔJ - A PL for SPL

- Extension of Java with Core and Δ-Modules
- Core Product is implemented by Core Module.
- Product-Δs are implemented by Δ-Modules.
- A Product Implementation is obtained by Application of Δ-Modules to Core Module.
- Type System ensures Safety of Δ-application.

I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella: Delta-oriented Programming of Software Product Lines. February 2010 (submitted)

Core Module

A core module contains a set of Java classes.

```
core BaseAccount {
        class Account extends Object {
                int balance;
                void update(int x) { balance += x; }
        }
}
```

Δ-Modules

- Modifications on Class Level:
  - Addition, Removal and Modification of Classes
- Modifications of internal Class Structure:
  - Adding, Removing, Renaming Fields
  - Adding, Removing, Renaming Methods
- Application Condition in **`when`** clause: Boolean Constraint on Features in Feature Model
- Partial Ordering of Δ-Modules by **`after`** clauses

Δ-Module for Sync

```
delta DsyncUpdate after Dretirement, Dinvestment when Sync {
        modifies class Account {
                adds Lock lock;
                renames update to unsync_update;
                adds void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
        }
}
```

Δ-Application

Core Module

```
core BaseAccount {
    class Account extends Object {
        int balance;
        void update(int x) { balance += x; }
    }
}
```

Δ-Module

```
delta DsyncUpdate after Dretirement, Dinvestment when Sync {
    modifies class Account {
        adds Lock lock;
        renames update to unsync_update;
        adds void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
    }
}
```

Product

```
class Account extends Object {
    int balance;
    Lock lock;
    void unsync_update(int x) { balance += x;  }
    void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
}
```

Type System for FΔJ

- The core and Δ-modules can be typed in isolation.
- If a core module and a set of Δ-modules are type correct, Δ-application is safe:
  - removed and modified classes exists and added classes do not exist
  - all renamed/removed fields and methods exist
  - all added fields and methods do not exist
  - there are not conflicting modifications that are not ordered by `after` clauses

Verification of FΔJ SPL

- We use the KeY System for deductive verification of FΔJ SPL.
- Input to KeY:
  - Java Program (generated from FΔJ SPL) with JML Specifications
- KeY generates proof obligations in dynamic logic and supports automatic and interactive verification.

Specification of Base Account

We want to prove that the balance of an account is always positive.

```
/*@
 @ public instance invariant balance >= 0;
 @*/

public class BaseAccount {

    int balance;

    /*@
     @ ensures \result.balance==0;
     @*/
    public BaseAccount(){
    balance = 0;
    }

    /*@
     @ public normal_behavior
     @ requires x > 0;
     @ assignable \everything;
     @ ensures balance >= \old(balance);
     @*/
    public void update(int x){
          balance = balance + x;
    }

}
```

← Instance Invariant

← Method Contract

Specification of SyncAccount

We want to prove that the balance of a synchronized account is always positive.

```
/*@
 @ public instance invariant balance >= 0;
 @*/

public class SyncAccount {

    int balance;
    Lock lock;


/*@
 @ ensures \result.balance==0;
 @*/
public SyncAccount(){
balance = 0;
lock = new Lock();
}
```

```
/*@
    @ public normal_behavior
    @ requires x > 0;
    @ assignable \everything;
    @ ensures balance >= \old(balance);
     @*/
    public void unsync_update(int x){
          balance = balance + x;
    }

    /*@
     @ public normal_behavior
     @ requires x > 0;
     @ assignable \everything;
     @ ensures balance >= \old(balance);
     @*/
    public void update(int x){
    lock.lock(); unsync_update(x); lock.unlock();
          }

}
```

Comparison

```
public class BaseAccount {

[...]

 /*@
  @ public normal_behavior
  @ requires x > 0;
  @ assignable \everything;
  @ ensures balance >= \old(balance);
  @*/
 public void update(int x){
        balance = balance + x;
 }

}
```

```
public class SyncAccount {

[...]

 /*@
      @ public normal_behavior
      @ requires x > 0;
      @ assignable \everything;
      @ ensures balance >= \old(balance);
      @*/
     public void unsync_update(int x){
            balance = balance + x;
     }

 /*@
  @ public normal_behavior
  @ requires x > 0;
  @ assignable \everything;
  @ ensures balance >= \old(balance);
  @*/
 public void update(int x){
 lock.lock(); unsync_update(x); lock.unlock();
        }

}
```

Method Renaming

→ Proof Reuse for Method Contract

More Proof Reuse

```
public class BaseAccount {

int balance;

[...]

  /*@
   @ public normal_behavior
   @ requires x > 0;
   @ assignable \everything;
   @ ensures balance >= \old(balance);
   @*/
  public void update(int x){
        balance = balance + x;
  }

}
```

Field Renaming

```
public class RetAccount {

  int bbalance;

[...]

  /*@
   @ public normal_behavior
   @ requires x > 0;
   @ assignable \everything;
   @ ensures bbalance >= \old(bbalance);
   @*/
  public void update(int x){
        bbalance = bbalance + x;
  }
```

Method Renaming

```
  /*@
   @ public normal_behavior
   @ requires x > 0;
   @ assignable \everything;
   @ ensures bbalance >= \old(bbalance);
   @*/
  public void addBonus(int x){
        bbalance = bbalance + x;
  }
}
```

→ Proof Reuse for Both Method Contracts

Even More Proof Reuse?

```
public class BaseAccount {

[...]

  /*@
   @ public normal_behavior
   @ requires x > 0;
   @ assignable \everything;
   @ ensures balance >= \old(balance);
   @*/
  public void update(int x){
        balance = balance + x;
  }

}
```

Wrapping of Original Method Call

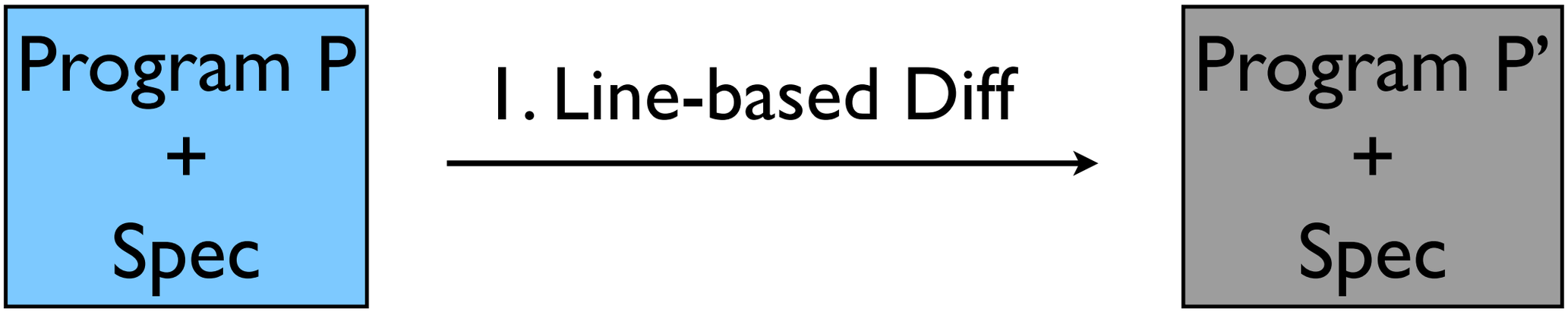
```
public class SyncAccount {

[...]

/*@
     @ public normal_behavior
     @ requires x > 0;
     @ assignable \everything;
     @ ensures balance >= \old(balance);
     @*/
    public void unsync_update(int x){
          balance = balance + x;
    }

     /*@
      @ public normal_behavior
      @ requires x > 0;
      @ assignable \everything;
      @ ensures balance >= \old(balance);
      @*/
     public void update(int x){
     lock.lock(); unsync_update(x); lock.unlock();
           }

}
```

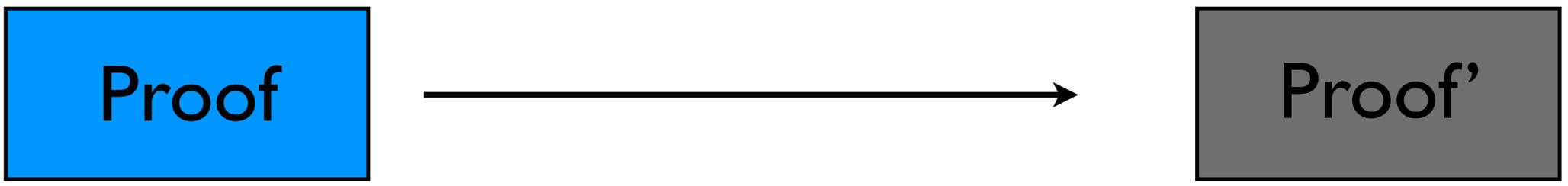
→ Partial Proof Reuse:
New Proof Steps for Wrapping

Observations

- Verified 17 method contracts in 6 variants of Bank account SPL.
- Only 3 contracts have to be proven from scratch.
- General Approach to Proof Reuse for Verification of Delta-oriented SPL?

Proof Reuse in KeY





2. Mark proof steps for reuse

3. Determine new proof steps by heuristics

Δ-oriented Proof Reuse

Program P
+
Spec

Δ-Information
(Program-Δs + Spec-Δs)

Program P'
+
Spec'

1. Determine unchanged Specs by Δ-based Slicing

Proof

Proof'

2. Mark proof steps for reuse using Δ-Information

3. Determine new proof steps by heuristics using Δ-Information

with Bernhard Beckert and Vladmir Klevabov (Karlsruhe Institute of Technology)

Conclusion

- Model-driven Software Product Line Engineering based on Δ-Modelling
- Implementing SPL with FΔJ
- Proof Reuse for Efficient Verification of FΔJ SPLs

Future Work

- Modular Δ-based Slicing Techniques
- Proof Reuse for Specification-Δs
- Integration of Proof Reuse for SPL into KeY System
- Compositional Verification of Δ-Models