Efficient Verification of Software Product Lines

Ina Schaefer
Chalmers University
schaefer@chalmers.se

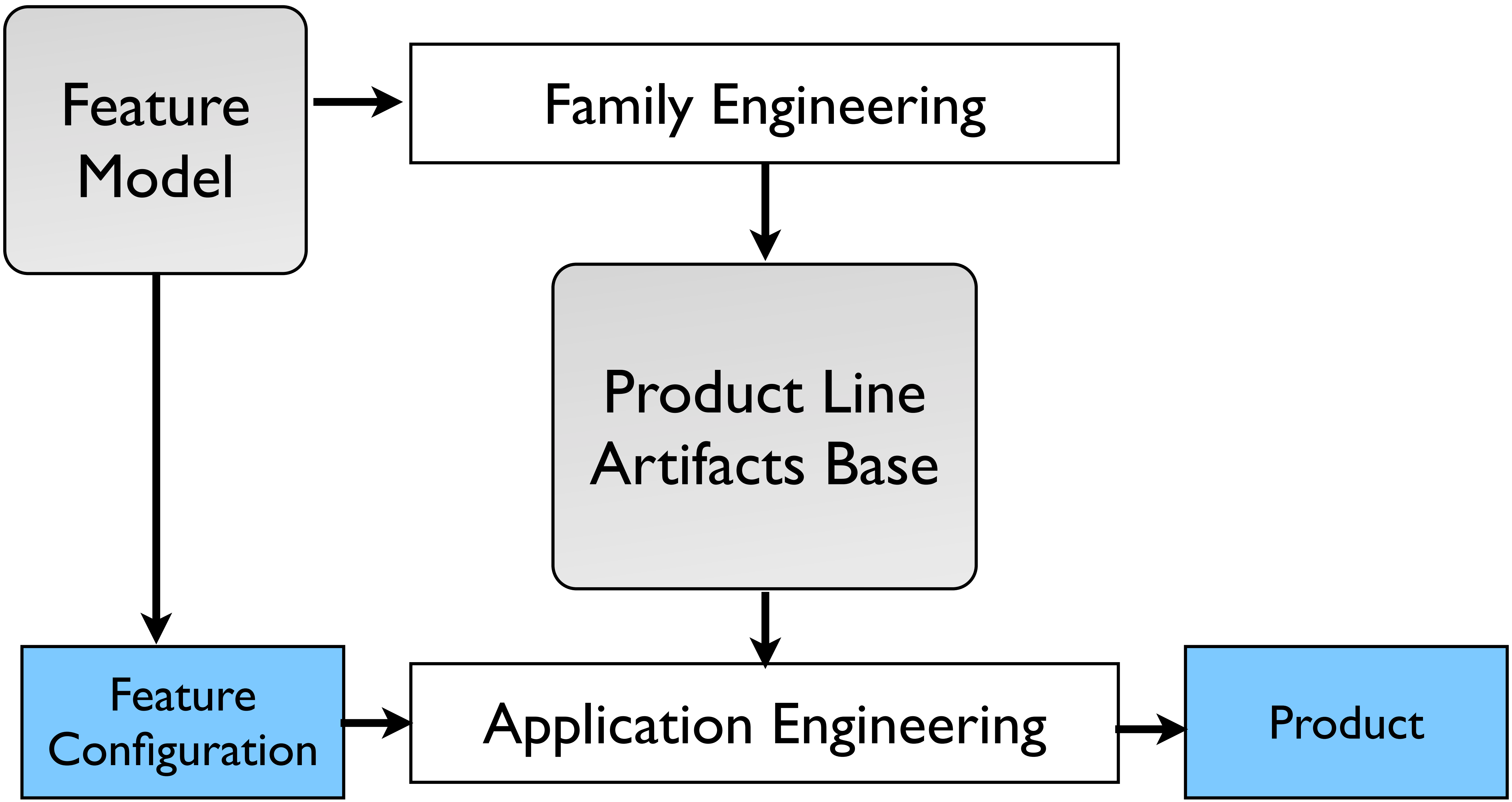
Guest Lecture
Software Product Line Engineering
8 December 2009

Software Product Lines

- A product line is "a family of products designed to take advantage of their common aspects and predicted variabilities."[Weiss; 1999]

- A product line is "a set of systems sharing a common set of features that satisfy the specific needs of a particular market segment." [Clements, Northrop; 2001]

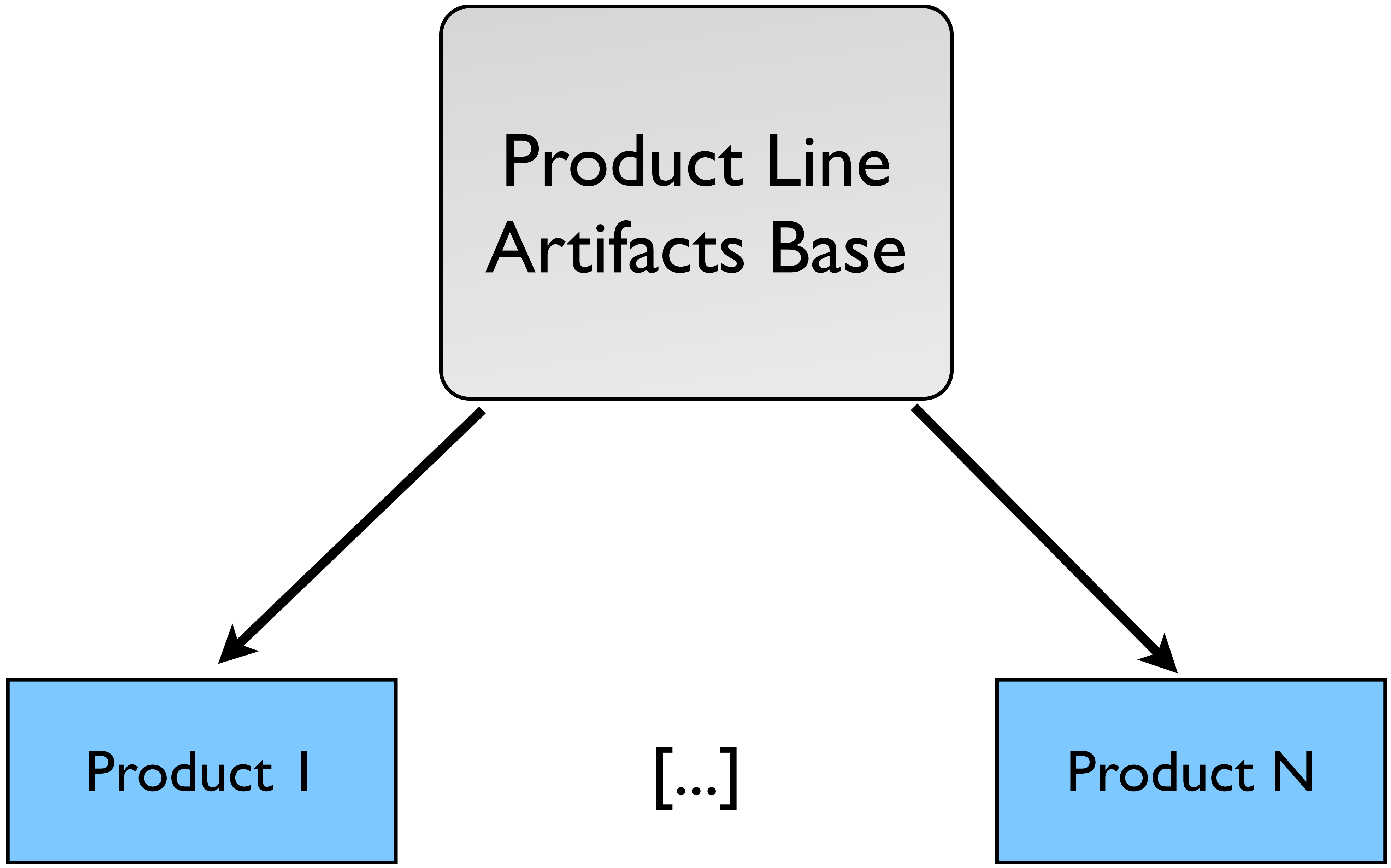
Product Line Development



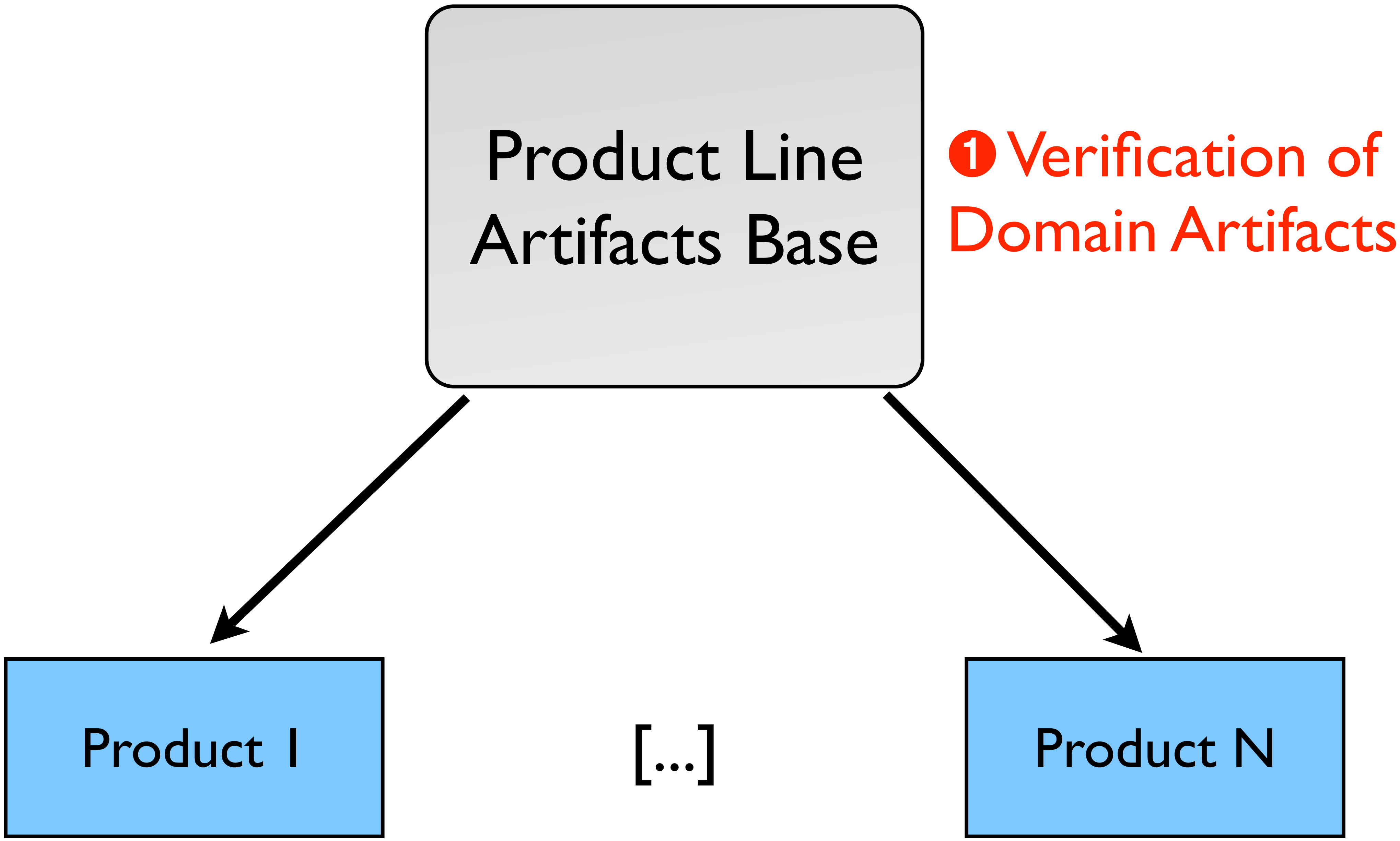
Verification of SPL

- High configurative variability of products
- Correctness of products is crucial.
- Formal verification by theorem proving and model checking can establish critical product properties.
- But, it is not feasible to verify each product in isolation.

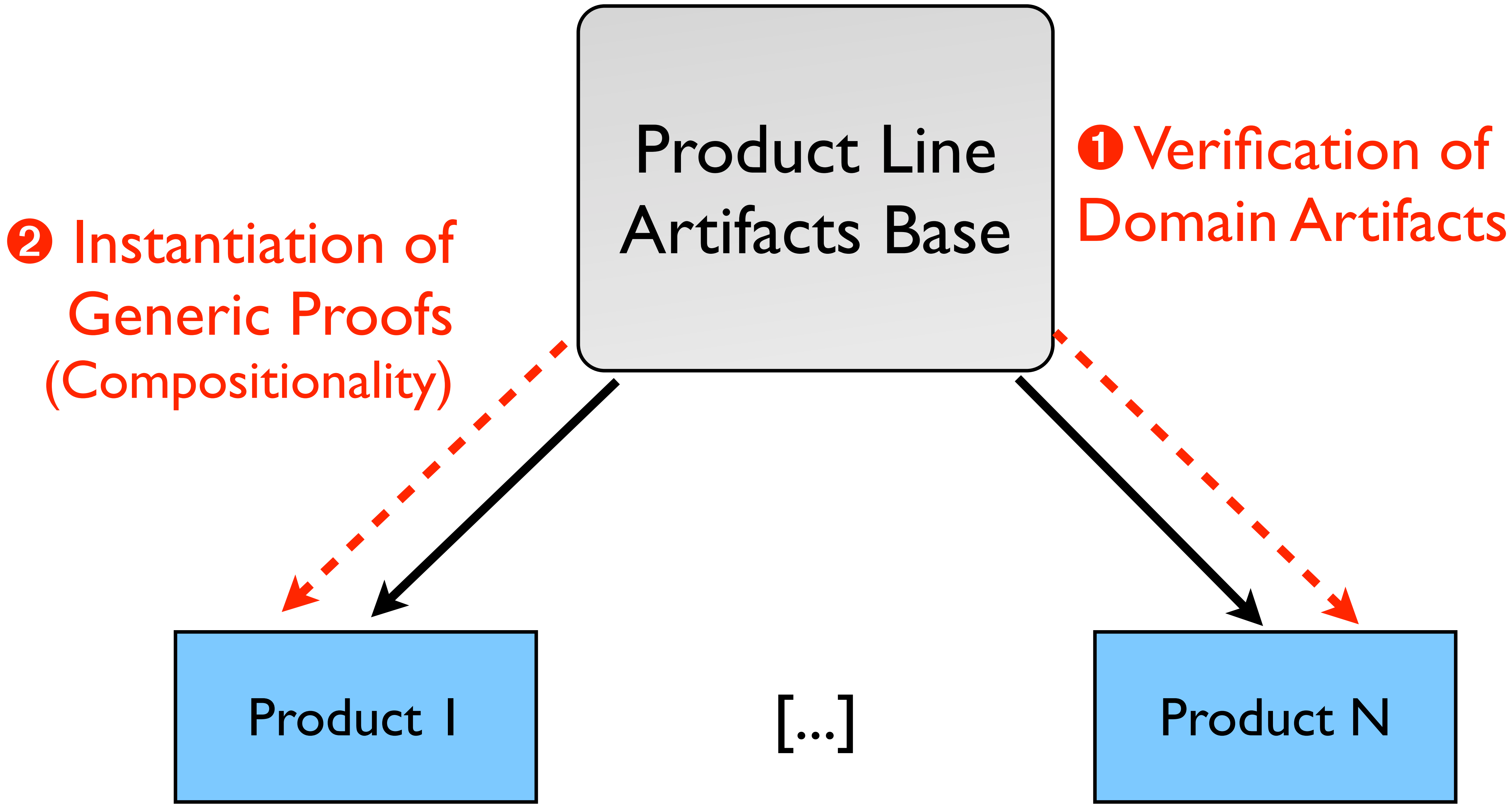
Reuse in Verification



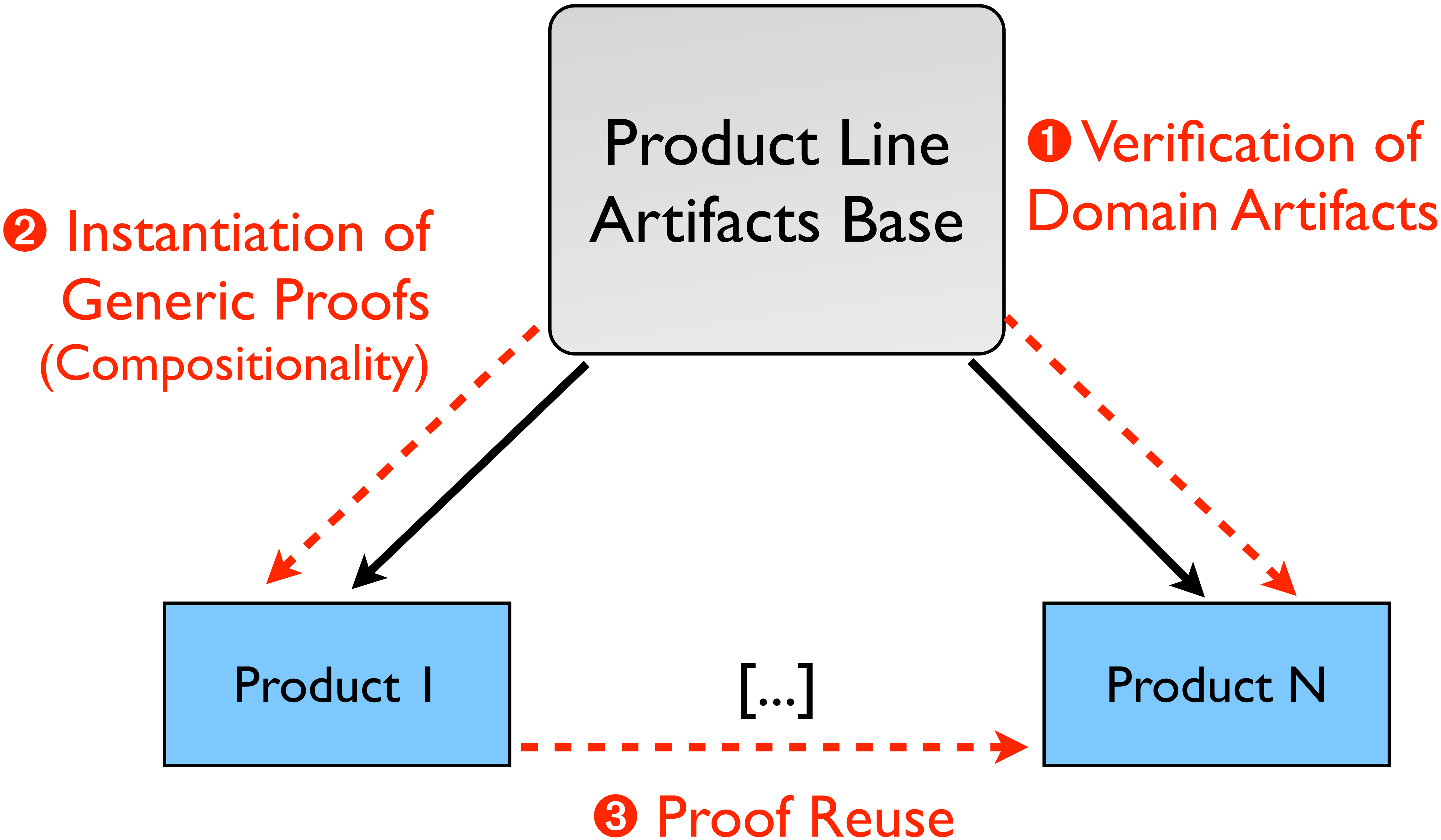
Reuse in Verification



Reuse in Verification



Reuse in Verification



Outline

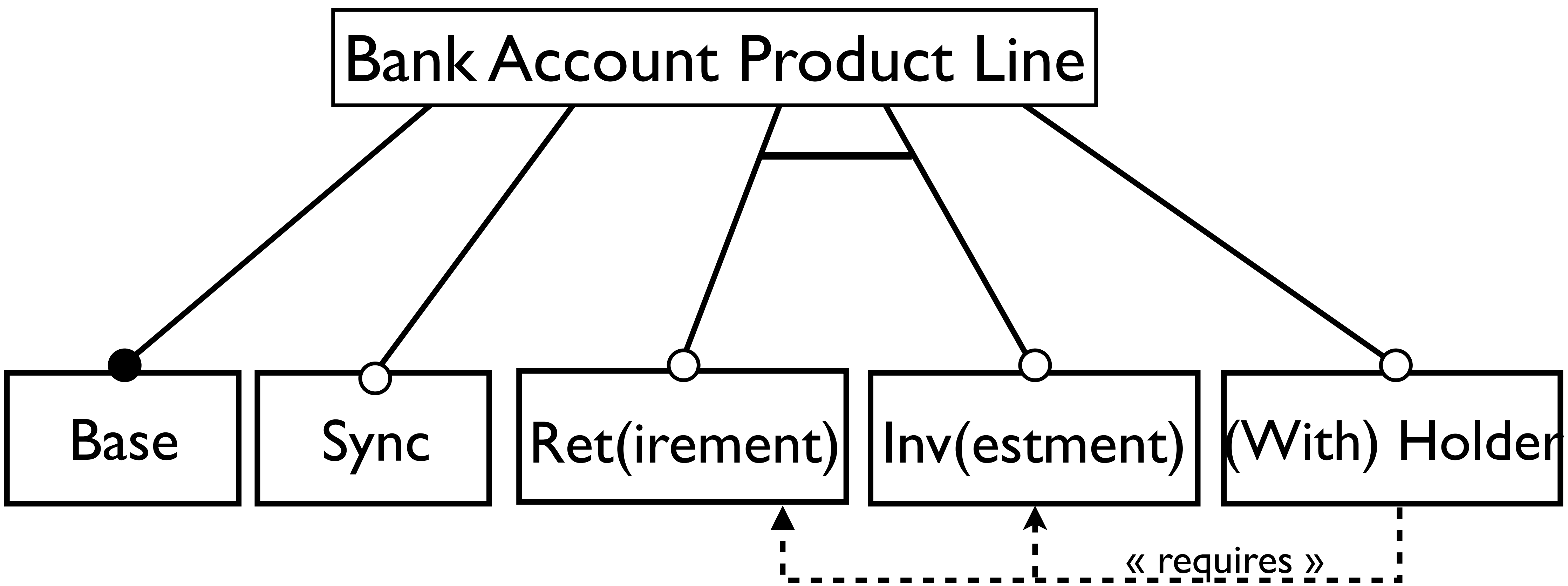
- Model-based Software Product Line Engineering
- Variability Modelling using Δs
- Implementing SPL with FΔJ
- Proof Reuse for Verification of FΔJs

Product Map

Example from [Batory et al., FOAL09]:

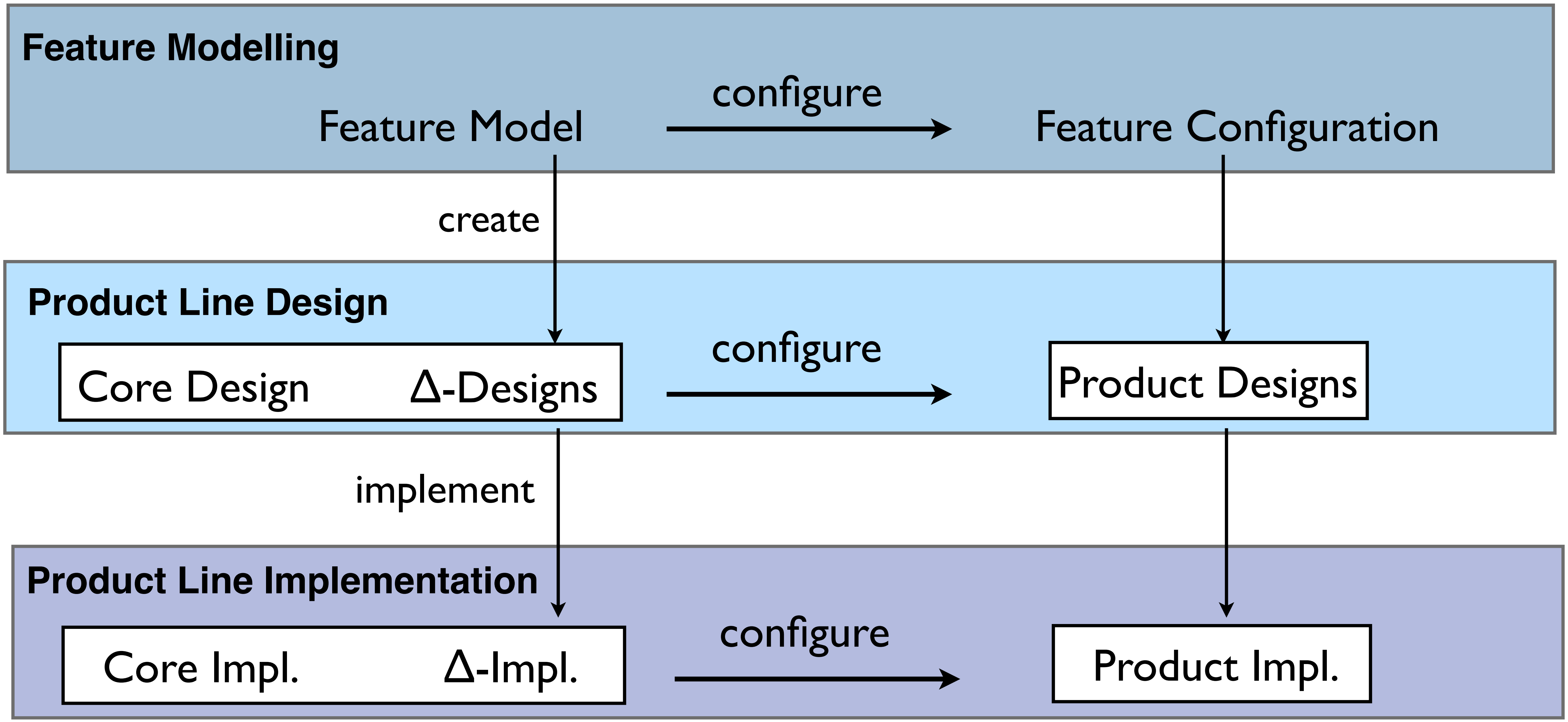
	Base	Sync	Ret	Inv	WHolder
BaseAccount	x				
SyncAccount	x	x			
AccWHolder	x				x
RetAccount	x		x		
SyncAccWH	x	x			x

Feature Model

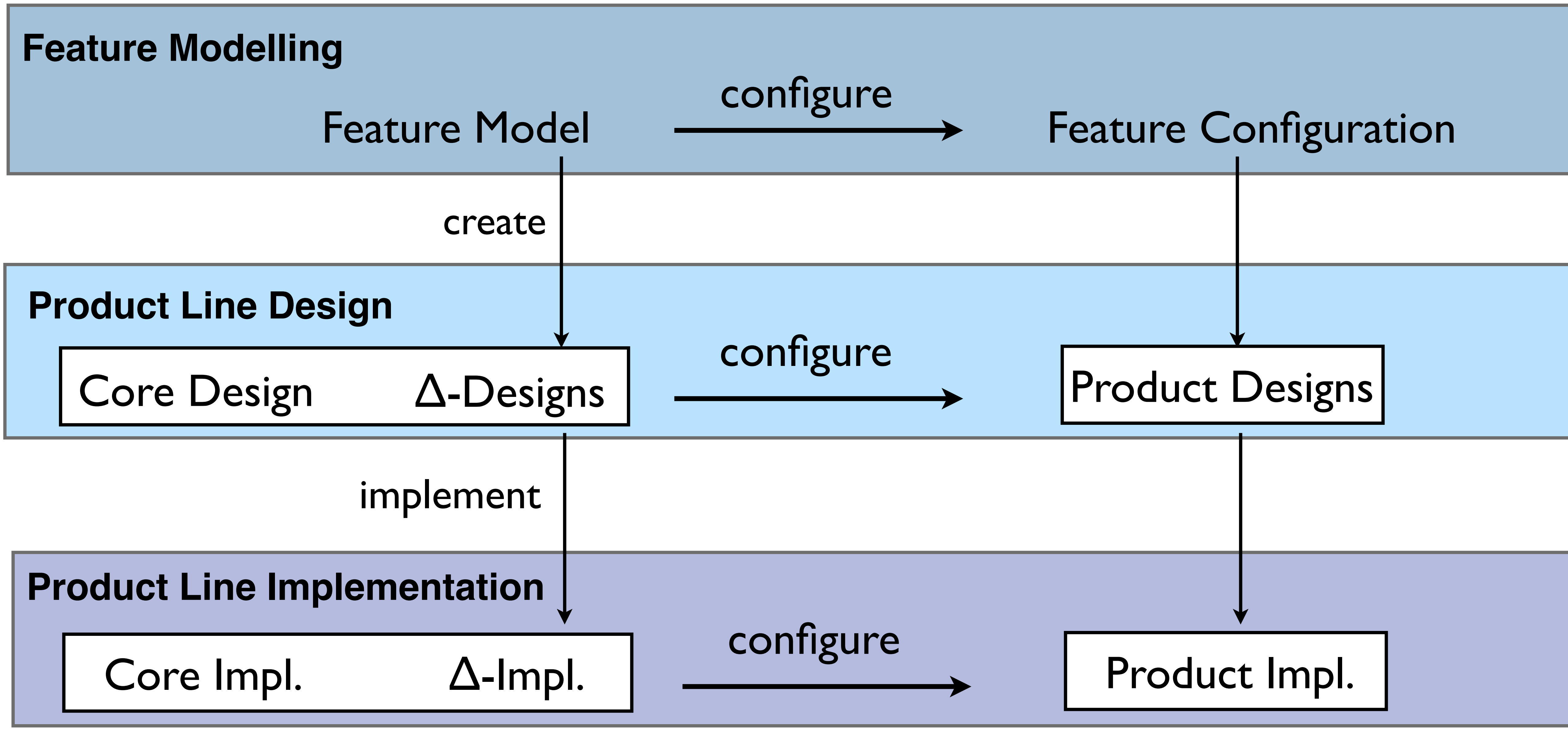


Master Thesis: Comparison between Product Maps and Feature Models

Model-based Development

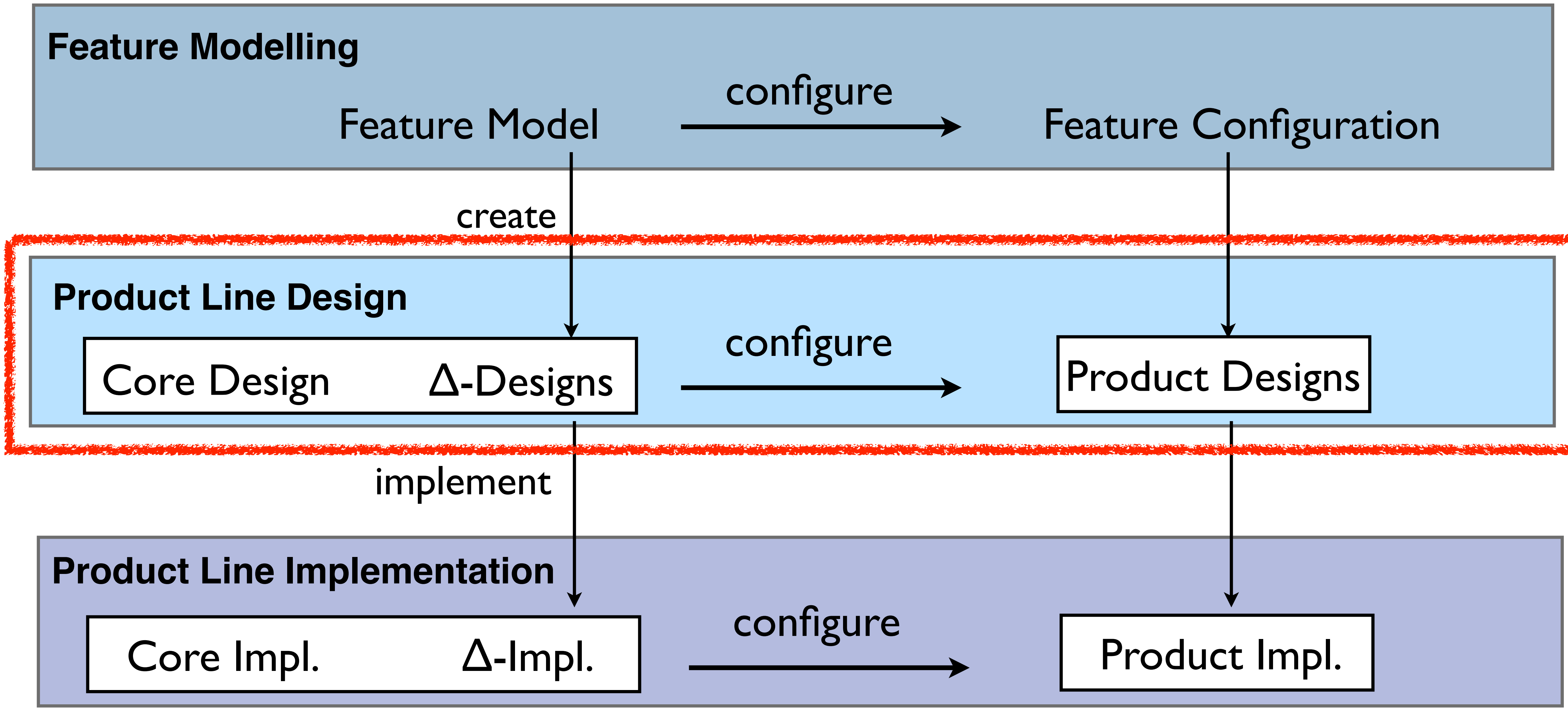


Model-based Development



I. Schaefer, A. Worret, A. Poetzsch-Heffter: A Model-based Framework for Automated Product Derivation.
Workshop on Model-driven Approaches to Product Line Engineering (MAPLE), August 2009

Model-based Development



I. Schaefer, A. Worret, A. Poetzsch-Heffter: A Model-based Framework for Automated Product Derivation.
Workshop on Model-driven Approaches to Product Line Engineering (MAPLE), August 2009

Δ-Modelling

A product line is represented by a core product and a set of product-Δs.

The core product is a complete product for a valid feature configuration. It is not uniquely determined.

Guideline: The core product contains

- all mandatory features.
- a minimal number of required alternative features

Master Thesis: Impact of Core Product to SPL Design and Implementation

Δ-Modelling (2)

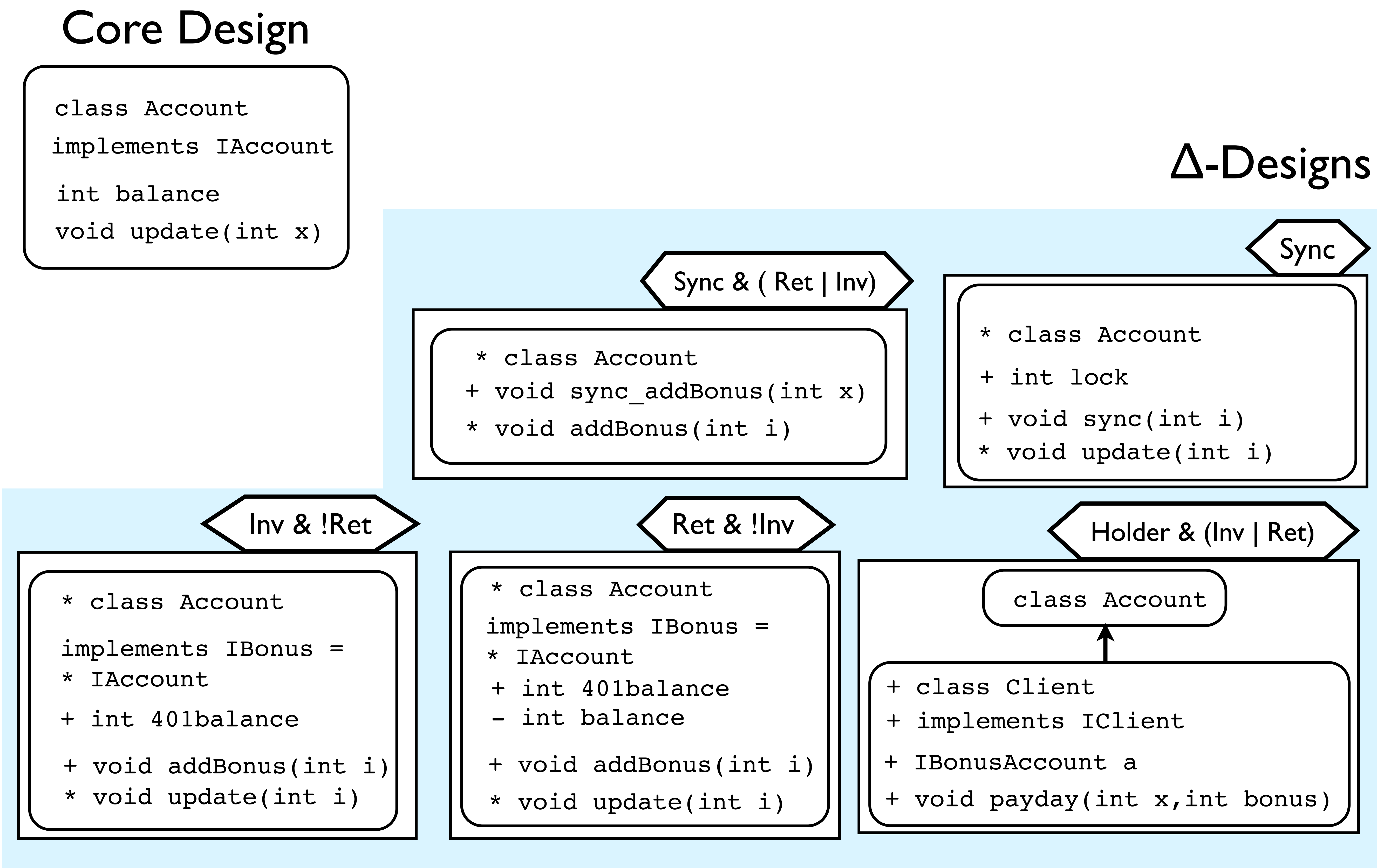
- Product-Δs specify Additions, Modifications, Removals to the Core Product.
- Application constraint over the features in the feature model determines for which feature configuration the product-Δ has to be applied to the core product.
- Product-Δs can be partially ordered to avoid conflicting changes of the core product.

Configuration

For a Feature Configuration:

- Determine product-Δs with valid application condition.
- Determine linear ordering of product-Δs compatible with partial ordering.
- Apply changes specified by product-Δs to core product in the linear order.

Variability in System Design



Configuring Designs

Product: Base & Sync

Applicable Δs

Sync

```
* class Account
+ int lock
+ void sync(int i)
* void update(int i)
```

Core Design

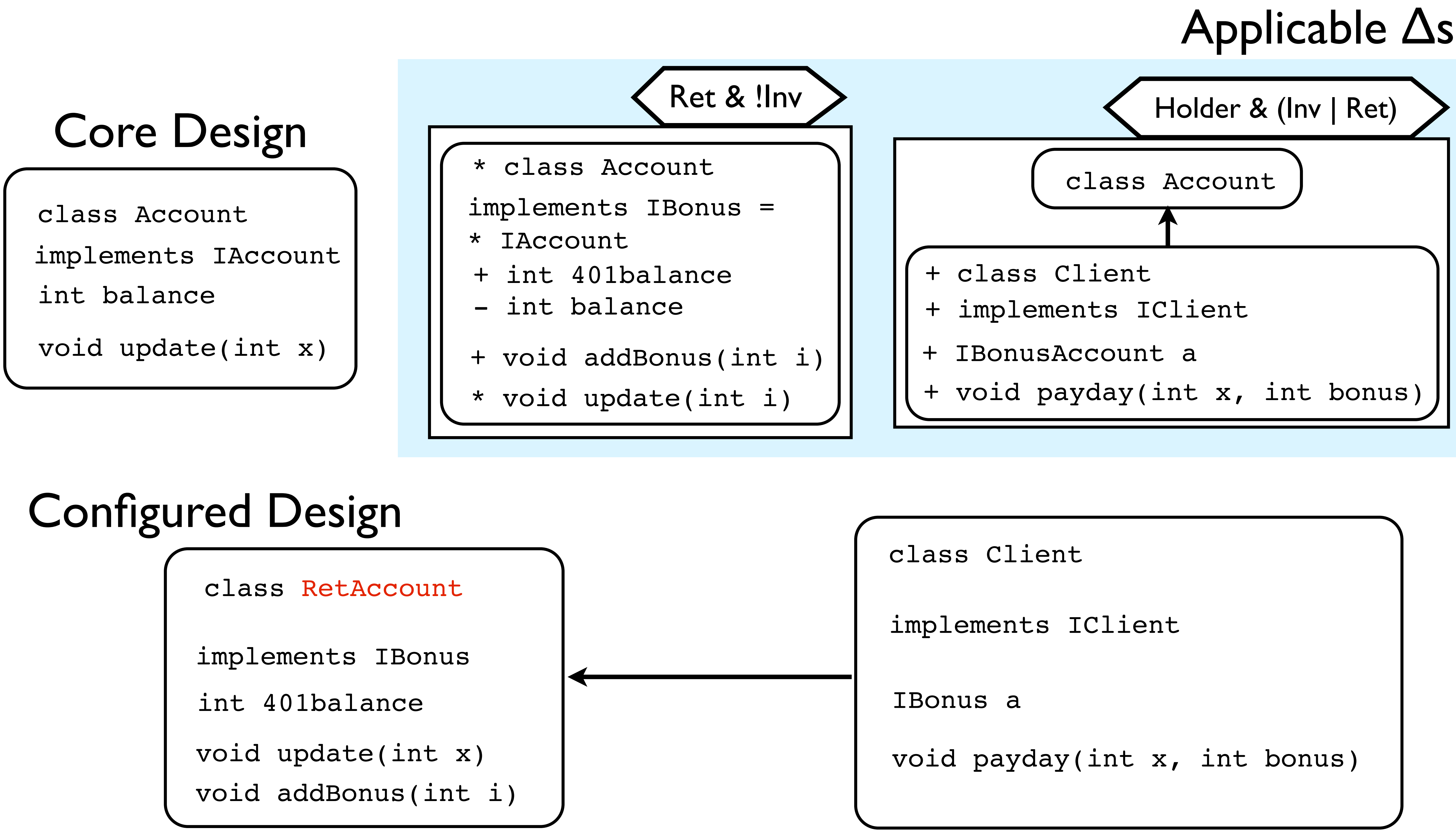
```
class Account
implements IAccount
int balance
void update(int x)
```

Configured Design

```
class SyncAccount
implements IAccount
int lock
int balance
void update(int x)
void sync(int i)
```

Configuring Designs

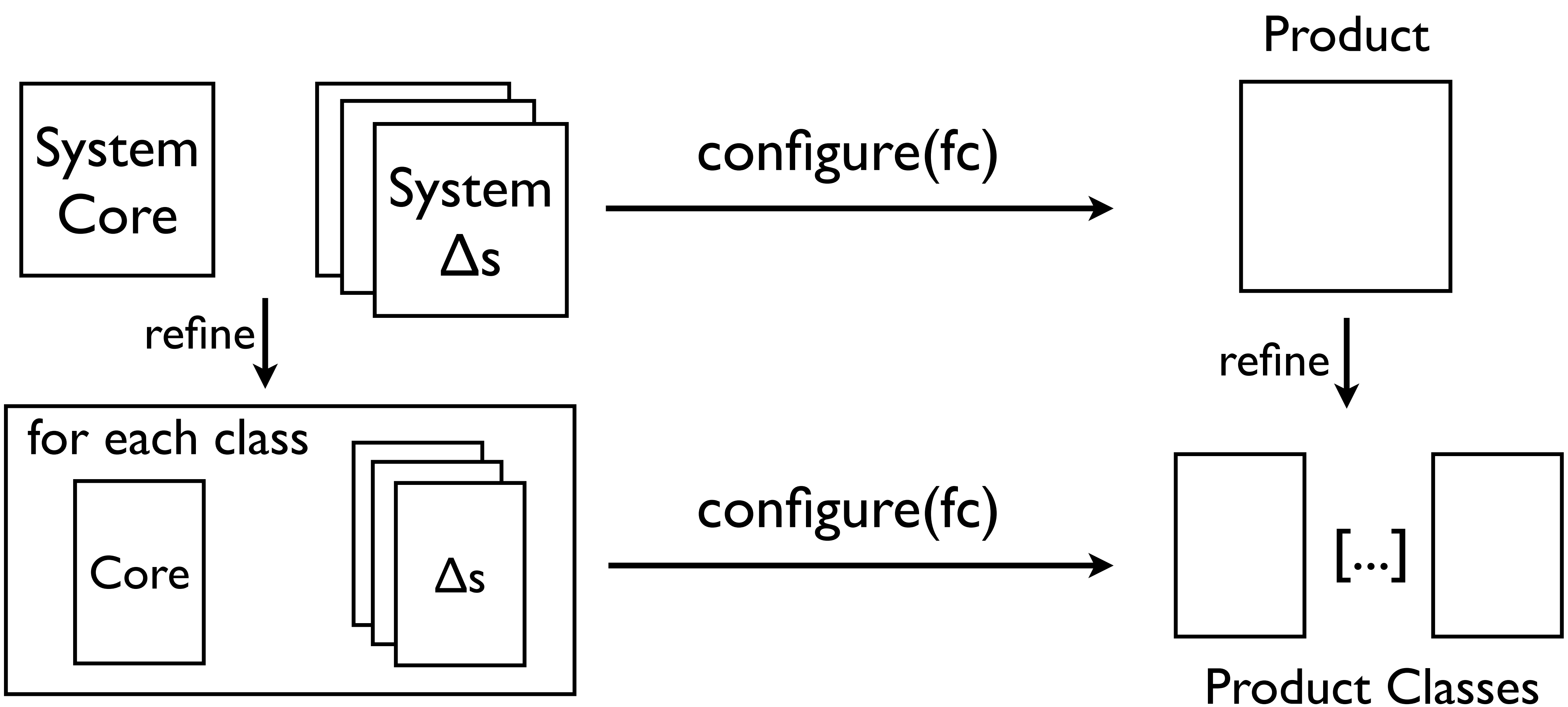
Product: Base & Ret & With Holder



Variability Modelling with Δs

- Evolutionary Development by Adding Product-Δs
- Explicit Treatment of Combinations of Features by Complex Application Conditions
- Usable with Different Modelling Formalisms and Implementation Techniques
- Model Refinements are Orthogonal to Variability Modelling.

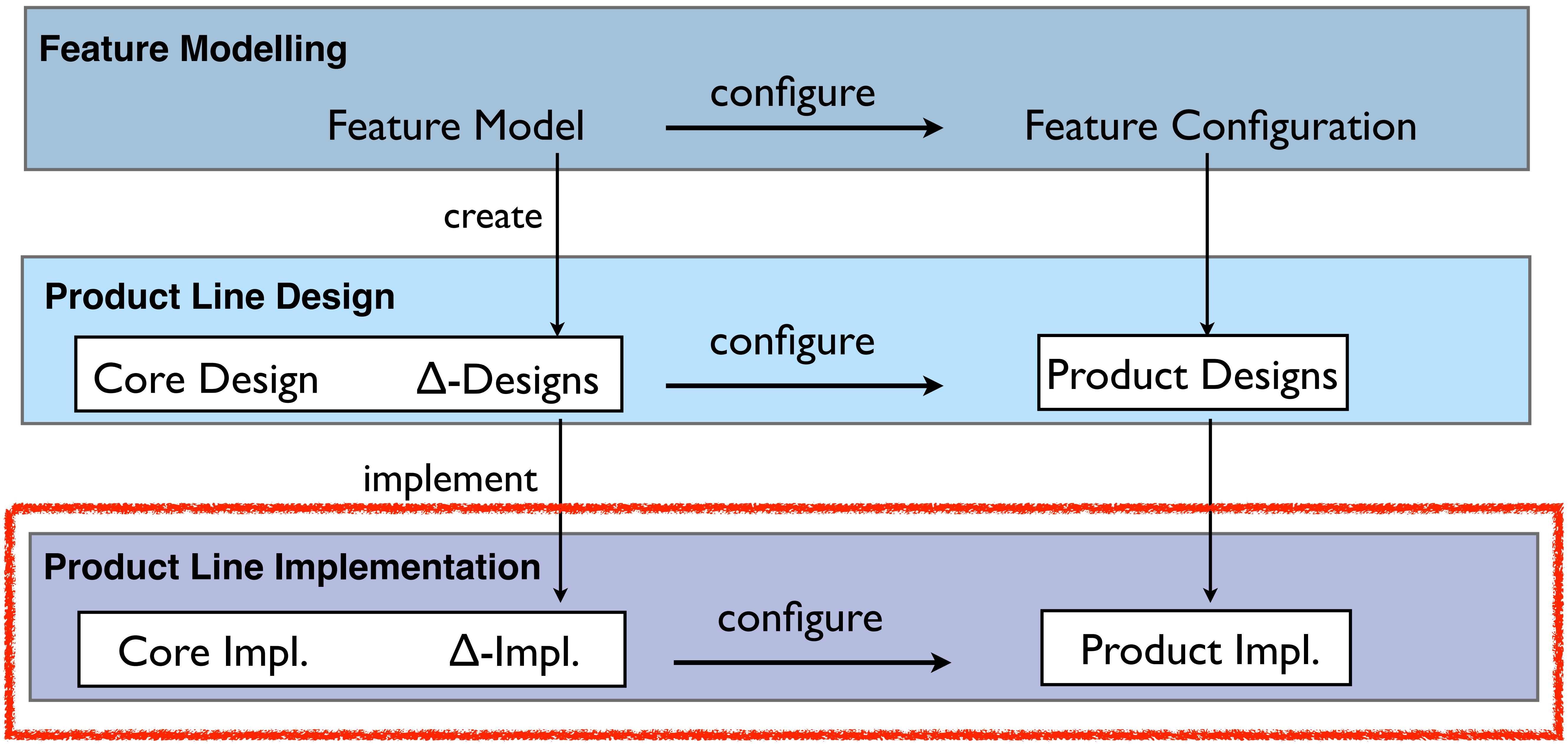
Model Refinement



It holds that:
refine(configure((Core, Δs),fc)) = configure(refine(Core,Δs),fc)

Master Thesis: Case Study and Tool Support for MDD with Δs

Model-based Development



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.

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.

L. Bettini, V. Bono, F. Damiani, I. Schaefer: A Programming Language for Software Product Lines.
Draft, December 2009

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(); }
}
```

Δ-Application

Δ- Module

```
delta DwithHolder when WithHolder {
    adds class Client {
        Account a;
        void payday(int x, int bonus) { a.addBonus(bonus); a.update(x); }
    }
}
```

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(); }

    class Client {
        Account a;
        void payday(int x, int bonus) { a.addBonus(bonus); a.update(x); }
    }
}
```

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:
  - all renamed/removed fields and methods exist
  - all added fields and methods do not exist
  - removed classes exists and added classes do not exist
  - there are not conflicting modifications in a class

Verification of FΔJ

- We use the KeY System for deductive verification.
- Input: Java Program + JML Specifications
- KeY generates proof obligations in dynamic logic.
- KeY supports interactive and automatic verification of the proof obligations.

Specification of Base Account

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

```
/*@
 @ public instance invariant balance >= 0;      ←——— Instance Invariant
 @*/

public class BaseAccount {

   int balance;

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

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

}
```

Specification of SyncAccount

We want to prove that the balance of an 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();
       }

}
```

Comparison

```
public class BaseAccount {

[...]

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

}
```

Method Renaming

```
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();
        }

}
```

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;
   }

}
```

```
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;
   }

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

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;
 }

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

More Proof Reuse

```java
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

Method Renaming

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

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

Method 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;
 }
 /*@
  @ 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

Observations

- 17 Method Contracts in 6 Variants of the Bank Account SPL verified.
- Only 3 Contracts have to be proven from scratch.
- Δ-Modules imply Specification-Δs.
- Structure of Δ-Modules indicates Proof Reuse Potential
- Proofs can be reused if only fields and methods are renamed, but internal class structure is unchanged.
- More reuse scenarios to be identifed.

<u>Master Thesis:</u> Case Study on Proof Reuse for Example SPL

Conclusion

- Model-based Software Product Line Engineering
- Variability Modelling using Δs
- Implementing SPL with FΔJ
- Proof Reuse for Verification of FΔJs

Master Thesis Proposals

- Comparison between Product Maps and Feature Models
- Impact of the Core Product in Δ-Modelling
- Evaluation and Tool Support for Model-based Development using Δ-Modelling
- Case Study on Proof Reuse for FΔJ