TECHNISCHE
UNIVERSITÄT
DRESDEN

Well-Formed and Scalable Invasive Software Composition

Dissertation presentation and defense
Dipl.-Inf. Sven Karol
1st reviewer: Prof. Dr. Uwe Aßmann
2nd reviewer: Prof. Dr. Welf Löwe

Dresden, 18.05.2015

DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

Outline

Part I: Overview

- Invasive Software Composition
- Problem Analysis
- Thesis Contributions

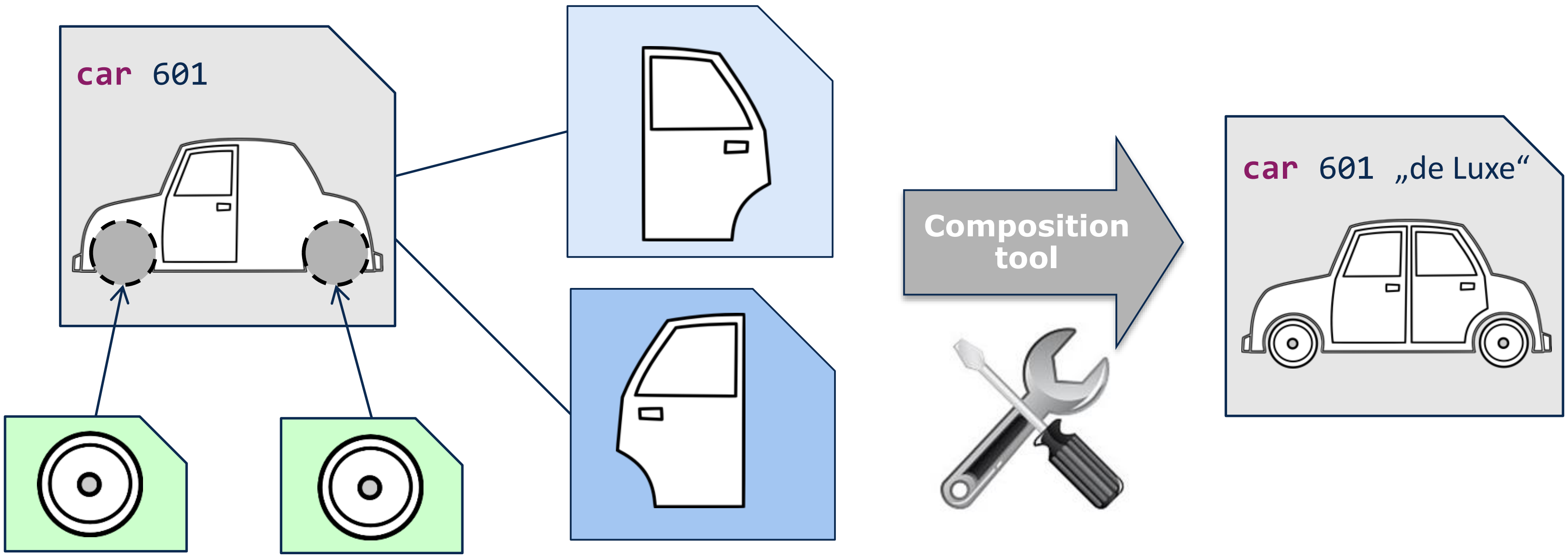
Part II: Well-Formed Invasive Software Composition

- Component Models with Attribute Grammars
- Composition Strategies
- Fragment Contracts
- Implementation in SkAT

Conclusion & Outlook

Composition in software development

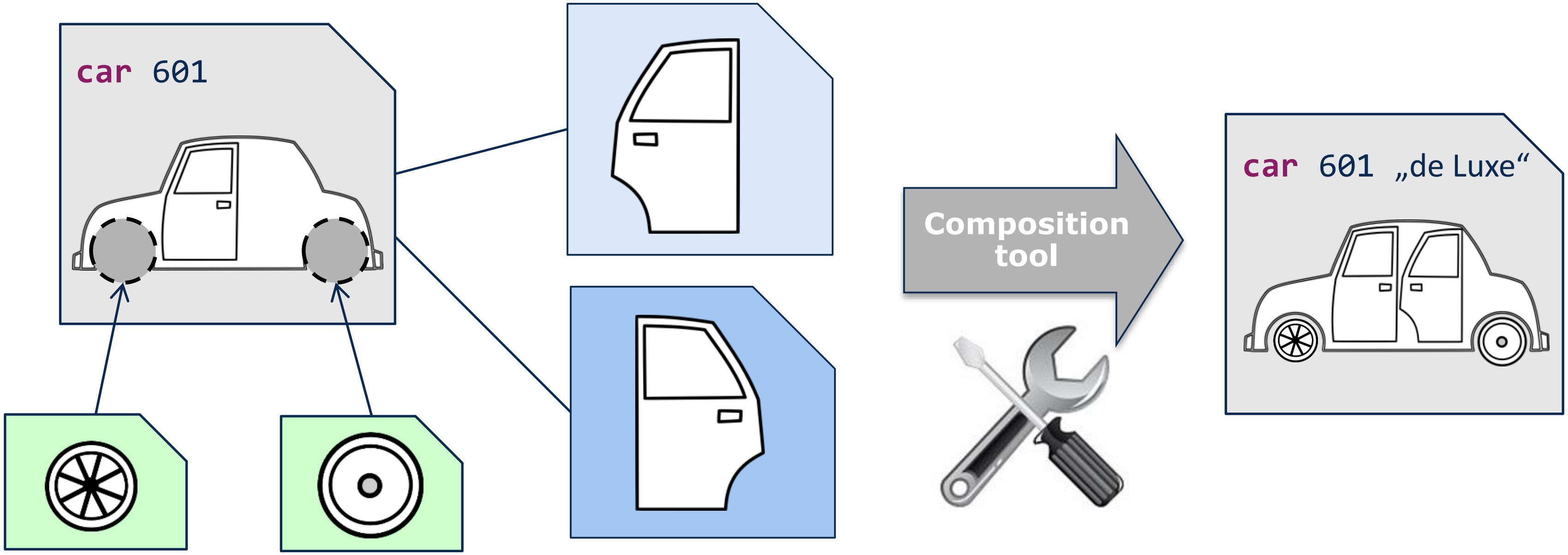
Software artifacts are automatically parameterized and extended by composition tools at every stage in the software life cycle.



Artifacts = code, models, documents ...

Composition in software development

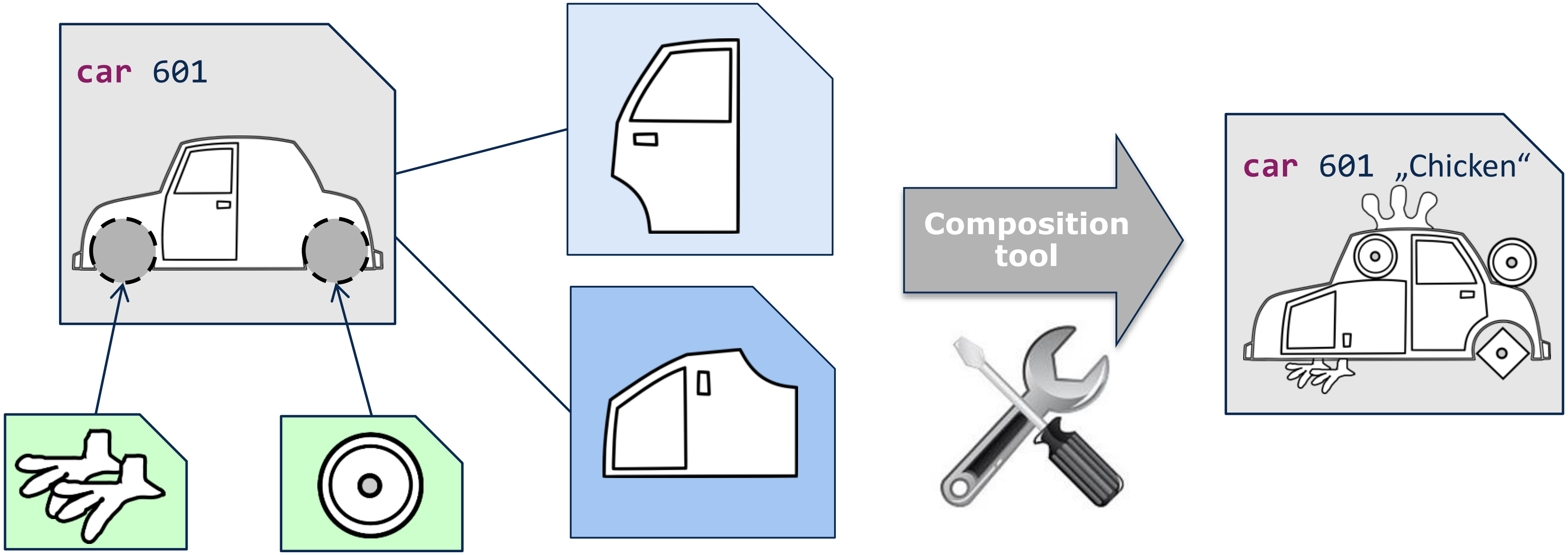
Software artifacts are automatically parameterized and extended by composition tools at every stage in the software life cycle.



Artifacts = code, models, documents ...

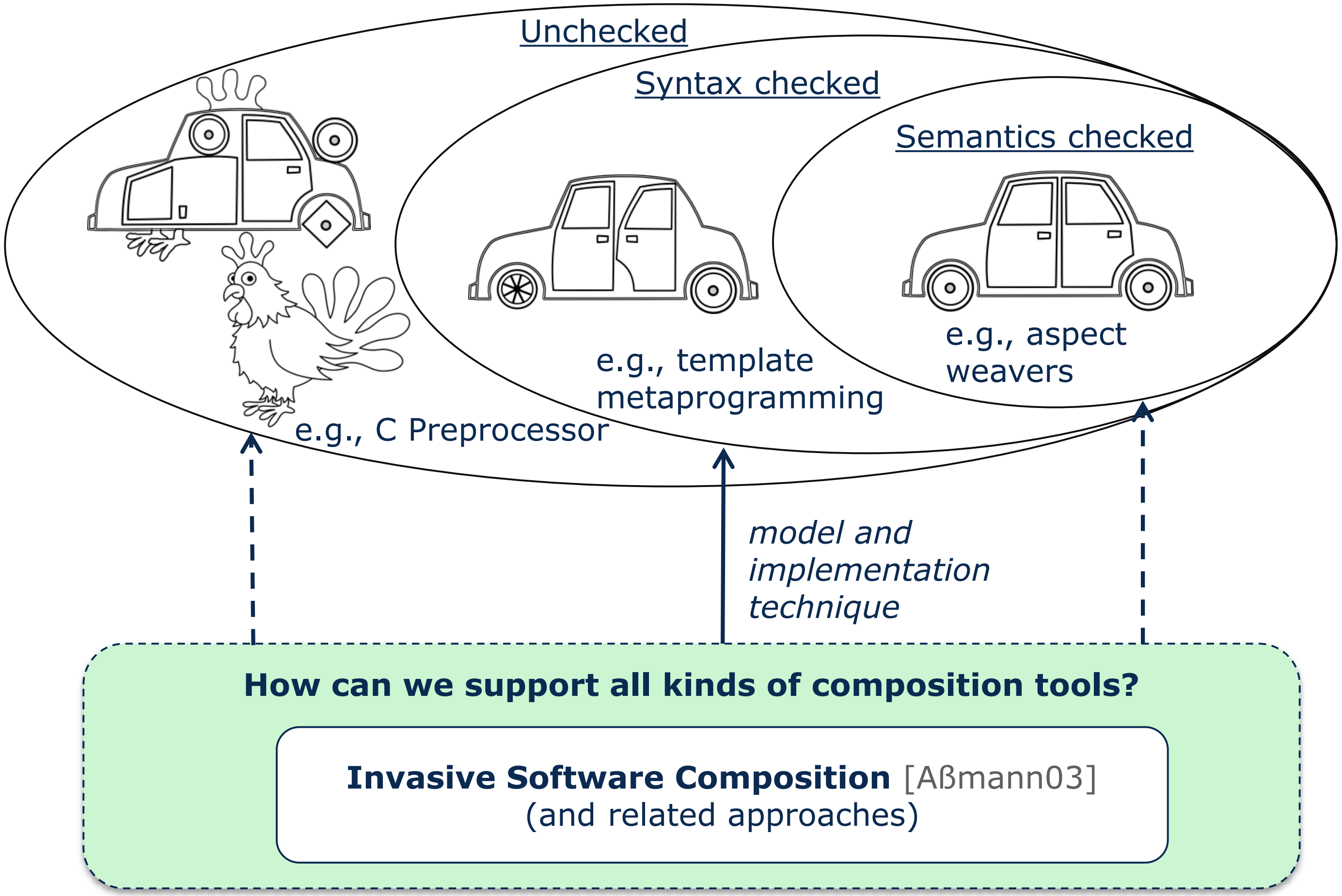
Composition in software development

Software artifacts are automatically parameterized and extended by composition tools at every stage in the software life cycle.



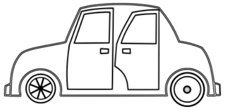
Artifacts = code, models, documents …

Motivation: Composition tooling



How can we support all kinds of composition tools?

Invasive Software Composition [Aßmann03]
(and related approaches)



Example: Composing Java fragments

Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }



}
```

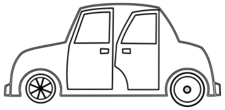
Fragment „brake“

```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

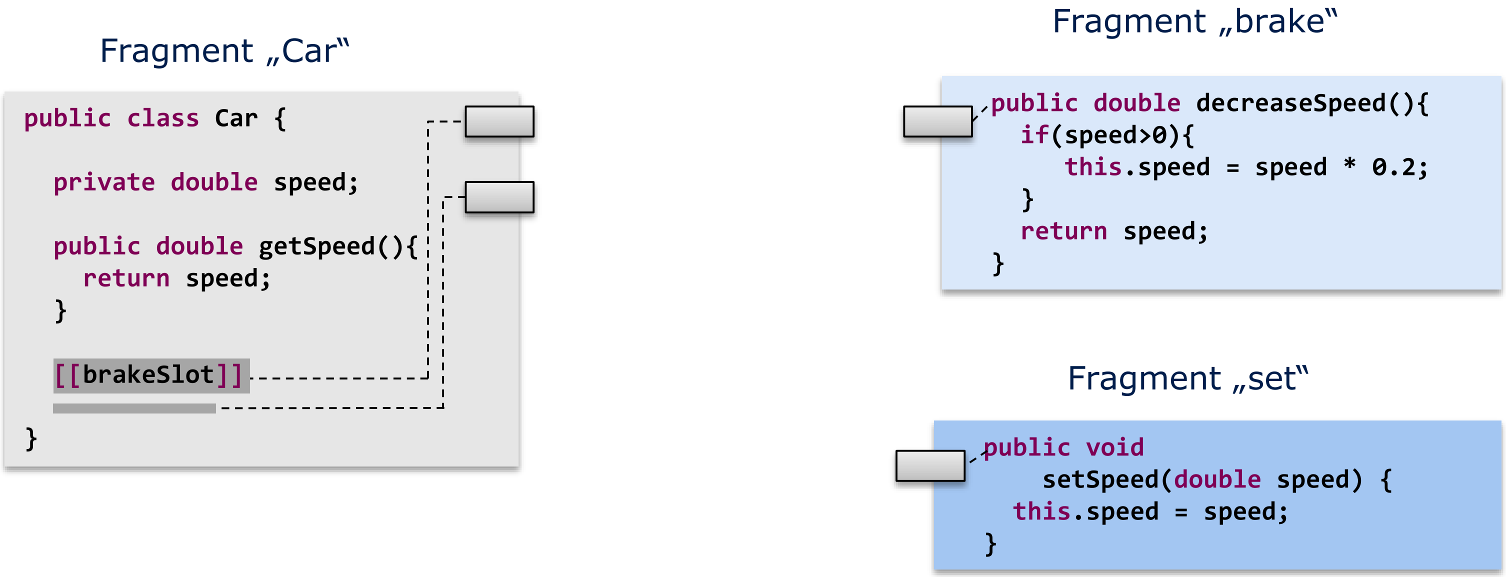
Fragment „set“

```
public void
    setSpeed(double speed) {
  this.speed = speed;
}
```

Fragment Component: partial or under-specified piece of an artifact (e.g., method, field declaration, class, expression, …)



Example: Composing Java fragments



Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  [[brakeSlot]]

}
```

Fragment „brake“

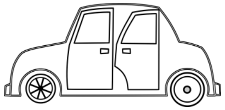
```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set“

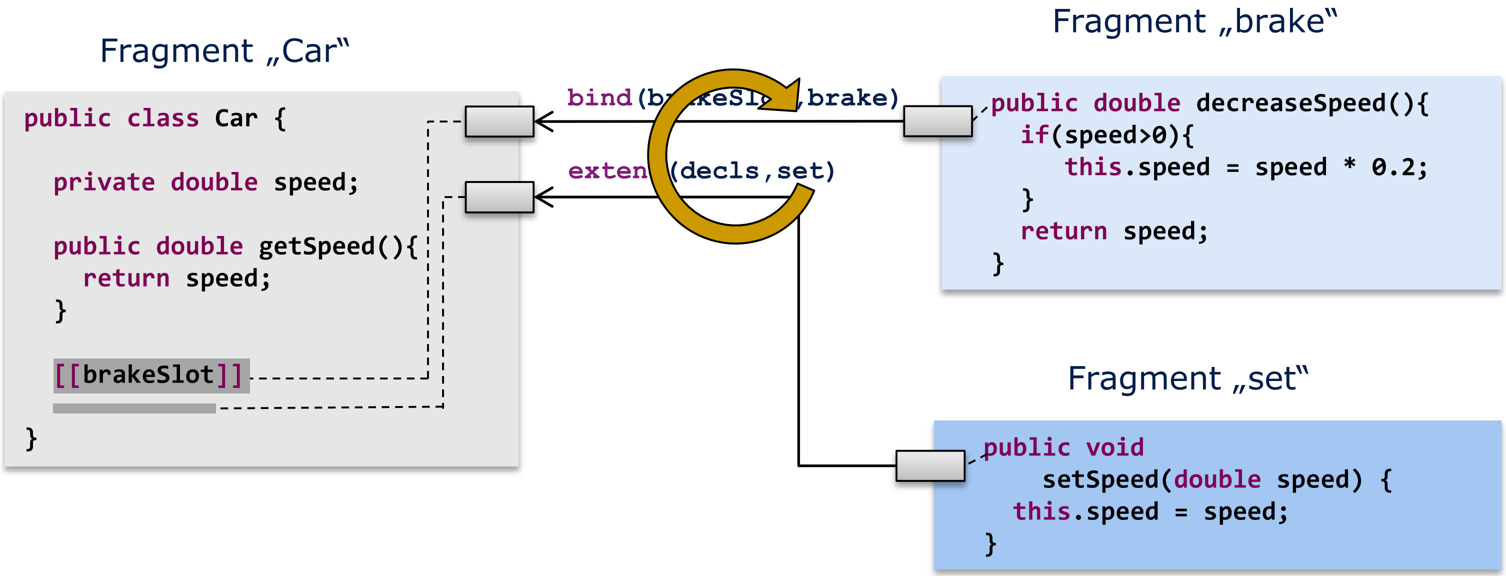
```
public void
   setSpeed(double speed) {
  this.speed = speed;
}
```

Fragment component model: interface definition

- **Slot**: *explicitly* declared *variation point* in a fragment
- **Hook**: *implicit extension point* in a fragment



Example: Composing Java fragments



Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  [[brakeSlot]]

}
```

Fragment „brake“

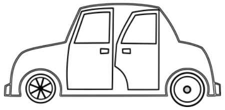
```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set“

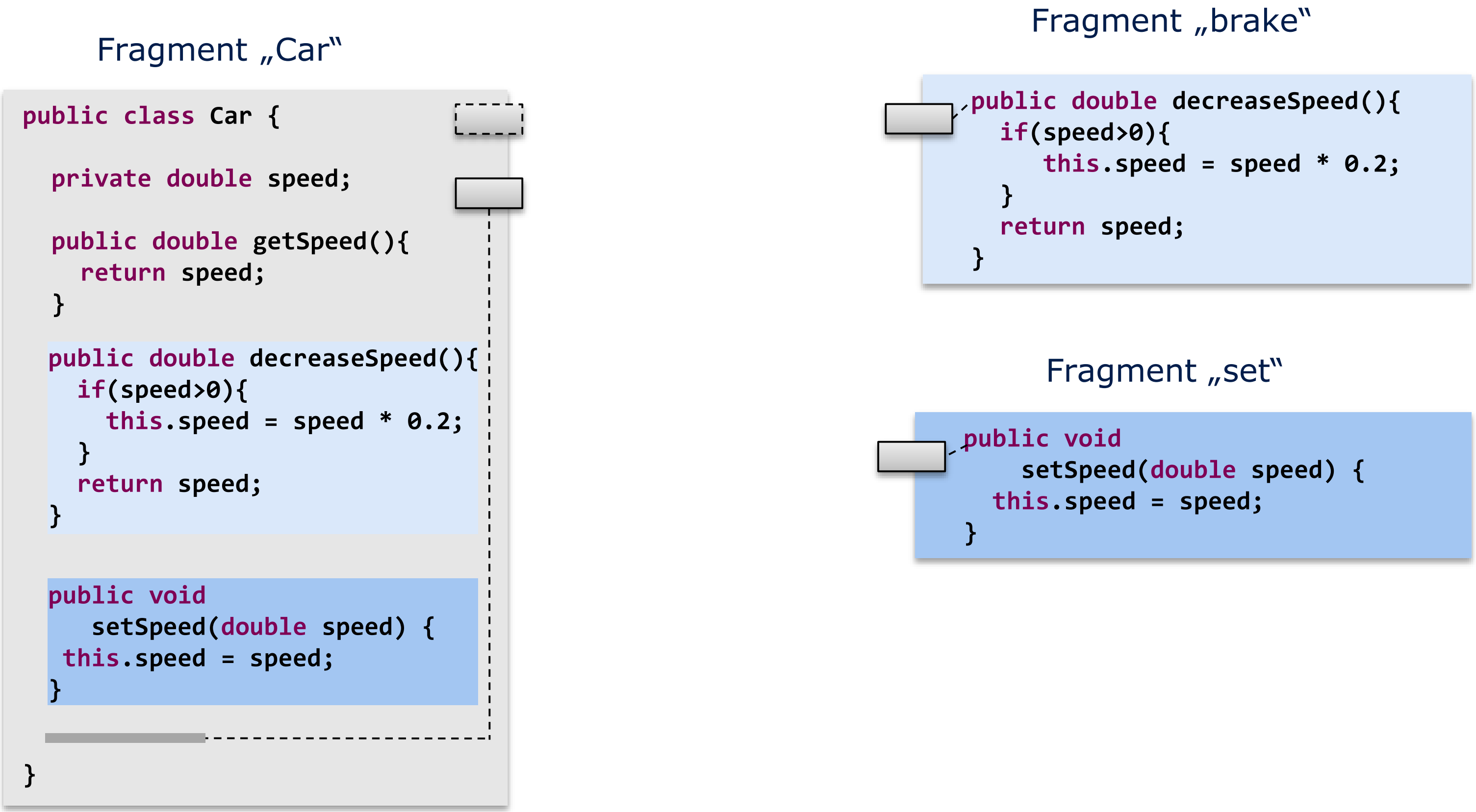
```
public void
   setSpeed(double speed) {
  this.speed = speed;
}
```

Composition program: specifies the actual composition

- imperative (e.g., template expansion, Java program)
- declarative (e.g., an aspect)
- domain-specific (e.g., embedded language, diagrams, …)



Example: Composing Java fragments



Fragment „Car“

```java
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }


  public void
     setSpeed(double speed) {
   this.speed = speed;
  }


}
```

Fragment „brake“

```java
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

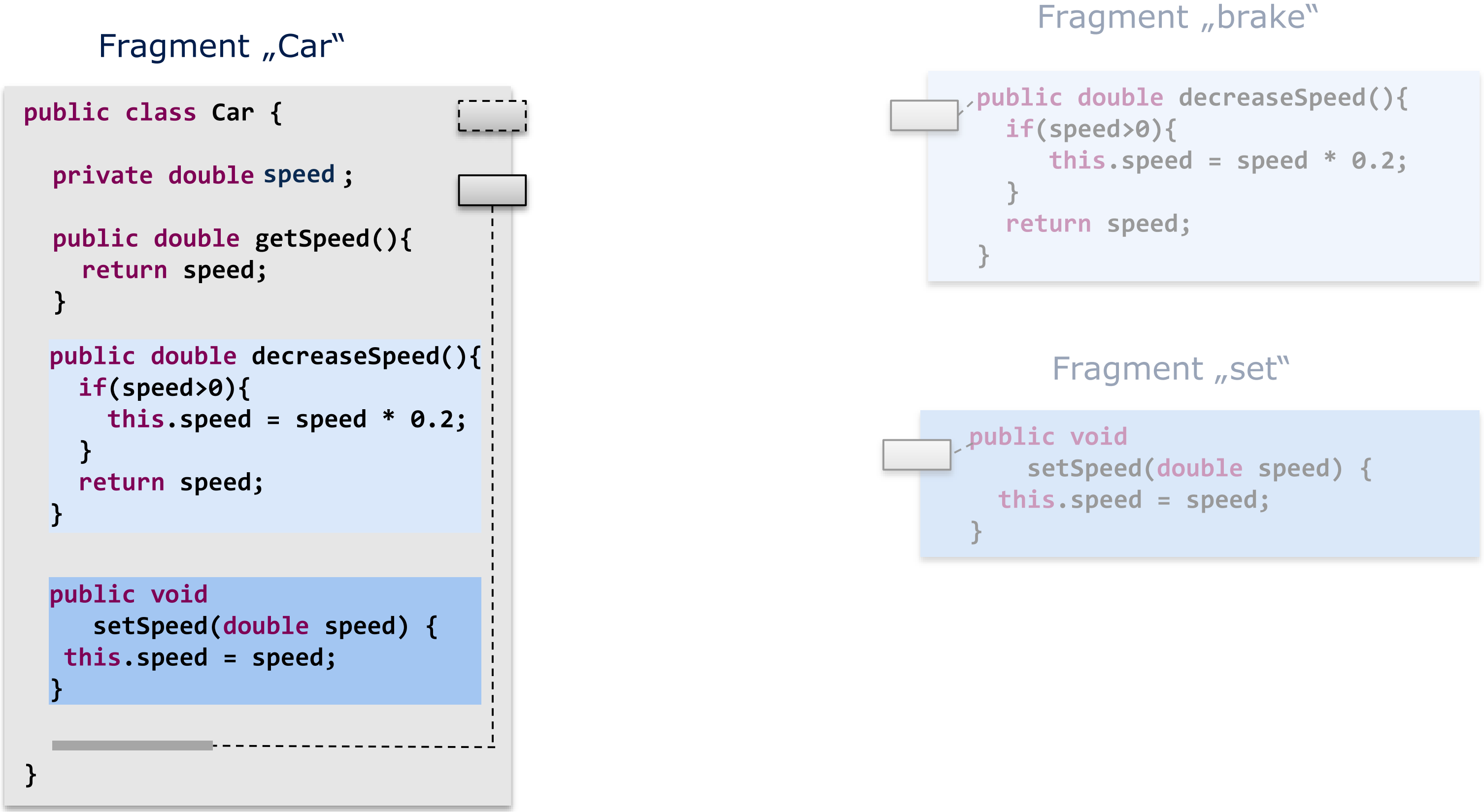
Fragment „set“

```java
public void
    setSpeed(double speed) {
  this.speed = speed;
}
```

Part I: Problem Analysis

Problem 1 [Validation]: Current ISC systems do not consider context-sensitive constraints (static semantics).

Example: Composing Java fragments



Fragment „Car“

```
public class Car {

  private double speed ;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }


  public void
     setSpeed(double speed) {
   this.speed = speed;
  }

}
```

Fragment „brake“

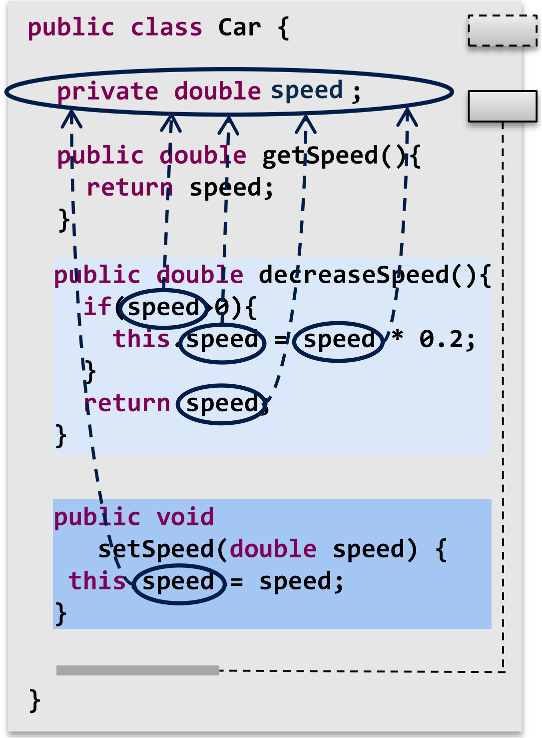
```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set“

```
public void
     setSpeed(double speed) {
   this.speed = speed;
 }
```

Example: Composing Java fragments

Fragment „Car"



```
public class Car {

  private double speed ;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }

}
```

Fragment „brake"

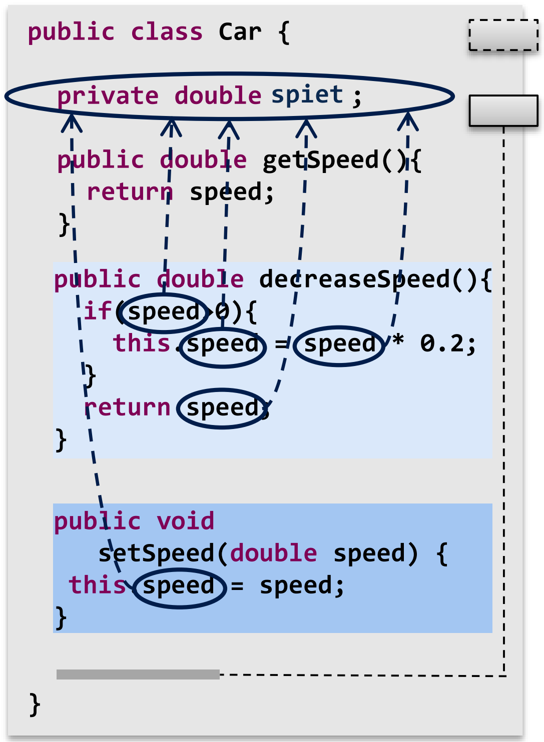
```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set"

```
public void
    setSpeed(double speed) {
  this.speed = speed;
}
```

Example: Composing Java fragments

Fragment „Car“



```
public class Car {

  private double spiet ;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }

}
```

Fragment „brake“

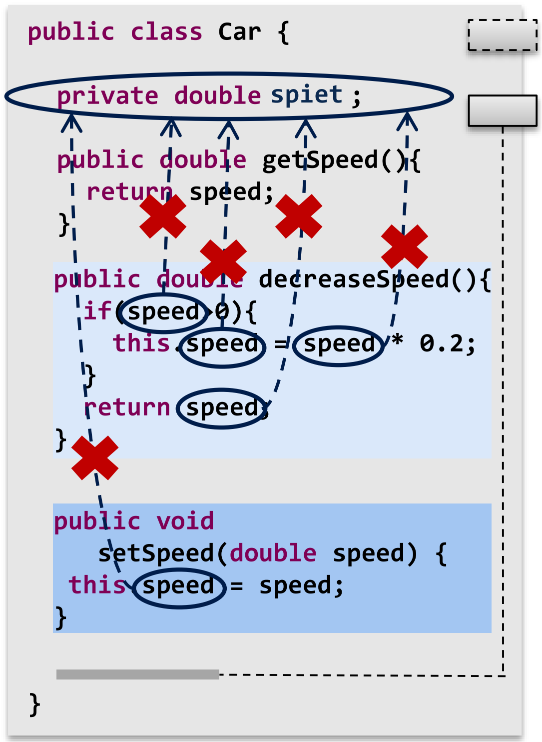
```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
 }
```

Fragment „set“

```
public void
     setSpeed(double speed) {
   this.speed = speed;
  }
```

Example: Composing Java fragments

Fragment „Car"



```
public class Car {

  private double spiet ;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }

  public void
     setSpeed(double speed) {
    this.speed = speed;
  }

}
```

Fragment „brake"

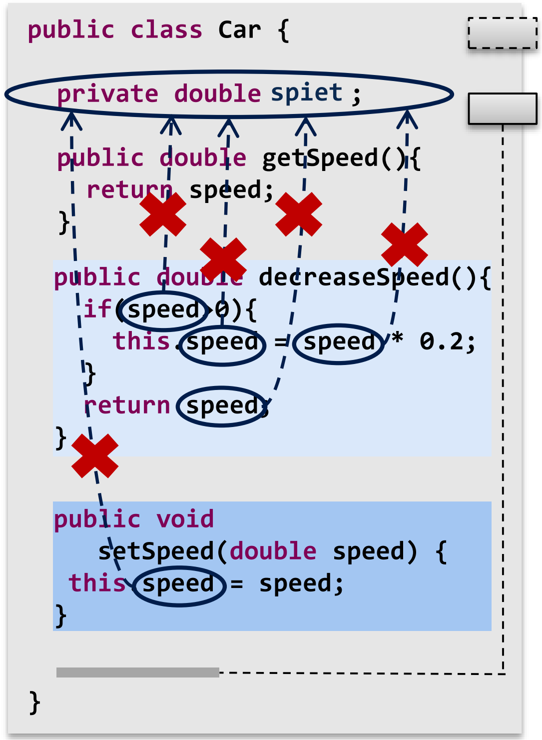
```
public double decreaseSpeed(){
  if(speed>0){
    this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set"

```
public void
   setSpeed(double speed) {
  this.speed = speed;
}
```

Example: Composing Java fragments

Fragment „Car“



```
public class Car {

  private double spiet ;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
    return speed;
  }

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }

}
```

Fragment „brake“

```
public double decreaseSpeed(){
  if(speed>0){
    this.speed = speed * 0.2;
  }
  return speed;
}
```

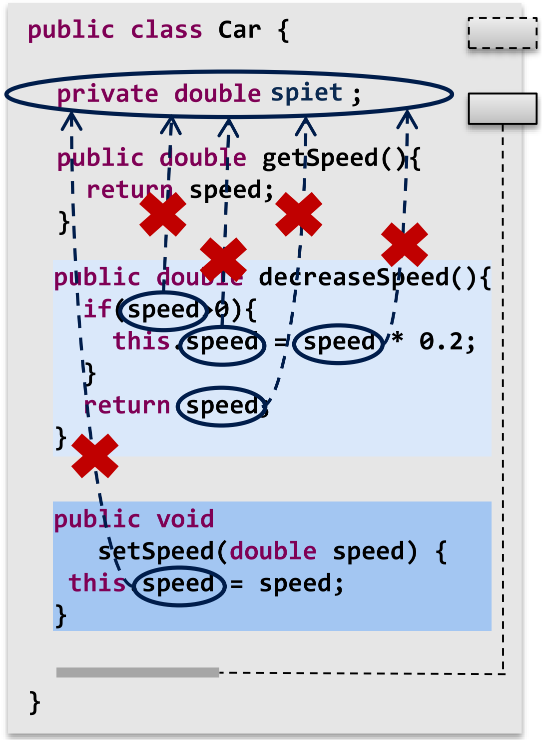
Fragment „set“

```
public void
     setSpeed(double speed) {
   this.speed = speed;
 }
```

```
Compile error(s) in Car.java:
  line 12: speed cannot be resolved to a variable
  line 13: speed cannot be resolved to a variable
  line 13: speed cannot be resolved or is not a field
  line 15: speed cannot be resolved to a variable
  line 20: speed cannot be resolved or is not a field
```

Example: Composing Java fragments

Fragment „Car"



```
public class Car {

  private double spiet ;

  public double getSpeed(){
   return speed;
  }

  public double decreaseSpeed(){
   if(speed>0){
     this.speed = speed * 0.2;
   }
   return speed;
  }

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }

}
```

Fragment „brake"

```
public double decreaseSpeed(){
  if(speed>0){
     this.speed = speed * 0.2;
  }
  return speed;
}
```

Fragment „set"

```
public void
   setSpeed(double speed) {
  this.speed = speed;
}
```

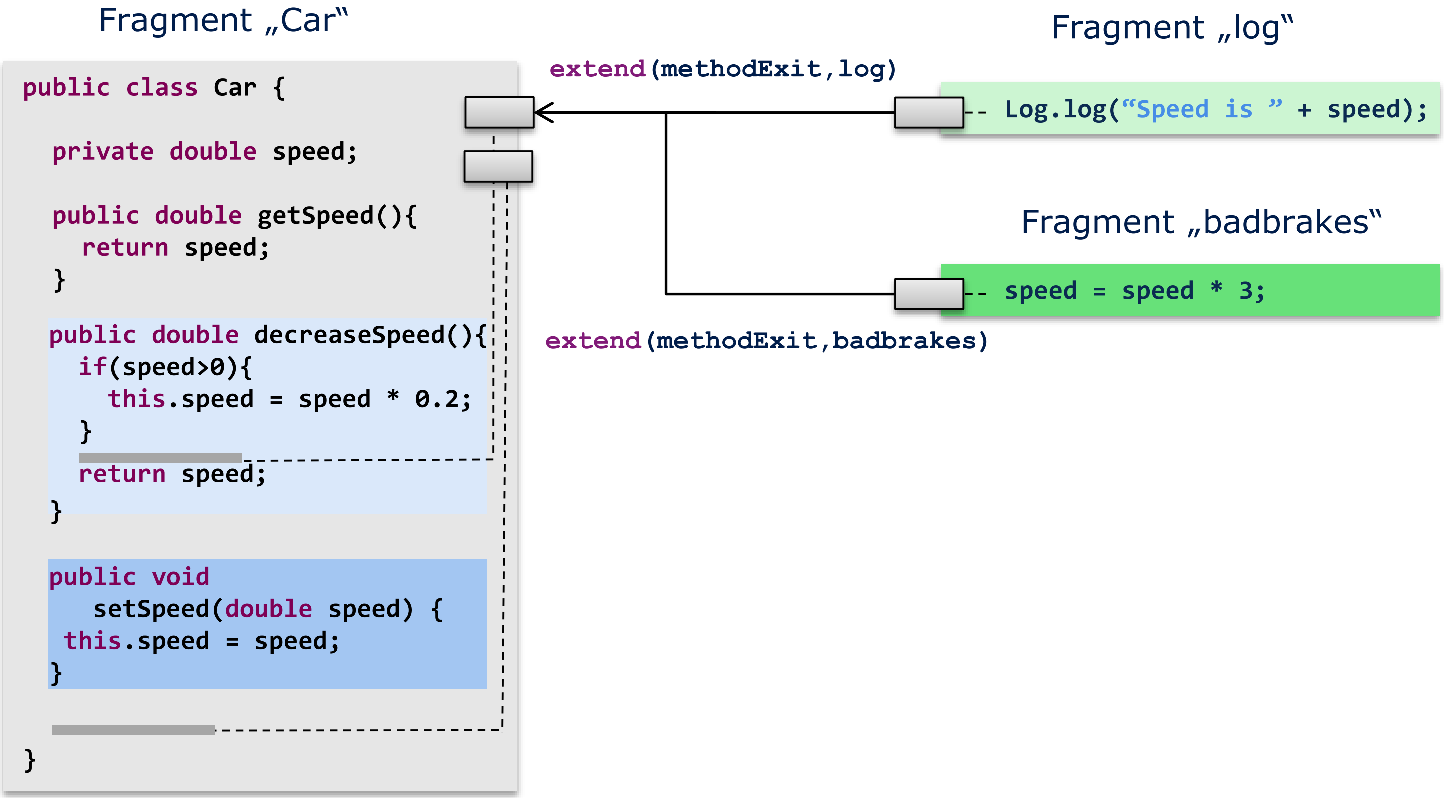
```
Problem(s) in composition program:
  line 1: cannot bind ‘brake’ to slot ‘brakeSlot’ –
          ‘speed’ not visible at ‘brakeSlot’
  line 2: cannot extend hook ‘decls’ with ‘set’ –
          ‘speed’ not visible at ‘decls’
```

Part I: Problem Analysis

Problem 1 [Validation]: Current ISC systems do not consider context-sensitive constraints (static semantics).

Problem 2 [Interaction]: Order of composition steps and their potential interactions have an impact on the result‘s semantics.

Example: Composing Java fragments (cont.)



Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }

    return speed;
  }

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }


}
```

extend(methodExit,log)

Fragment „log“

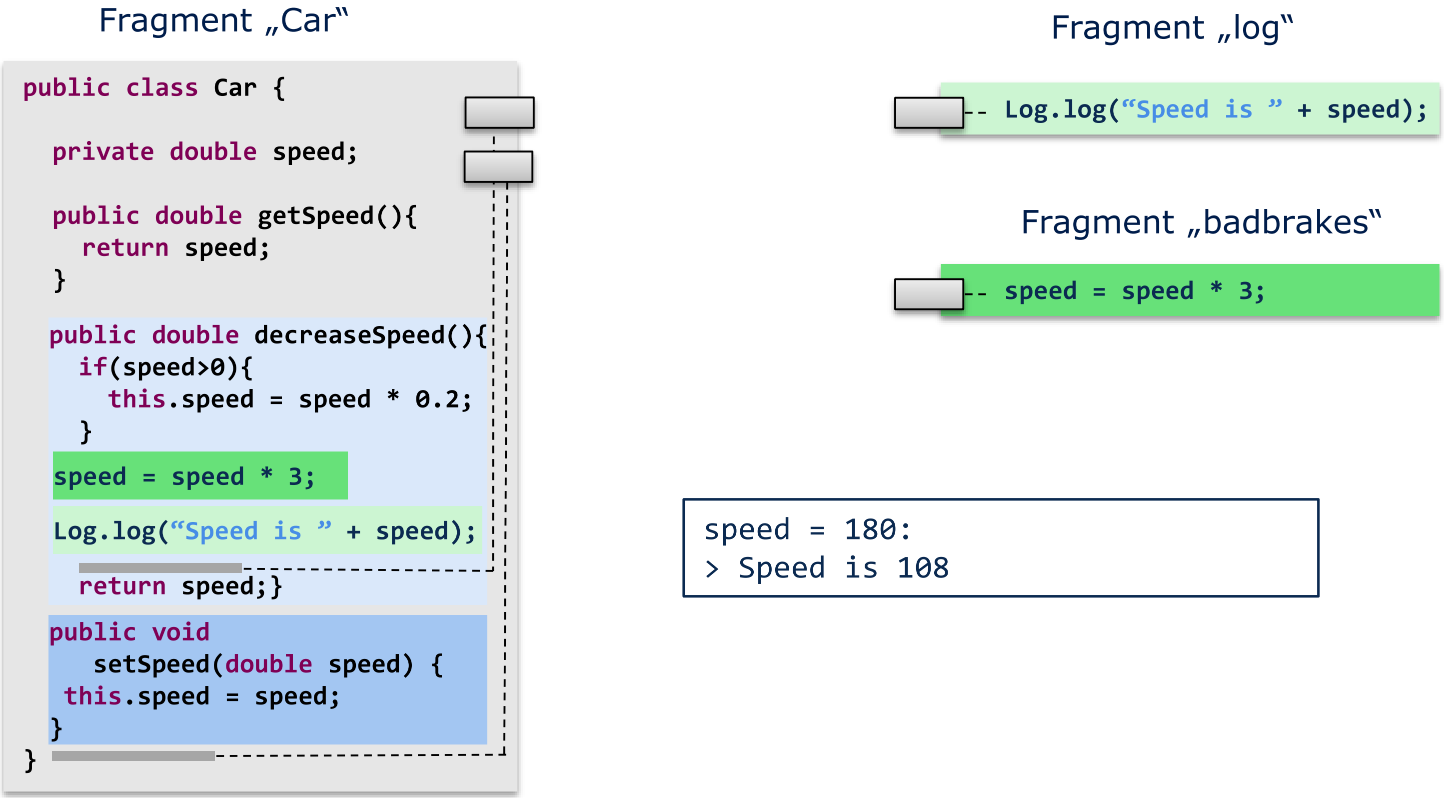
```
Log.log("Speed is " + speed);
```

Fragment „badbrakes“

```
speed = speed * 3;
```

extend(methodExit,badbrakes)

Example: Composing Java fragments (cont.)



Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
  speed = speed * 3;

  Log.log(“Speed is ” + speed);

    return speed;}

  public void
     setSpeed(double speed) {
   this.speed = speed;
  }
}
```

Fragment „log“

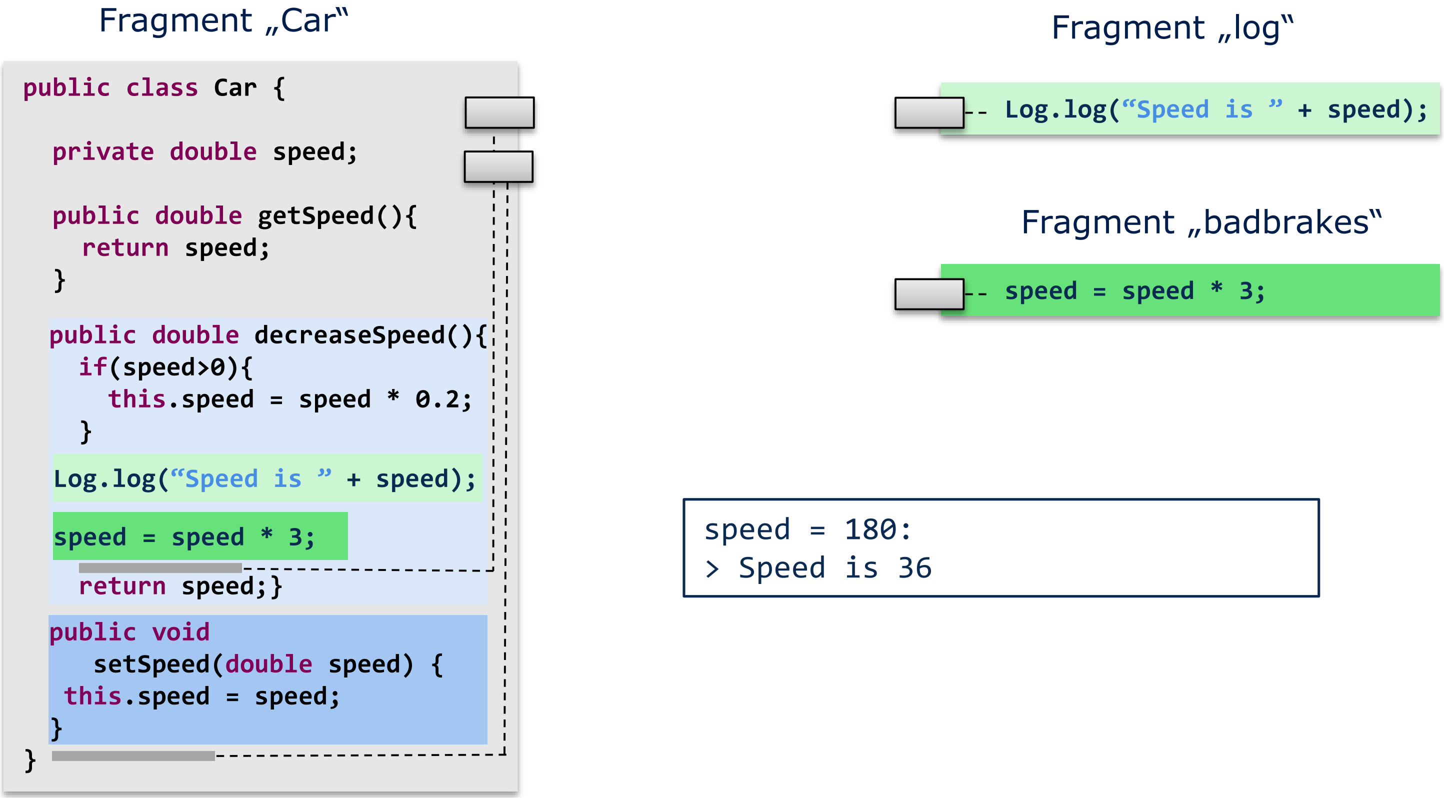
```
Log.log(“Speed is ” + speed);
```

Fragment „badbrakes“

```
speed = speed * 3;
```

```
speed = 180:
> Speed is 108
```

Example: Composing Java fragments (cont.)



Fragment „Car“

```
public class Car {

  private double speed;

  public double getSpeed(){
    return speed;
  }

  public double decreaseSpeed(){
    if(speed>0){
      this.speed = speed * 0.2;
    }
  Log.log(“Speed is ” + speed);

  speed = speed * 3;

    return speed;}
  public void
     setSpeed(double speed) {
   this.speed = speed;
  }
}
```

Fragment „log“

```
Log.log(“Speed is ” + speed);
```

Fragment „badbrakes“

```
speed = speed * 3;
```

```
speed = 180:
> Speed is 36
```

Part I: Problem Analysis

Problem 1 [Validation]: Current ISC systems do not consider context-sensitive constraints (static semantics).

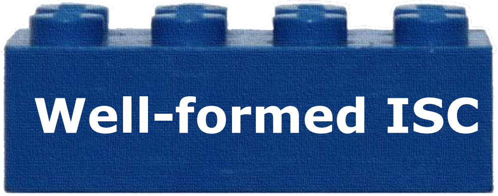
Problem 2 [Interaction]: Order of composition steps and their potential interactions have an impact on the result‘s semantics.

Problem 3 [Development costs]: The effort of implementing composition systems for complex or heterogeneous languages is high.

Development costs

- **ISC systems require parser, AST/model and component model**
    - n languages require at least 3*n specifications
        - format typically predetermined by tool
    - *good* if language grammar/metamodel available
    - *bad* if not
- **C++ (Draft 3797) has approx. 210 nonterminals**
    - practical implementations have far more
    - language is inherently ambiguous
    - *very* difficult for declarative and generative approaches

Part I: Thesis Contributions



Part I: Thesis Contributions

C1: Well-formed invasive software composition

Compartment	Addresses
C1a: Fragment component models based on *reference attribute grammars* (RAGs) ○ declarative specification of component models ○ supports context-sensitive constraints ○ extensible specification mechanism	P1 (validation) P3 (dev. costs)
C1b: Advanced composition technique based on strategies and rewrites	P2 (interaction)
C1c: Fragment contracts ○ integrate static semantics into composition ○ produce cause-related error messages	P1 (validation)

Part I: Thesis Contributions

Scalable ISC

Well-formed ISC

TECHNISCHE UNIVERSITÄT DRESDEN

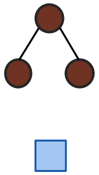
Part I: Thesis Contributions

C2: Scalable invasive software composition

Compartment	Addresses
C2a: Minimal invasive software composition ○ string-based ISC ○ minimal fragment component model **C2b**: Island fragment component models **C2c**: Agile composition system development ○ development in small iterations ○ rapid prototyping ○ minimal, classic and well-formed ISC in one process	P3 (dev. costs)

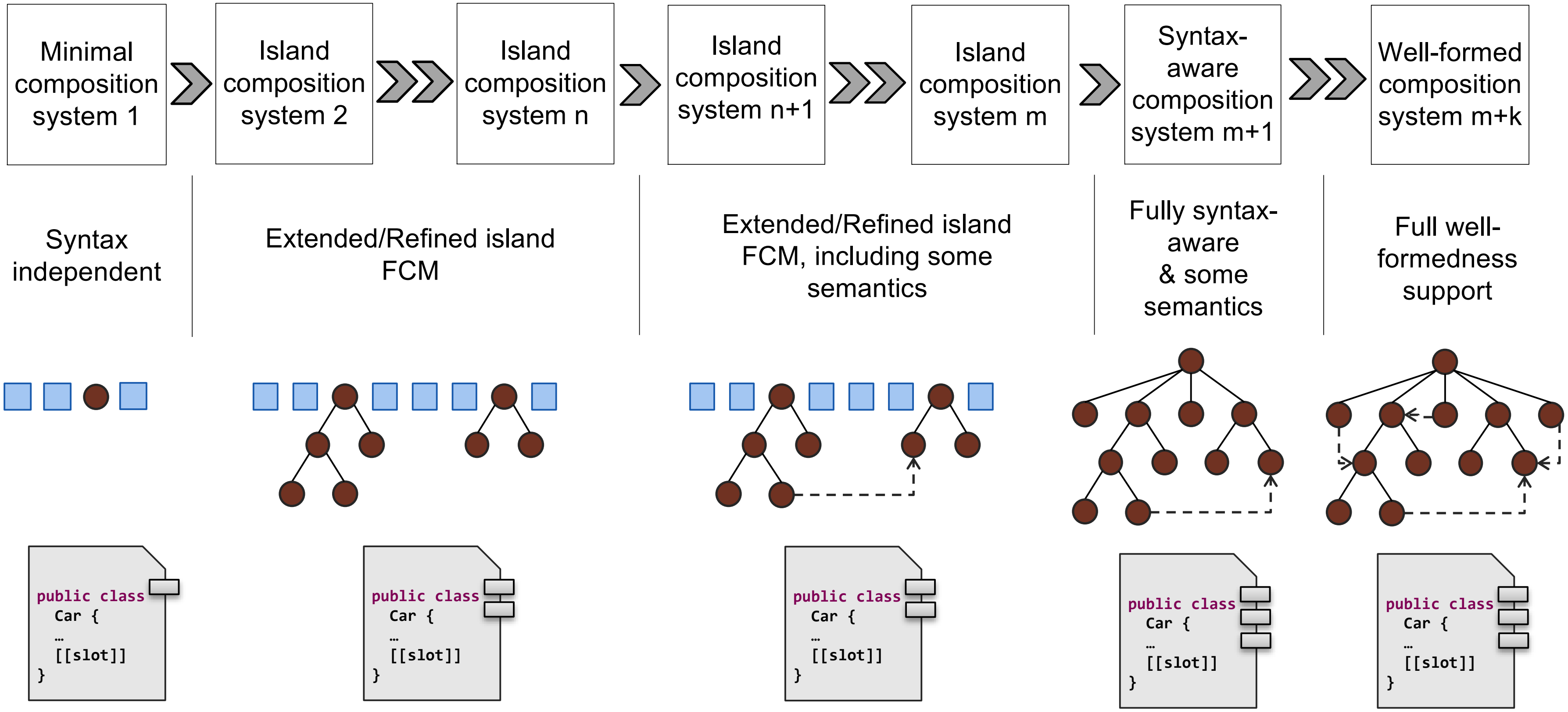
C2: Scalable invasive software composition

- Island component models: based on *island grammars* [Moonen01]
    - originally for „robust" parsing
    - context-free grammars
        - matching certain constructs – *islands*
        - ignoring the rest – *water*
    - well-suited for some extensible grammar formalisms (e.g., parsing expression grammars [Ford02,Ford04])



- Minimal fragment component model:
    - island component model with slots
    - suitable for any language

C2: Scalable invasive software composition



Minimal composition system 1

Island composition system 2

Island composition system n

Island composition system n+1

Island composition system m

Syntax-aware composition system m+1

Well-formed composition system m+k

Syntax independent

Extended/Refined island FCM

Extended/Refined island FCM, including some semantics

Fully syntax-aware & some semantics

Full well-formedness support

```
public class
  Car {
  …
  [[slot]]
}
```

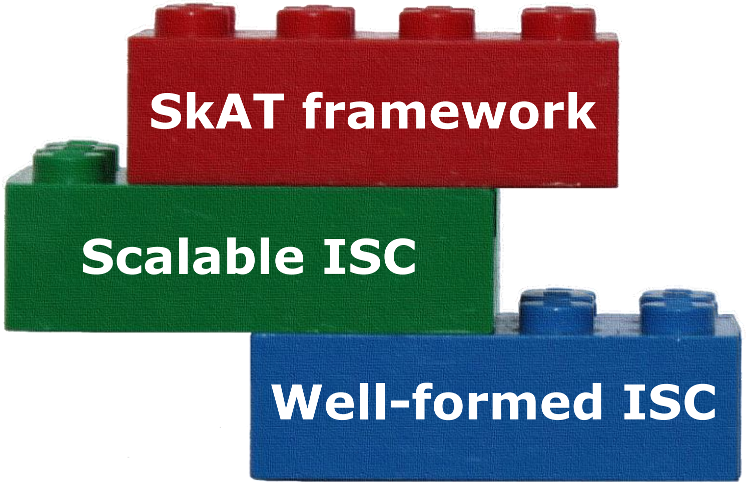
```
public class
  Car {
  …
  [[slot]]
}
```

```
public class
  Car {
  …
  [[slot]]
}
```

```
public class
  Car {
  …
  [[slot]]
}
```

```
public class
  Car {
  …
  [[slot]]
}
```

Part I: Thesis Contributions



SkAT framework

Scalable ISC

Well-formed ISC

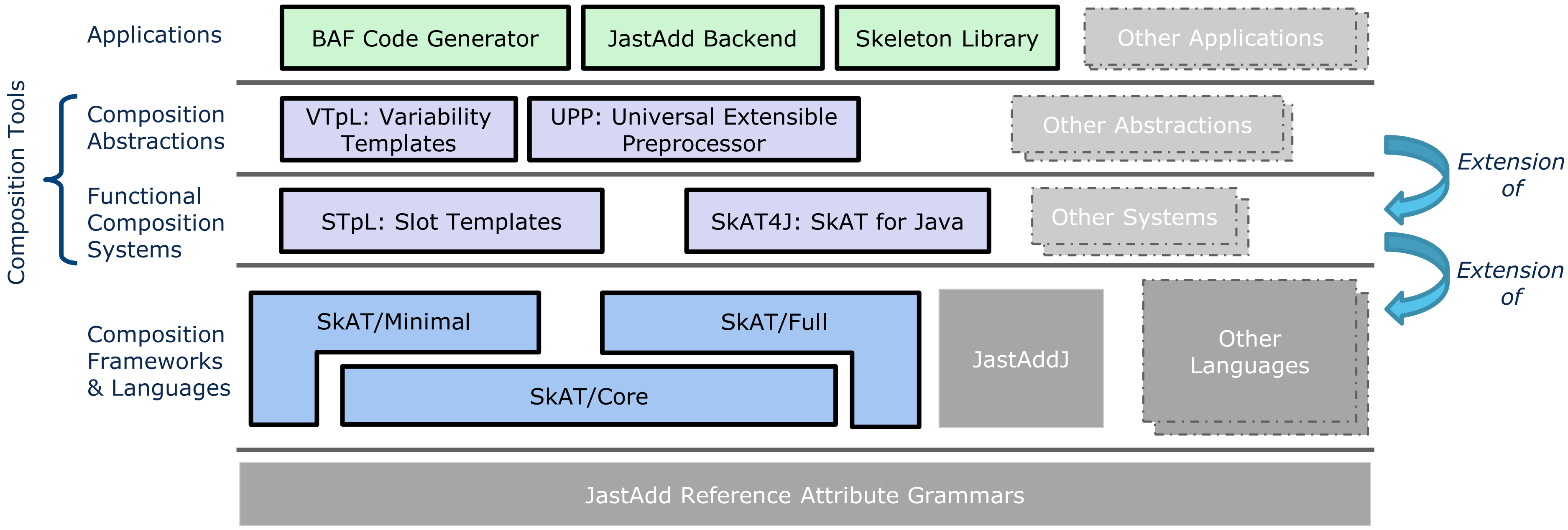
Part I: Thesis Contributions

C3: SkAT composition framework

Compartment	Implements
C3a: Generic implementation of well-formed ISC	C1a (FCM RAGs) C1b (strategies) C1c (contracts)
C3b: SkAT4J – a well-formed composition system for Java **C3c**: A library of parallel algorithmic skeletons	C1 demonstrator
C3d: Implementation of scalable ISC	C2a (minimal ISC) C2b (island FCMs)
C3e: Three extensible island composition tools ○ STpL – Slot Template Language ○ VTpL – Variability Template Language ○ UPP – Universal Extensible Preprocessor	C2 demonstrators

C3: SkAT composition framework – modular architecture

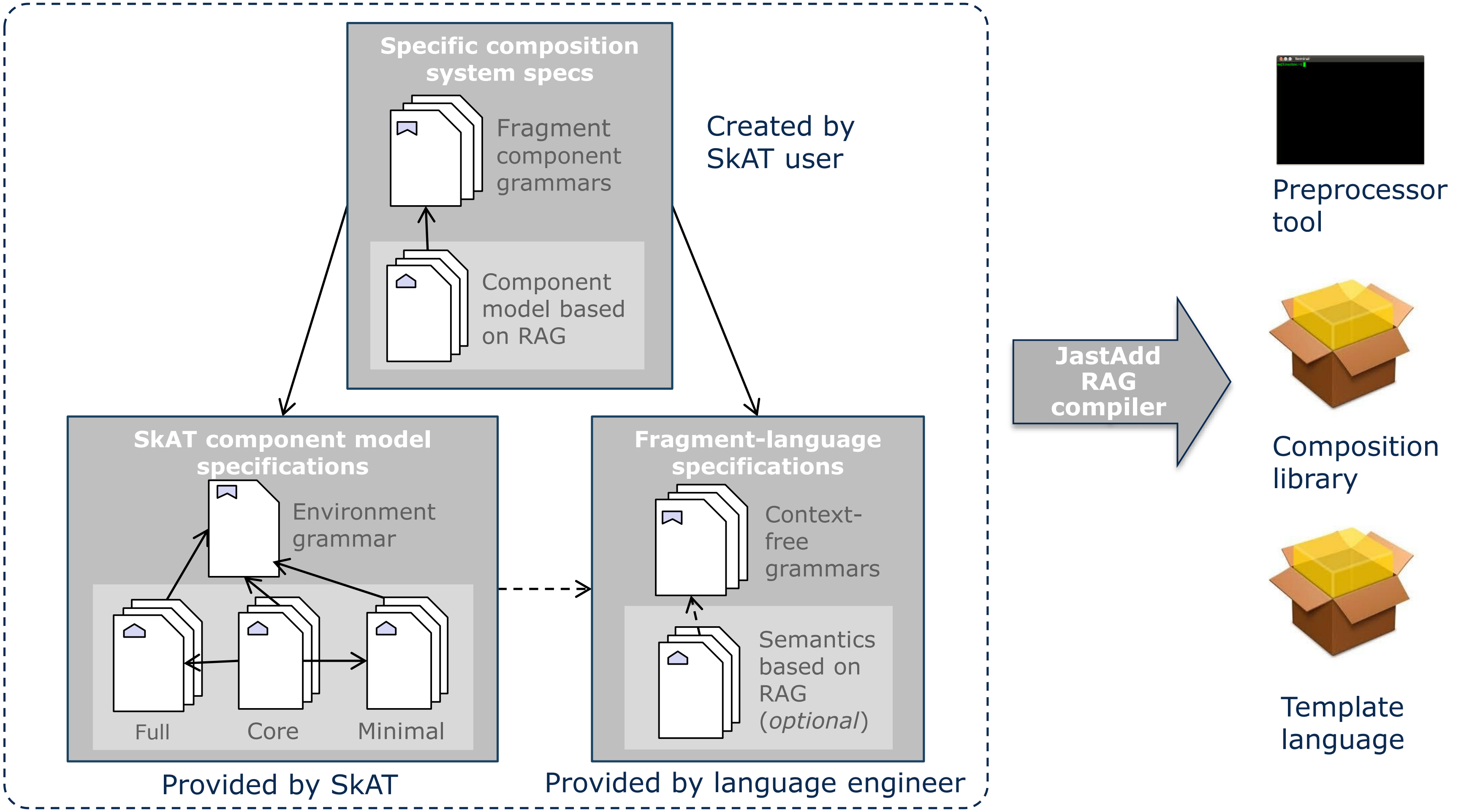
- "**Sk**eletons and **A**pplication **T**emplates"
- Implementation in Java
  - using JastAdd *reference attribute grammars (RAGs)* [Hedin00]
  - following the principles of extensible compiler construction [Ekman06]



TECHNISCHE UNIVERSITÄT DRESDEN

Part I: Thesis Contributions

C3: SkAT composition framework



Specific composition system specs

Fragment component grammars

Component model based on RAG

Created by SkAT user

SkAT component model specifications

Environment grammar

Full

Core

Minimal

Provided by SkAT

Fragment-language specifications

Context-free grammars

Semantics based on RAG (*optional*)

Provided by language engineer

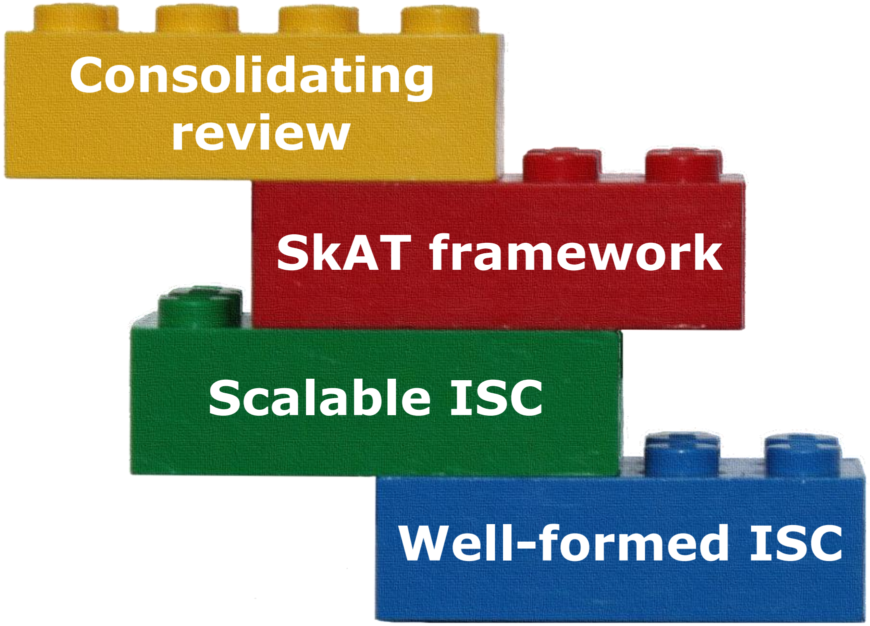
JastAdd RAG compiler

Preprocessor tool

Composition library

Template language

Part I: Thesis Contributions



Consolidating review

SkAT framework

Scalable ISC

Well-formed ISC

Part I: Thesis Contributions

C4: Consolidating review of the state-of-the-art in ISC

C4a: A formal model of classic ISC
- as least common model

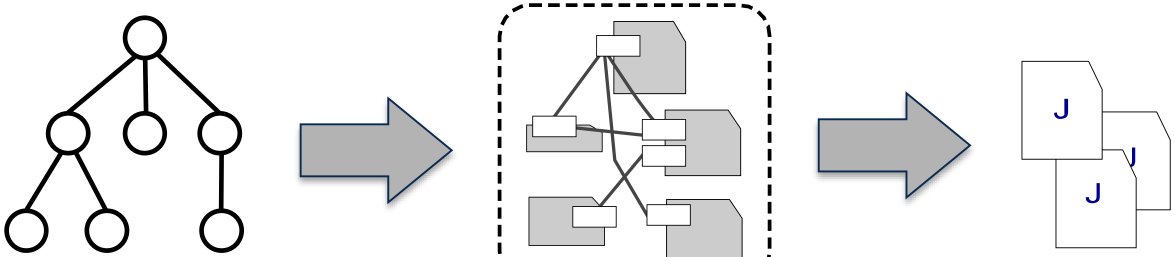
C4b: Case study - the *Business Application Framework* (BAF)
- common scenario implemented in different ISC frameworks

C4c: Analysis and comparison of existing ISC tools and SkAT
- focus on code generation
- and features of classic ISC

Part I: Thesis Contributions

C4: Consolidating review

BAF scenario: generating code from a business domain model



Business domain model

Code generator with fragments

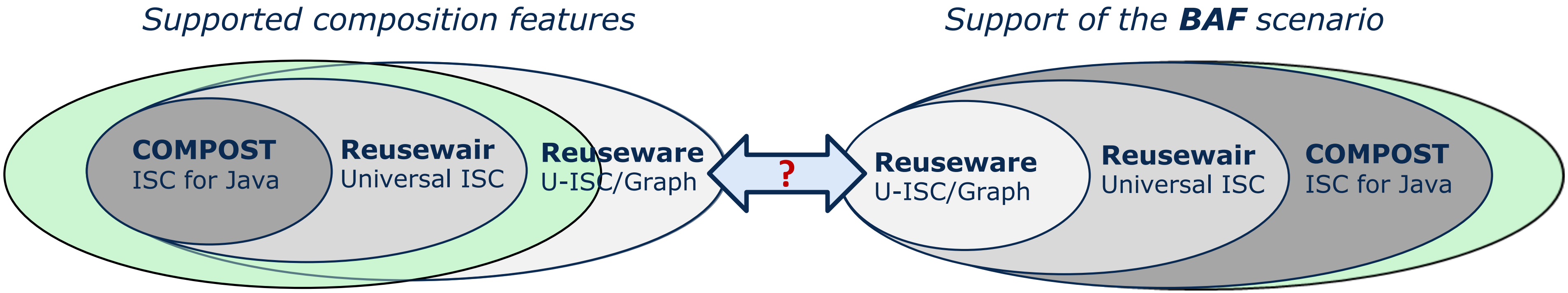
Platform code, e.g., for JavaEE

- typed by a metamodel
- textual specification
- implies various requirements on composition systems

Implementations:

- COMPOST [Aßmann03]
- Reuse*wair* [Henriksson09]
- Reuseware [Johannes11]
- SkAT

C4: Consolidating review – results in brief



Not a contradiction!

- **COMPOST**: handmade system for Java (*ok for scenario, nearly complete*)
- **Reusewair**: generated from DSL (*partial support, too restrictive*)
- **Reuseware**: graphical and interactive composition (*not suitable in this case*)

Improvements by SkAT

- adds new composition features
- but also consolidates on the core of ISC

Outline

Part I: Overview

- Invasive Software Composition
- Problem Analysis
- Thesis Contributions

Part II: Well-Formed Invasive Software Composition

- Component Models with Attribute Grammars
- Composition Strategies
- Fragment Contracts
- Implementation in SkAT

Conclusion & Outlook

Attribute grammars

Formalism to compute static semantics over syntax trees [Knuth68]

- Basis: context-free grammars + attributes + semantic functions (equations)

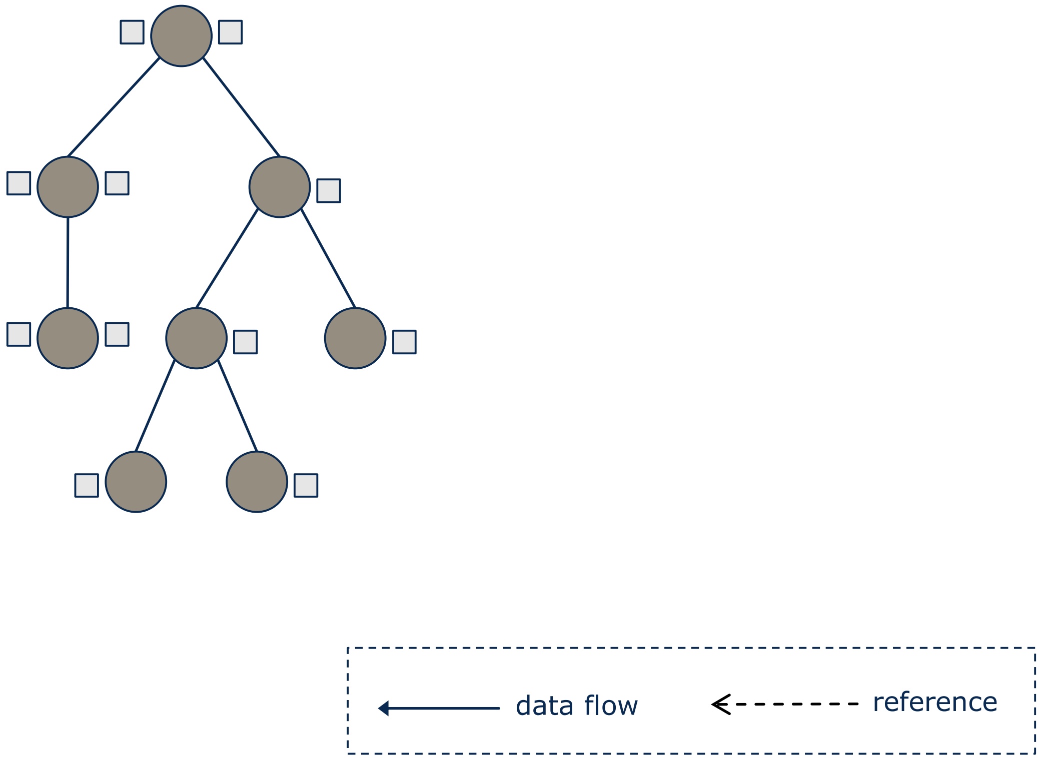
- Evaluation by tree visitors with different visiting strategies
  - static: ordered attribute grammars (OAGs)
  - dynamic: demand-driven evaluation

- AGs are modular and extensible

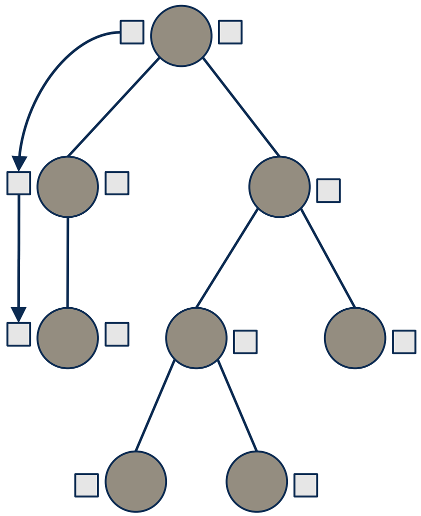
Extensions

- Higher order attribute grammars (HOAGs) [Vogt+89]
- Reference attribute grammars (RAGs) [Hedin00,Boyland05]

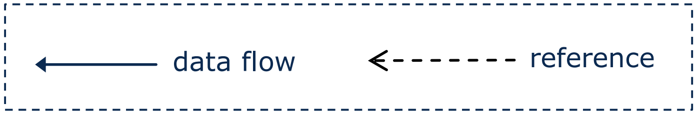
Attribute grammars: kinds of attributes



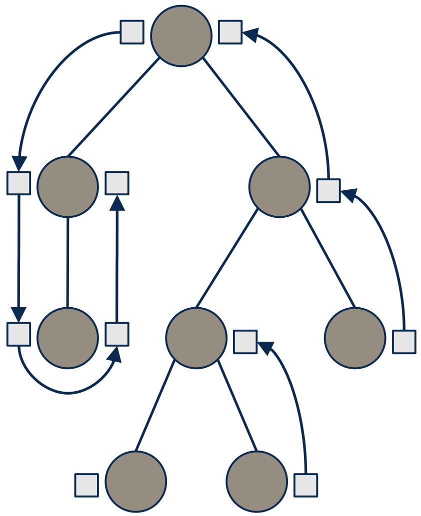
Attribute grammars: kinds of attributes



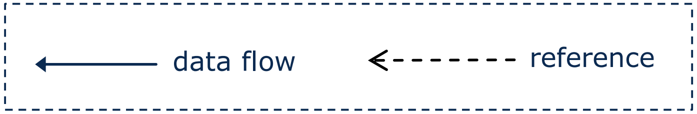
- **Inherited attributes** (inh): top-down value dataflow/computation



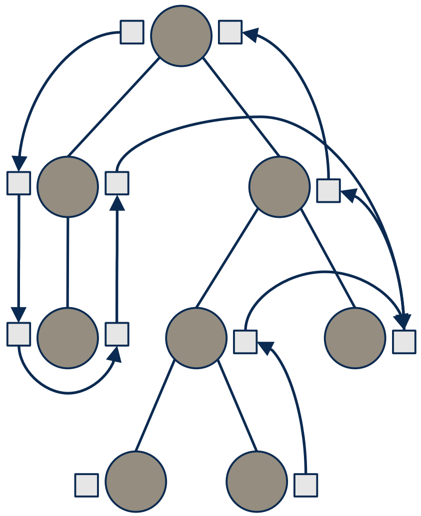
Attribute grammars: kinds of attributes



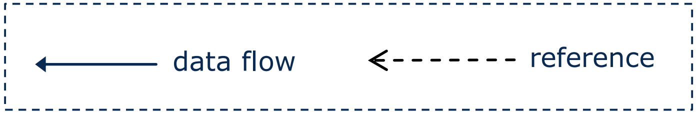
- **Inherited attributes** (inh): top-down value dataflow/computation
- **Synthesized attributes** (syn): bottom-up value dataflow/computation



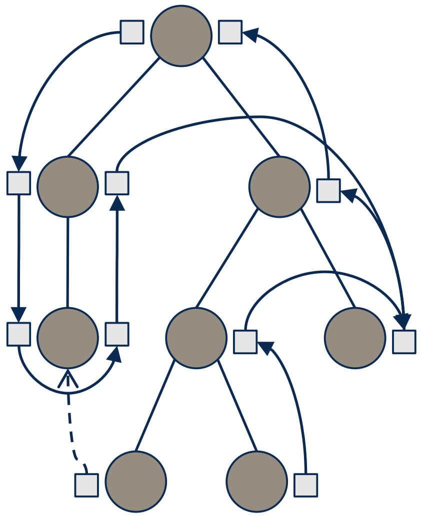
Attribute grammars: kinds of attributes



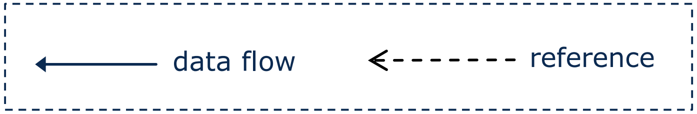
- **Inherited attributes** (inh): top-down value dataflow/computation
- **Synthesized attributes** (syn): bottom-up value dataflow/computation
- **Collection attributes** (coll): collect values freely distributed over the AST



Attribute grammars: kinds of attributes



- **Inherited attributes** (inh): top-down value dataflow/computation
- **Synthesized attributes** (syn): bottom-up value dataflow/computation
- **Collection attributes** (coll): collect values freely distributed over the AST
- **Reference attributes**: compute references to existing nodes in the AST

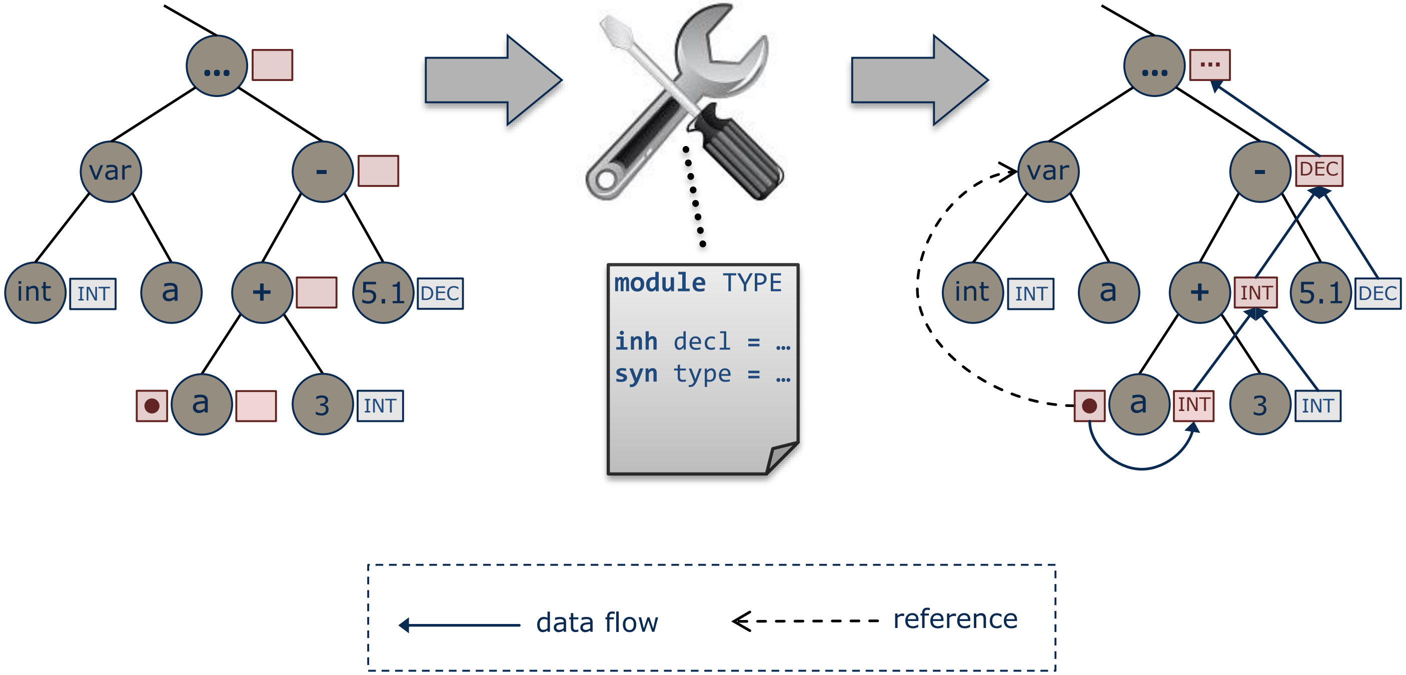


Attribute grammars: RAG tools

Input: AST (from parser/ editor/transformer)

Tool: RAG evaluator (generated or interpreted)

Result: AST with overlay graph



```
module TYPE

inh decl = …
syn type = …
```

How to marry attribute grammars and ISC?

1. Provide an integrated composition environment grammar
2. Use attributes and equations to identify compositional points
3. Extend composition environment with composer declarations
4. Define composition algorithms transforming the environment
5. Add contract attributes for well-formed composition

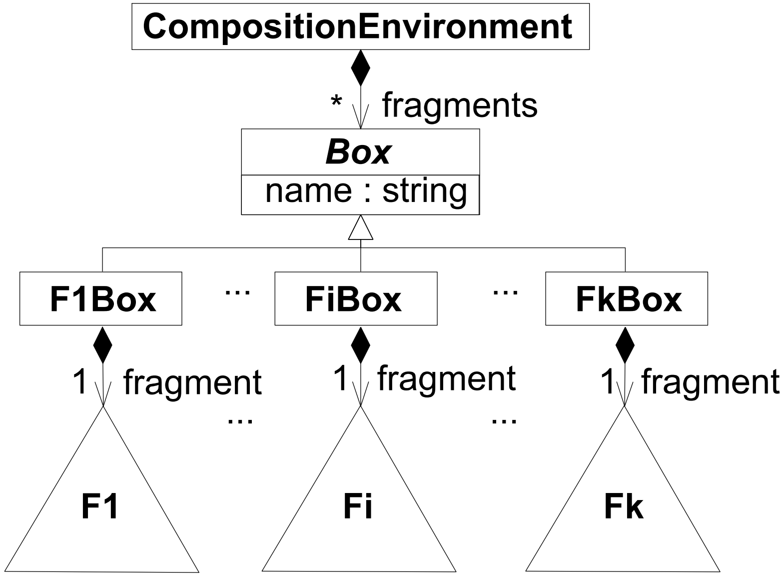
RAG-based component models: composition environments

- Provides *spanning tree* for attribute evaluation
- Hosts fragments and attributes of the component model
- Defines one *box type* per fragment type (“Boxology” [Aßmann03])

```
external F1,...,Fi,...,Fk
CompositionEnvironment ::= fragments:Box*
@Box ::= name:<string>
F1Box ▷ Box  ::= fragment:F1
          ...
FkBox ▷ Box  ::= fragment:Fk
```

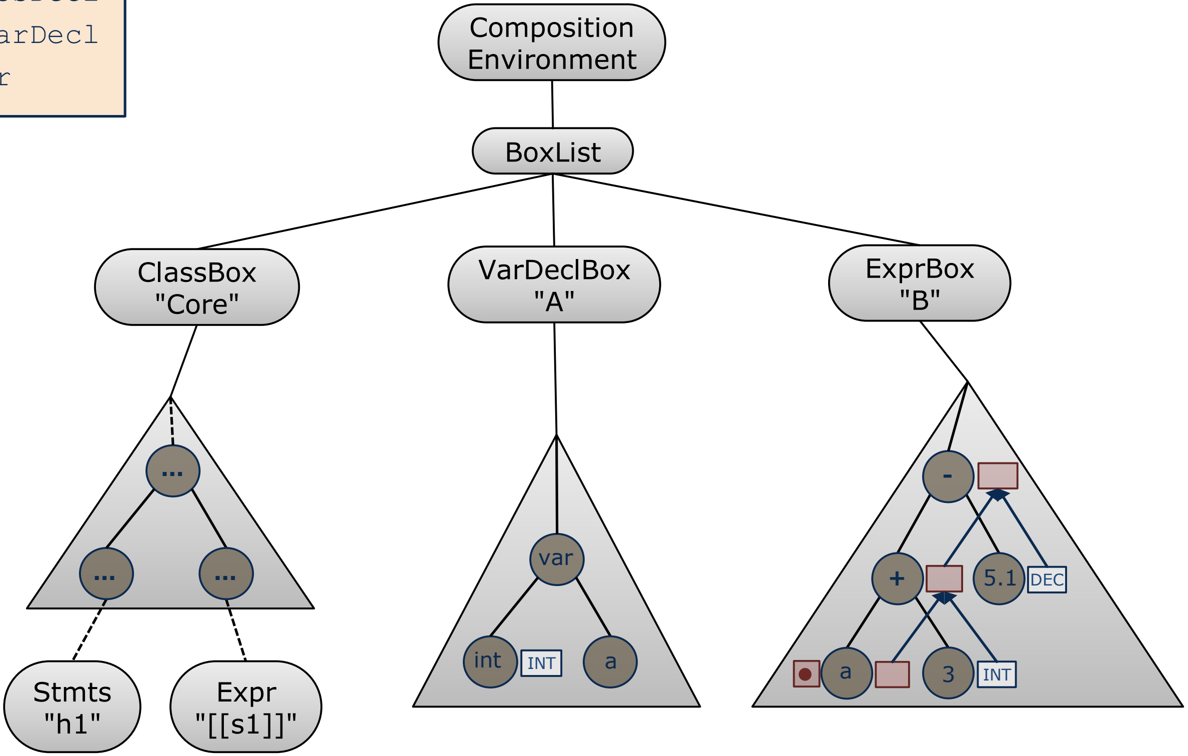
SimpAG specification language

- flat EBNF for context-free grammars
- abstract nonterminals: @
- nonterminal inheritance: ▷



RAG-based component models: composition environments

```
external ClassDecl,VarDecl,Expr
ClassBox ▷ Box ::= fragment:ClassDecl
VarDeclBox ▷ Box ::= fragment:VarDecl
ExprBox ▷ Box ::= fragment:Expr
```



SimpAG specification language

- flat EBNF for context-free grammars
- abstract nonterminals: @
- nonterminal inheritance: ▷

RAG-based component models: compositional points

- Points are identified and collected by attributes
    - slots: identified using *synthesized* attributes → self-defined
    - hooks: identified using *inherited* attributes → context-depend

Slot *identification*

$\mathbf{syn}\ \texttt{bool}_\perp\ \{n \mid n \in N\}.\texttt{isSlot}$ $\mathbf{syn}\ \texttt{string}_\perp\ \{n \mid n \in N\}.\texttt{slotName}$

$$\mathbf{fun}\ n_i.\texttt{isSlot} = \begin{cases} \mathit{true} & \mathbf{if}\ \textit{node matches pattern}, \\ \mathit{false} & \mathbf{else}. \end{cases}$$

$$\mathbf{fun}\ n_i.\texttt{slotName} = \begin{cases} \textit{name from } n_i\textit{-context} & \mathbf{if}\ \textit{node is a slot}, \\ \perp & \mathbf{else}. \end{cases}$$

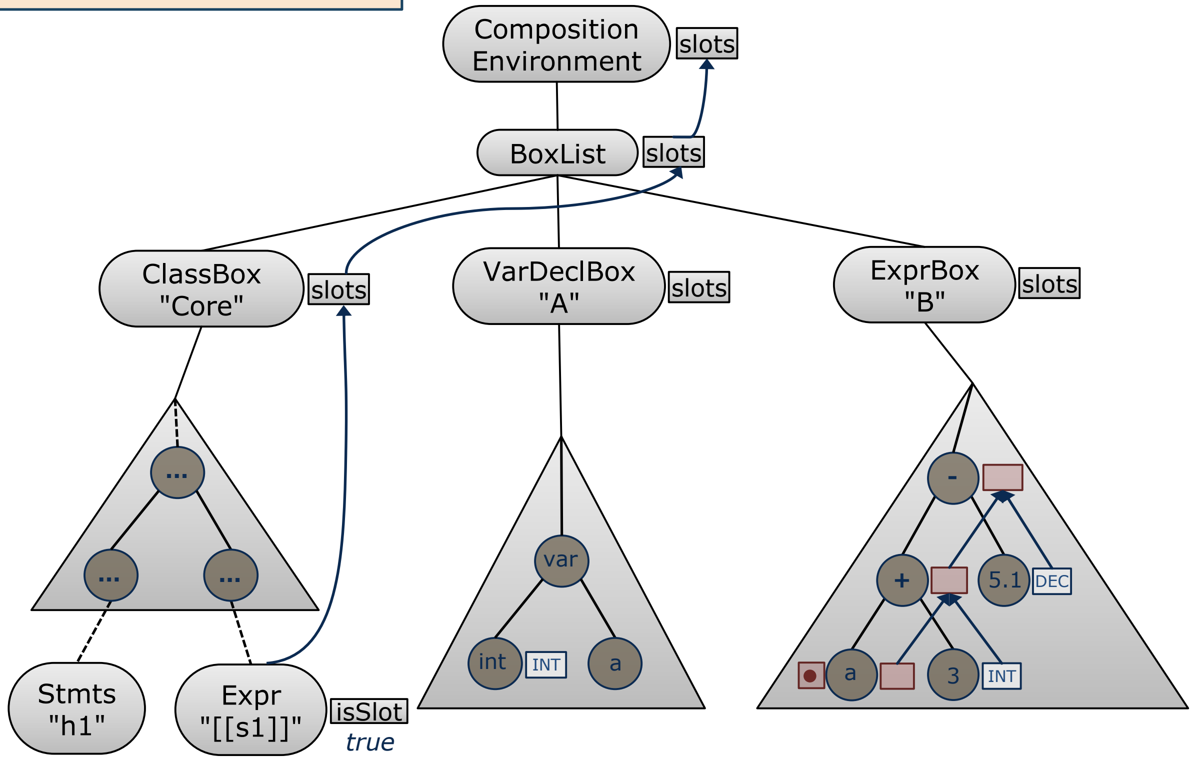
Slot *collection*

$\mathbf{syn}\ \texttt{Node}\star\ \{n \mid n \in N\}.\texttt{slots}$

$$\mathbf{fun}\ \{n \mid n \in N\}.\texttt{slots} = \begin{cases} \{\mathit{node}\} & \mathbf{if}\ \texttt{isSlot} = \mathit{true}, \\ \bigcup\limits_{c \in \texttt{child}_{\texttt{all}}} c.\texttt{slots} & \mathbf{else}. \end{cases}$$

RAG-based component models: compositional points

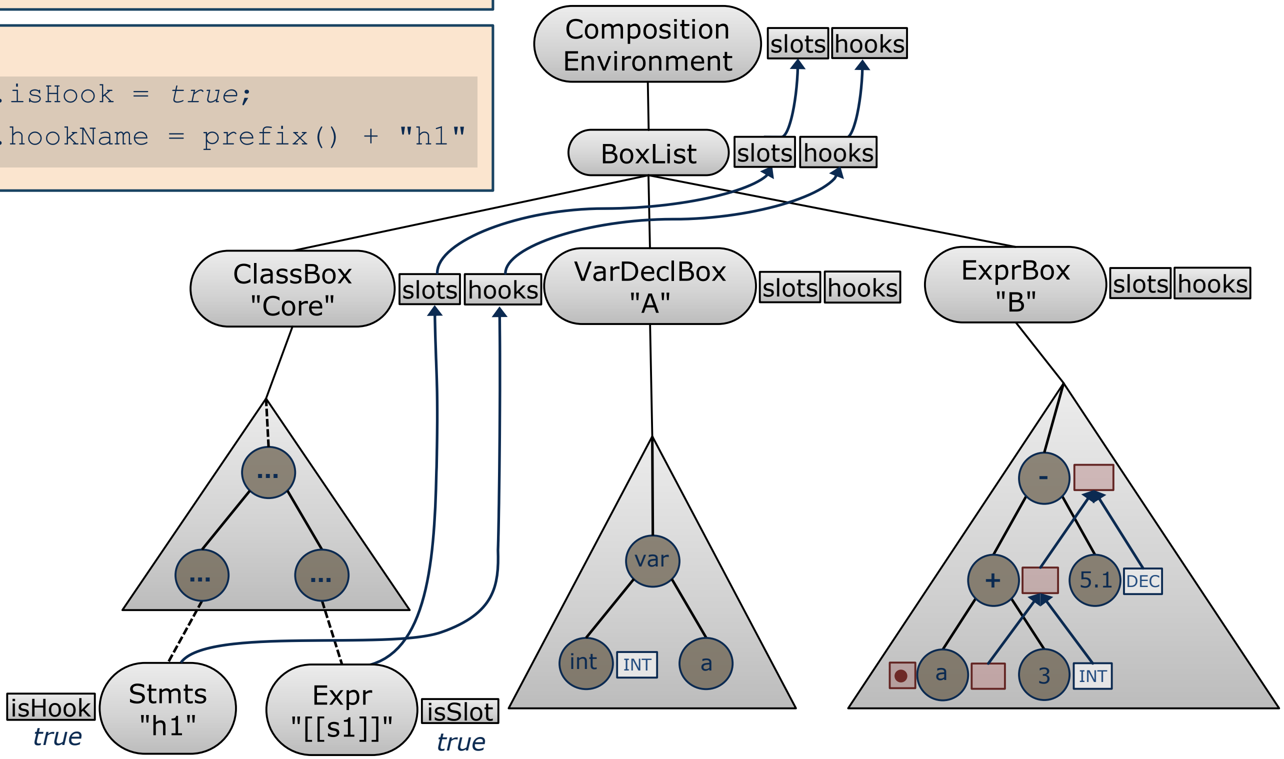
```
aspect slots
fun Expr.isSlot = node.isHedged("[[", "]]")
fun Expr.slotName = node.extractName()
```



RAG-based component models: compositional points

```
aspect slots
fun Expr.isSlot = node.isHedged("[[", "]]")
fun Expr.slotName = node.extractName()
```

```
aspect hooks
fun Block.Stmts.isHook = true;
fun Block.Stmts.hookName = prefix() + "h1"
```



RAG-based composition: composition operators

- The environment is extended with composer nonterminals:
  - Bind, Extend (and Extract: deletion)
- Attributes take over point matching and fragment provision

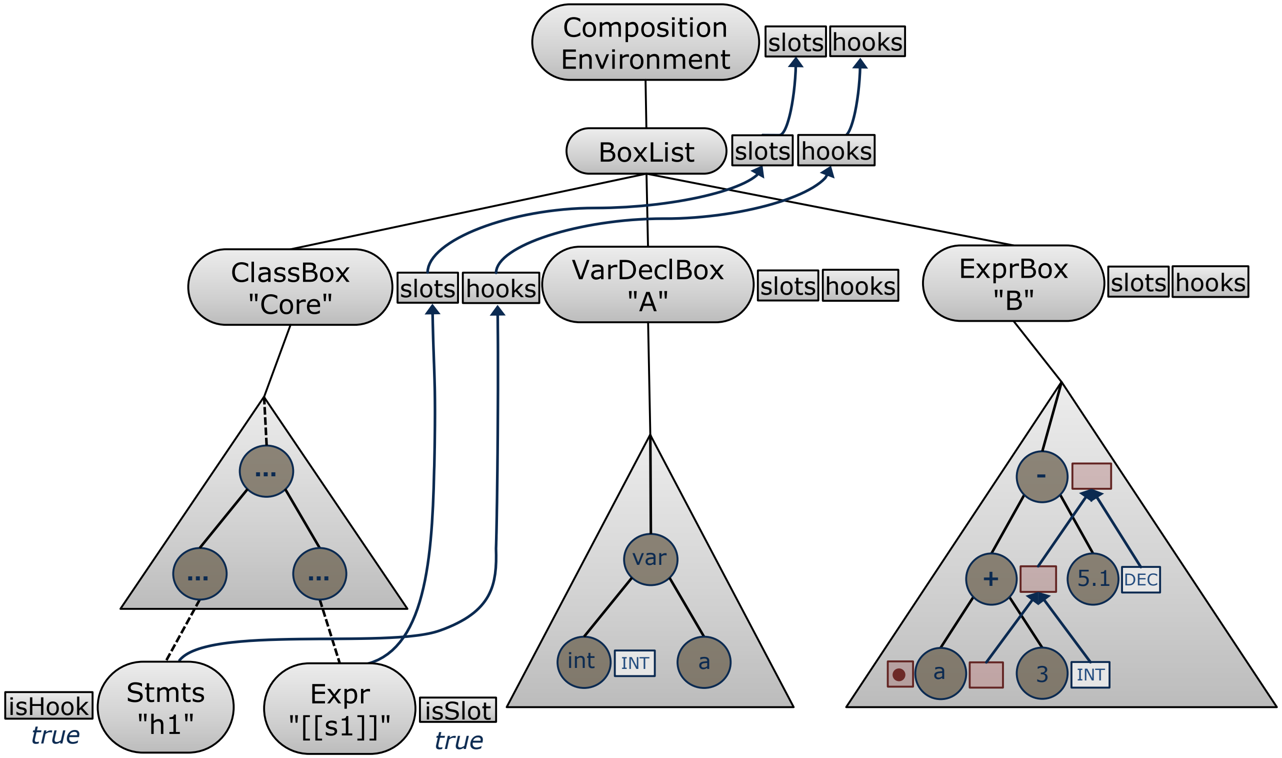
Environment *extension*

```
CompositionEnvironment ::=
              fragments:Box* composers:Composer*
@Composer ::= pointName:<string> fragmentName:<string>
Bind▷Composer ::=
Extend▷Composer ::= position:<int>
Extract▷Composer ::=
```

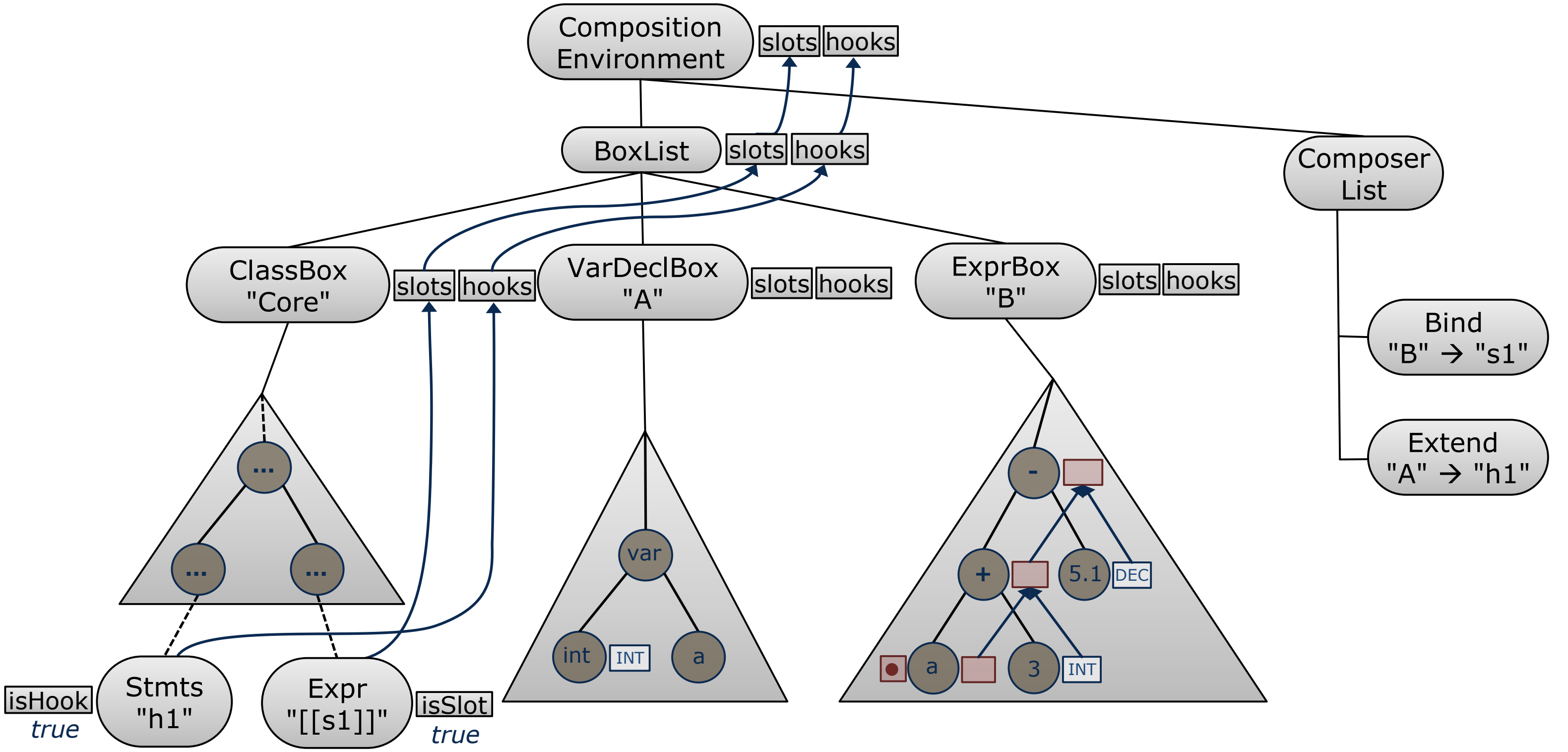
Point/fragment *lookup*

```
syn Node {Bind,Extend}.srcFragment
syn Node* Composer.points
```

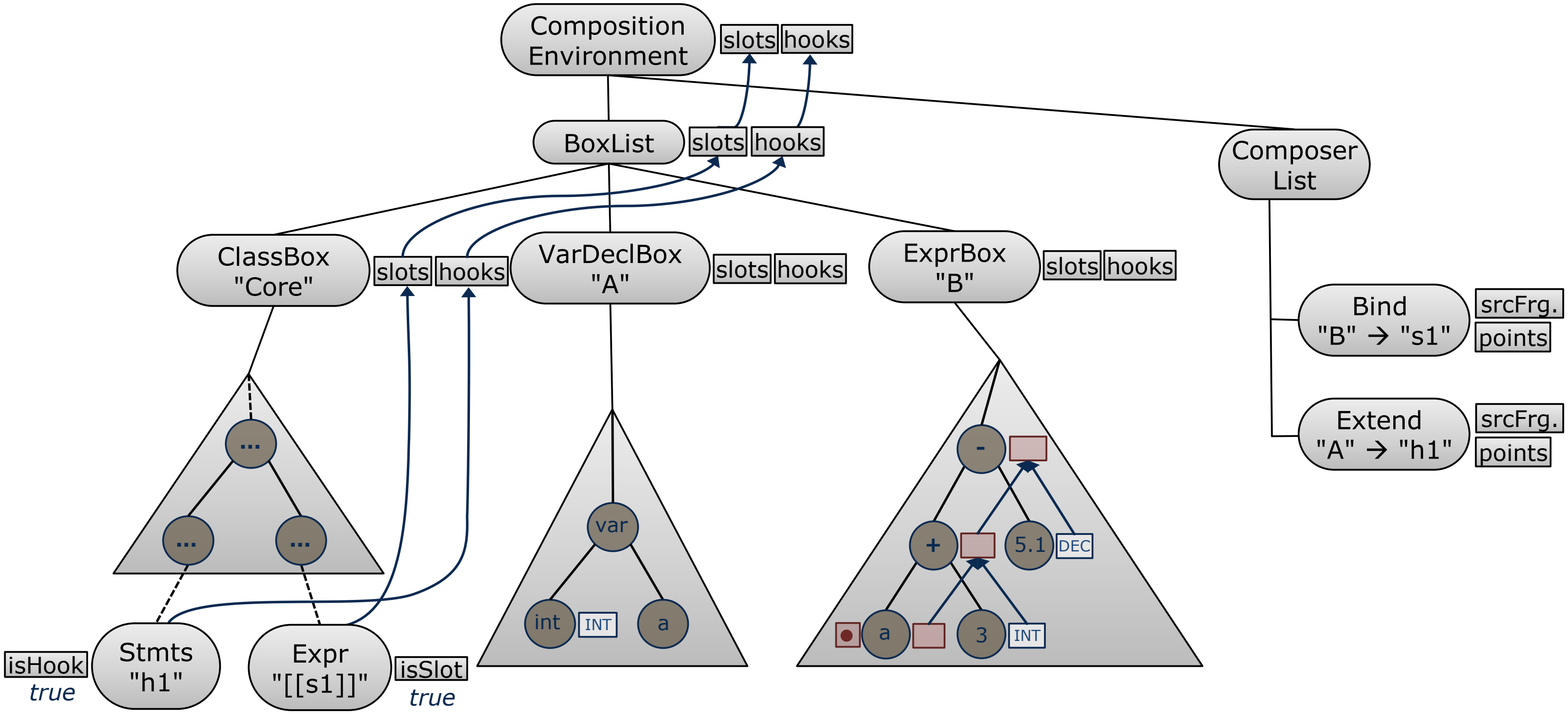
RAG-based composition: composition program



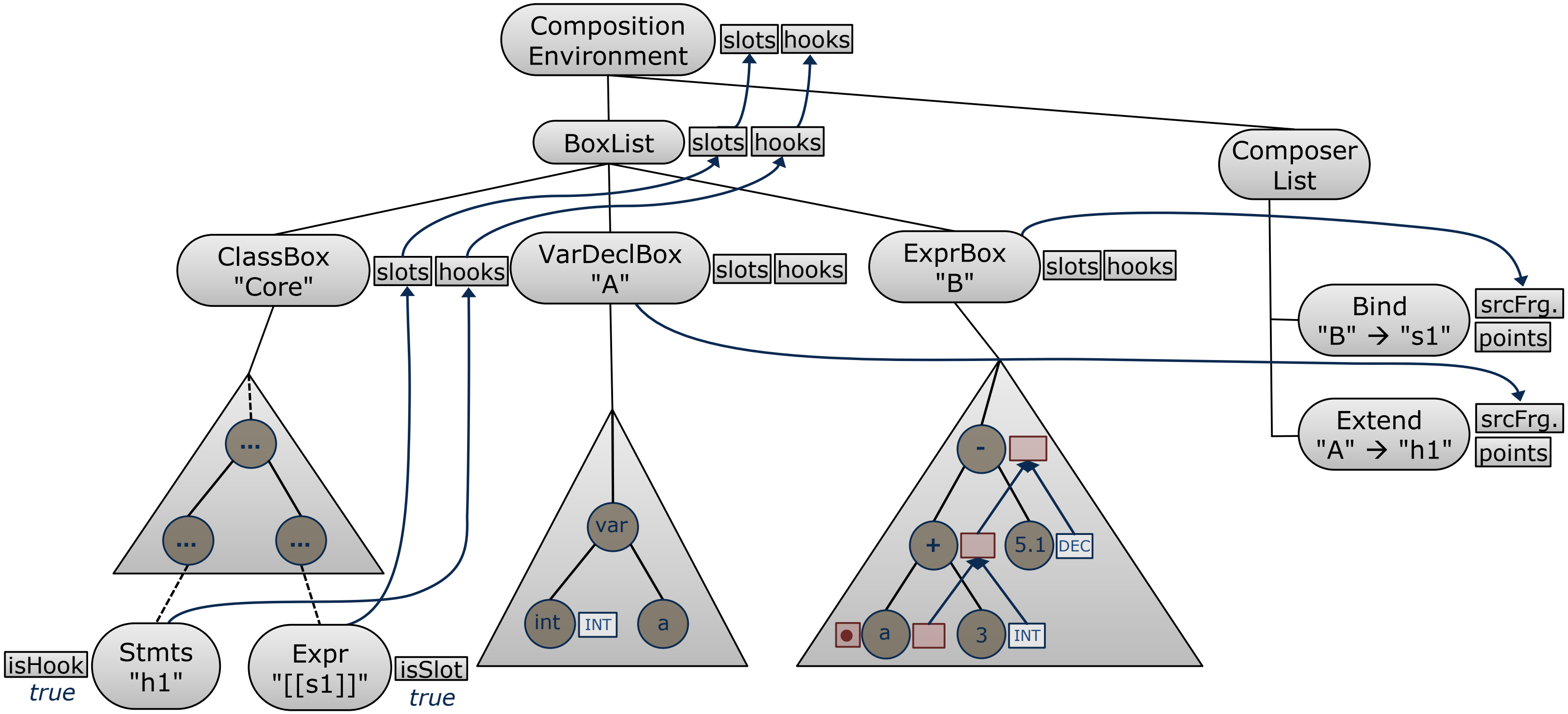
RAG-based composition: composition program



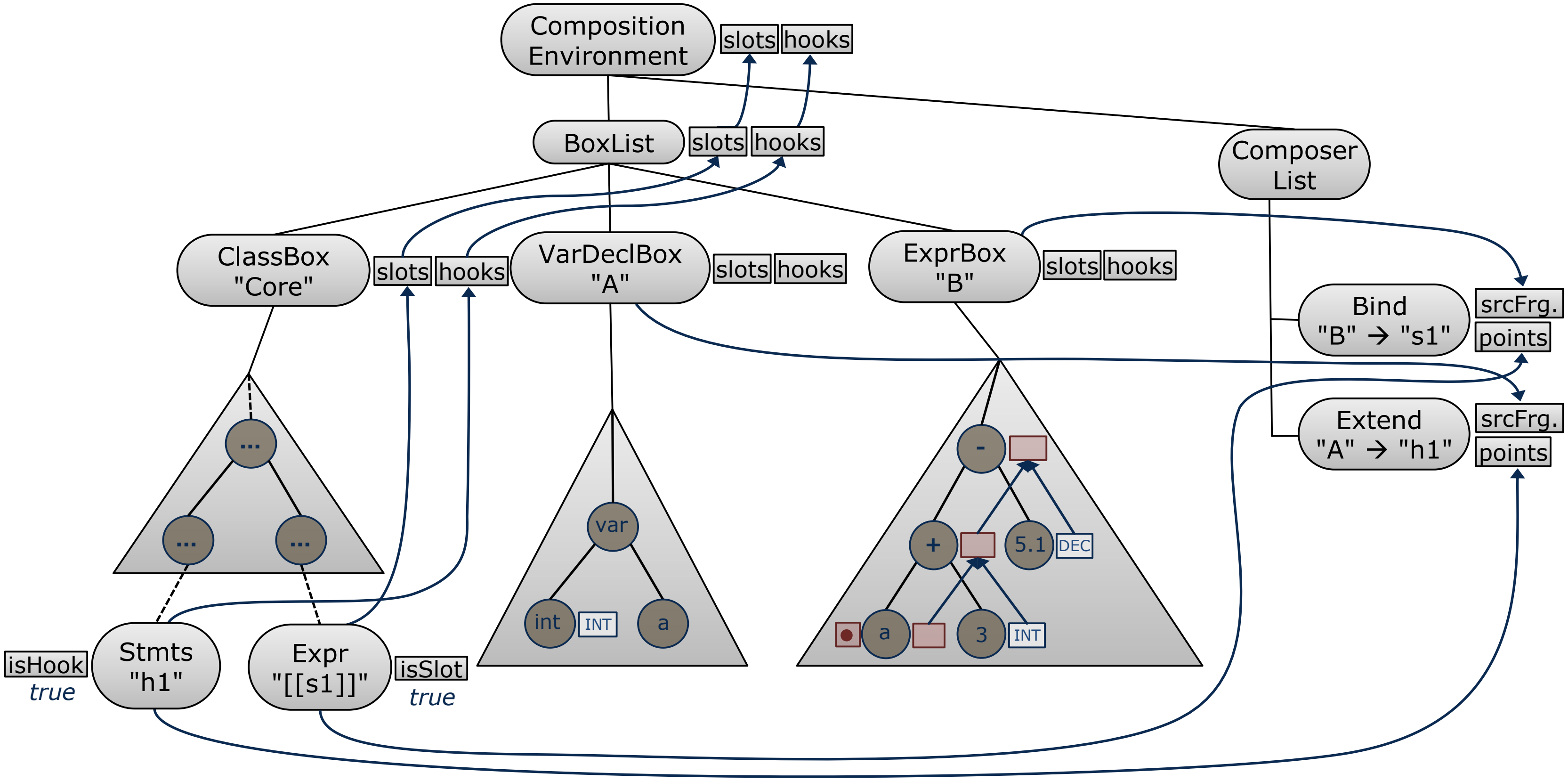
RAG-based composition: composition program



RAG-based composition: composition program



RAG-based composition: composition program

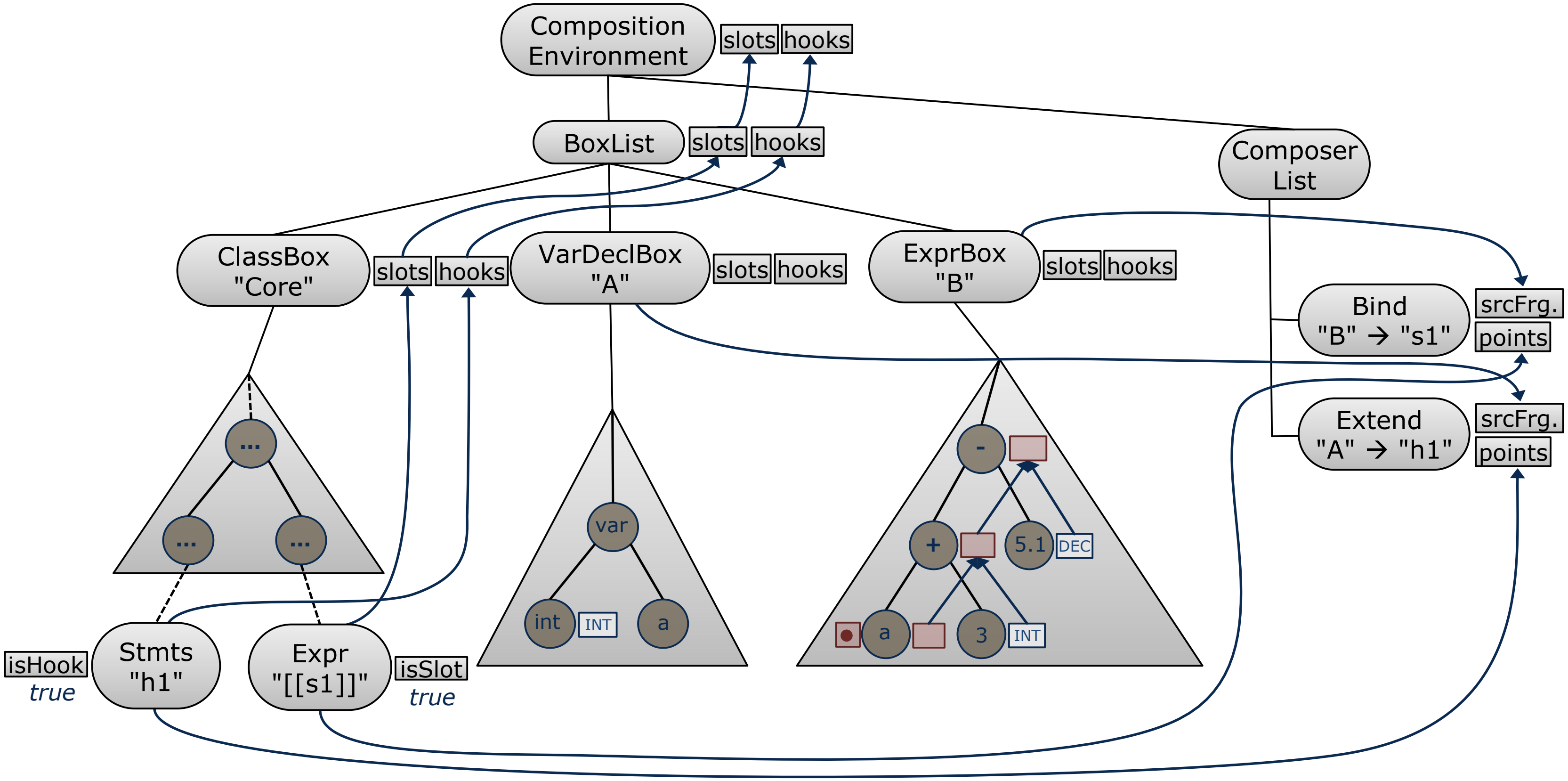


RAG-based composition: composition strategies

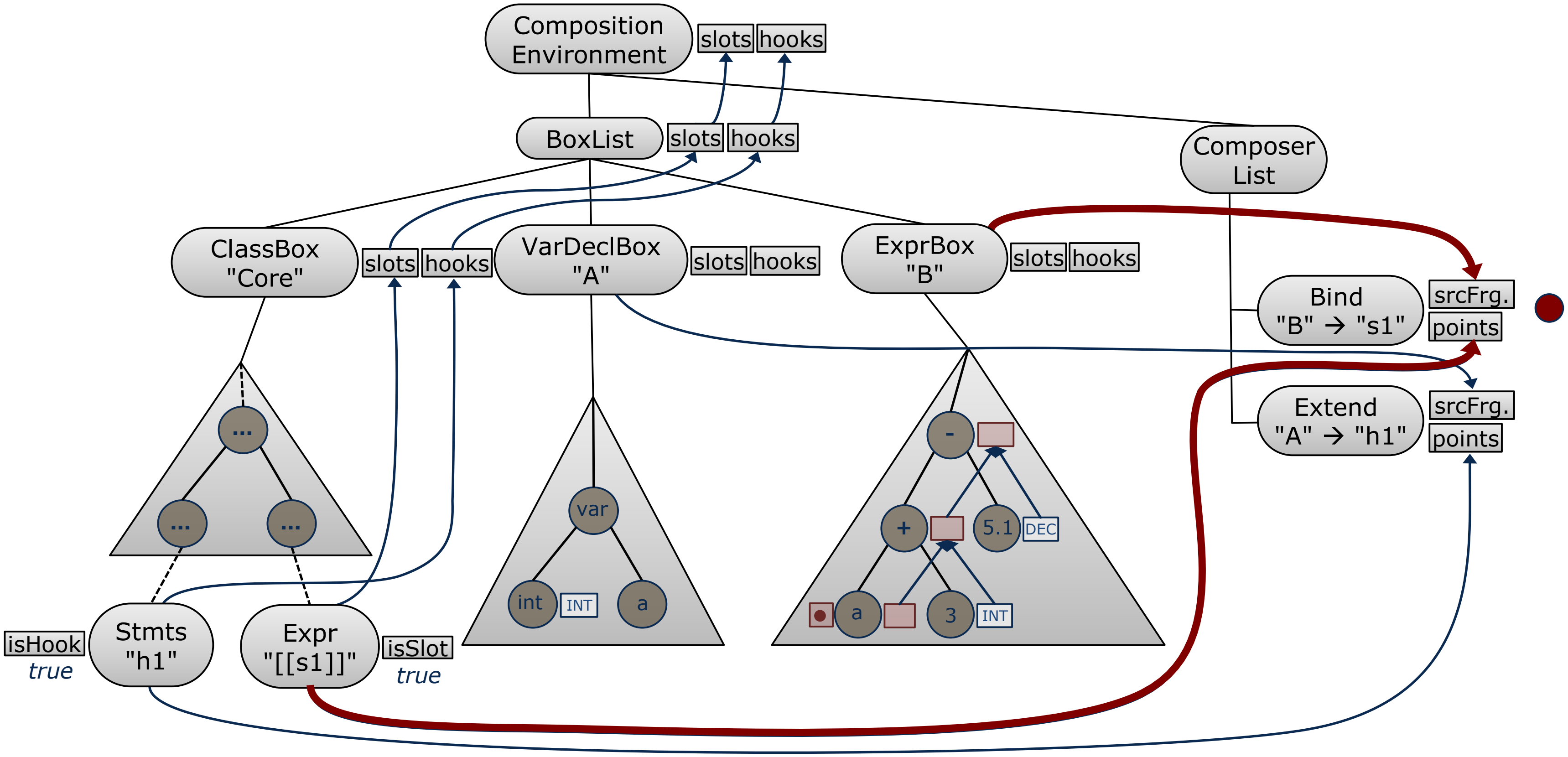
- Strategies help users managing their composition problems
- Well-formed ISC comprises three basic strategies:

1. **Operator-determined composition**
   - composer declarations: a set of rewrite rules
   - fix-point iteration
   - application order: by occurrence in environment, kind of operator or analysis based
2. **Point-determined composition**
   - composer declarations: a set of advices (cf. [Kiczales+01])
   - depth-first traversal over fragments
   - at points: look up composers and perform composition
3. **Attribute-determined composition**
   - interleaved attribute evaluation and composition (cf. ReRAGs [Ekman06])
   - fragment traversal induced by attribute dependencies
     - problematic: result depends on where evaluation starts

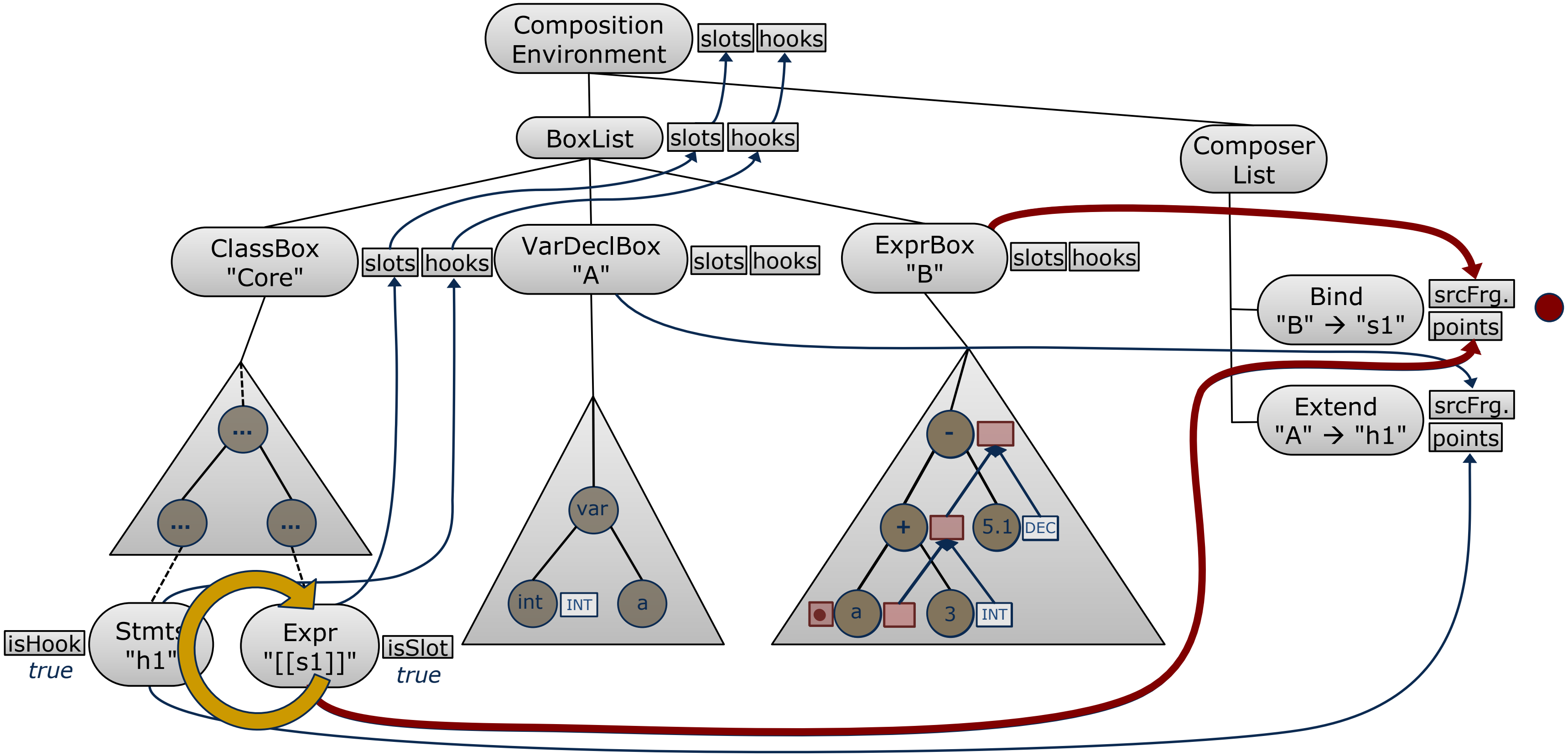
Strategy example: operator-determined composition



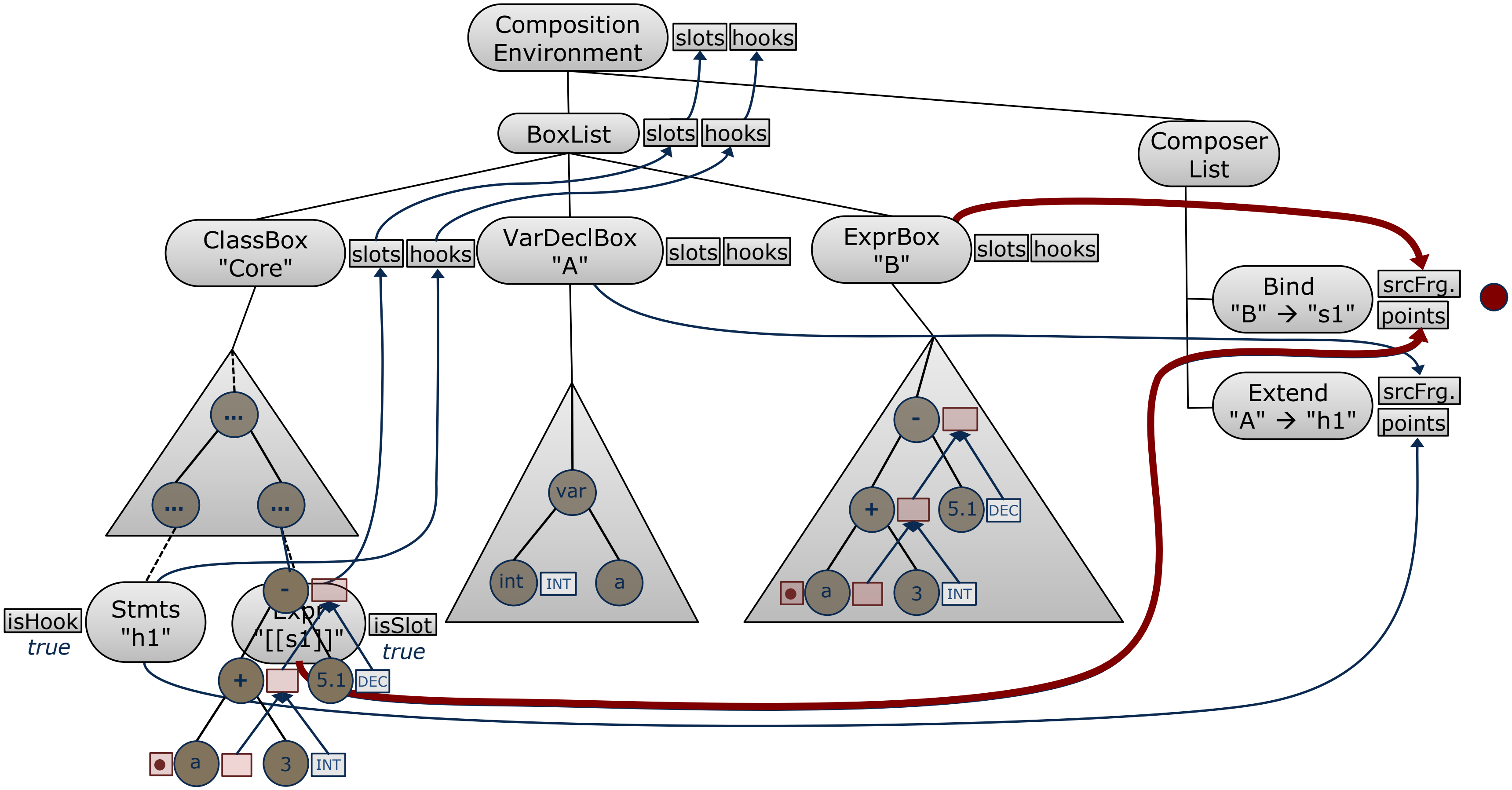
Strategy example: operator-determined composition



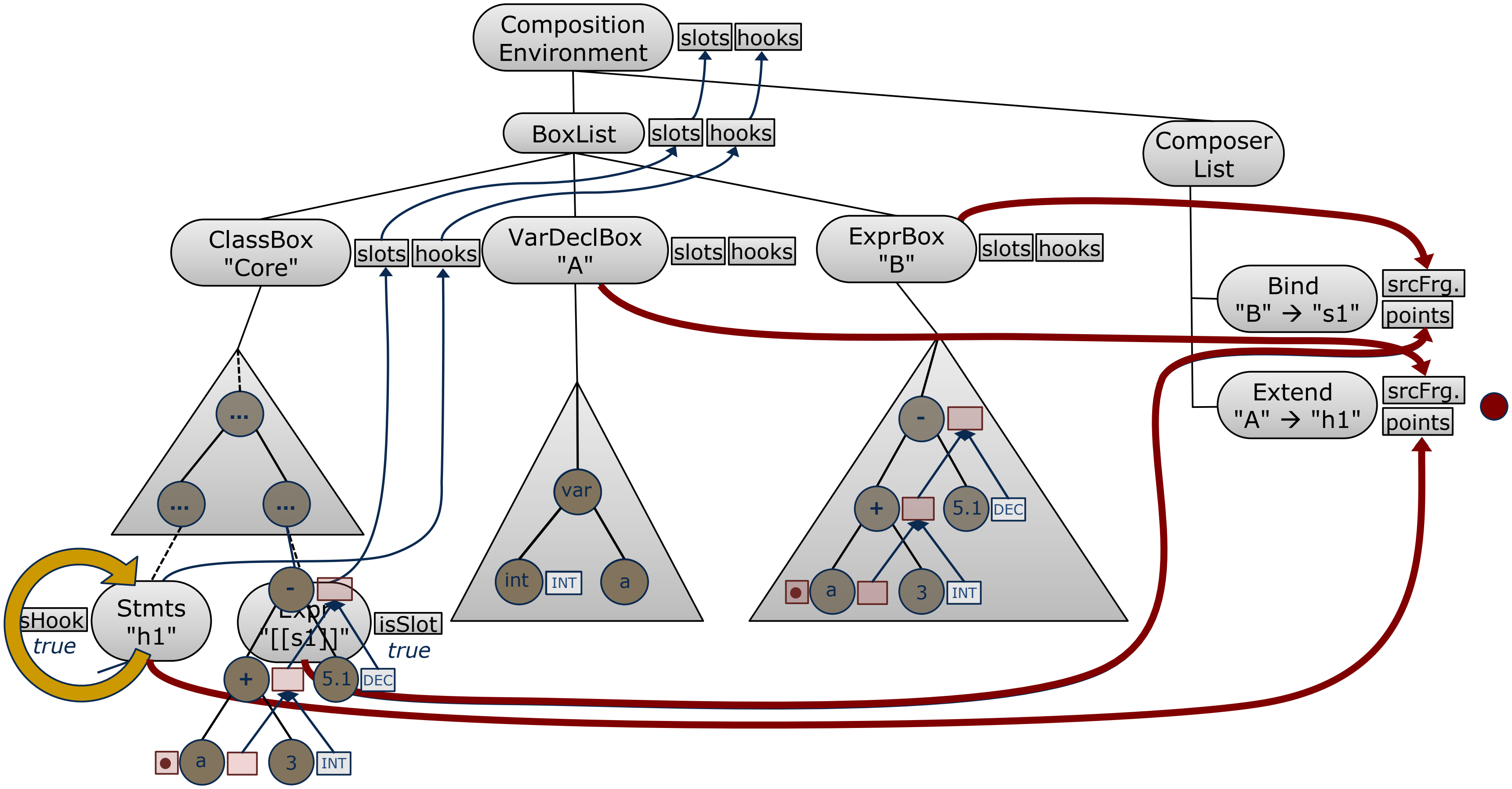
Strategy example: operator-determined composition



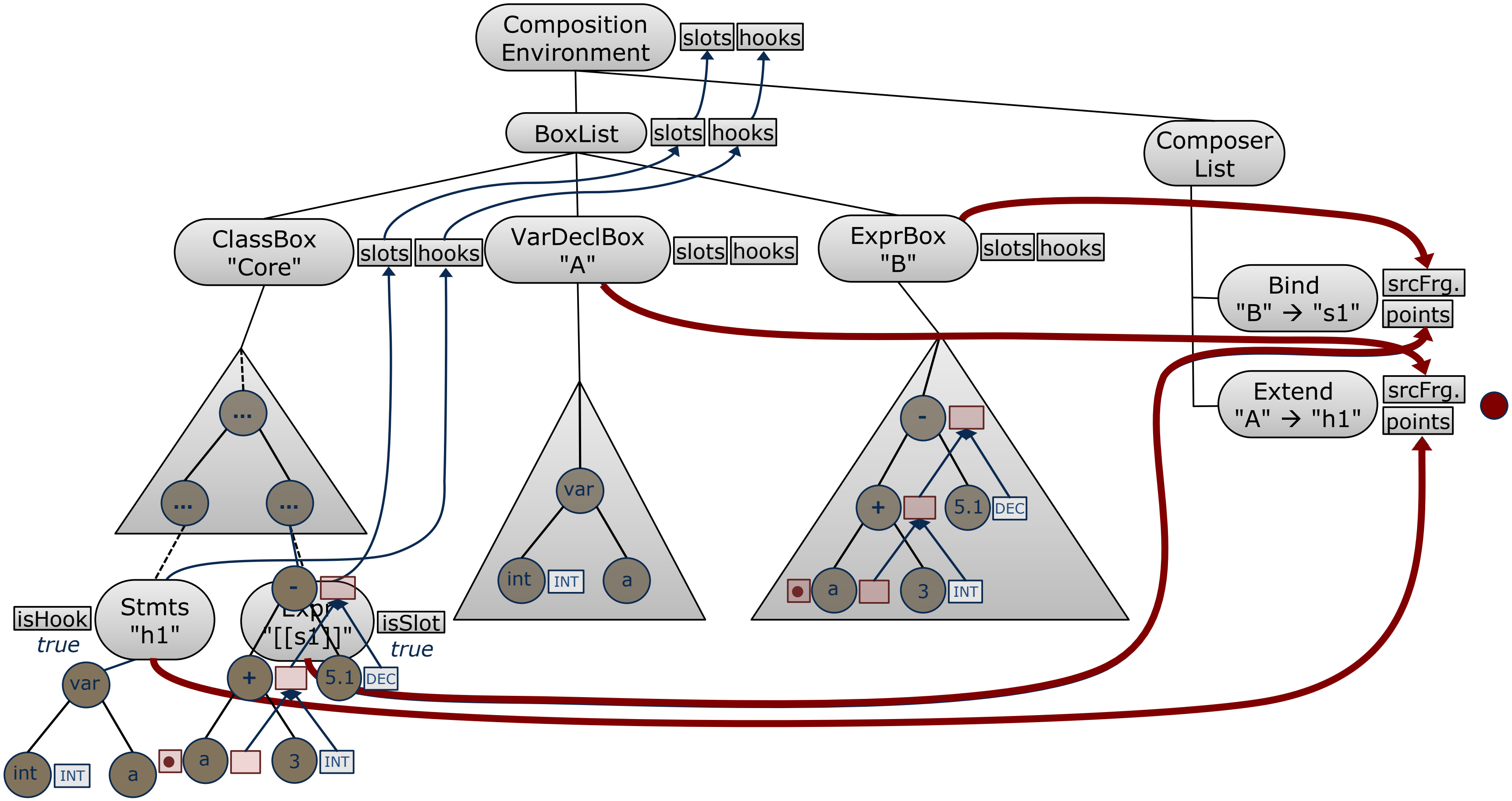
Strategy example: operator-determined composition



Strategy example: operator-determined composition



Strategy example: operator-determined composition



RAG-based composition: static fragment contracts

- *Contracts*: pre- and postconditions (assertions) of a method [Meyer92]
- *Fragment contracts*: pre- and postconditions of composition(s)
  - *fragment assertions* over *characteristic attributes*
  - ensuring fragment compatibility w.r.t. static semantics and other constraints
  - locate semantic errors in composition programs
  - automatically select a compatible fragment during composition

Characteristic attributes of compositional points *n*:

$$\textbf{syn}\ \kappa\ \{n \mid \mathit{prop} \in \mathrm{Ass}_x(n)\}\,.\,\mathit{prop}$$

$$\textbf{fun}\ n_i\,.\,\mathit{prop} = \begin{cases} \perp & \textbf{if}\ \texttt{isSlot} = \texttt{isHook} = \mathit{false} \\ \kappa\texttt{-}\mathit{expression} & \textbf{else}. \end{cases}$$

Characteristic attributes:

$$\textbf{syn}\ \kappa\ \{f \mid \mathit{prop} \in \mathrm{Ass}_x(f)\}\,.\,\mathit{prop}$$

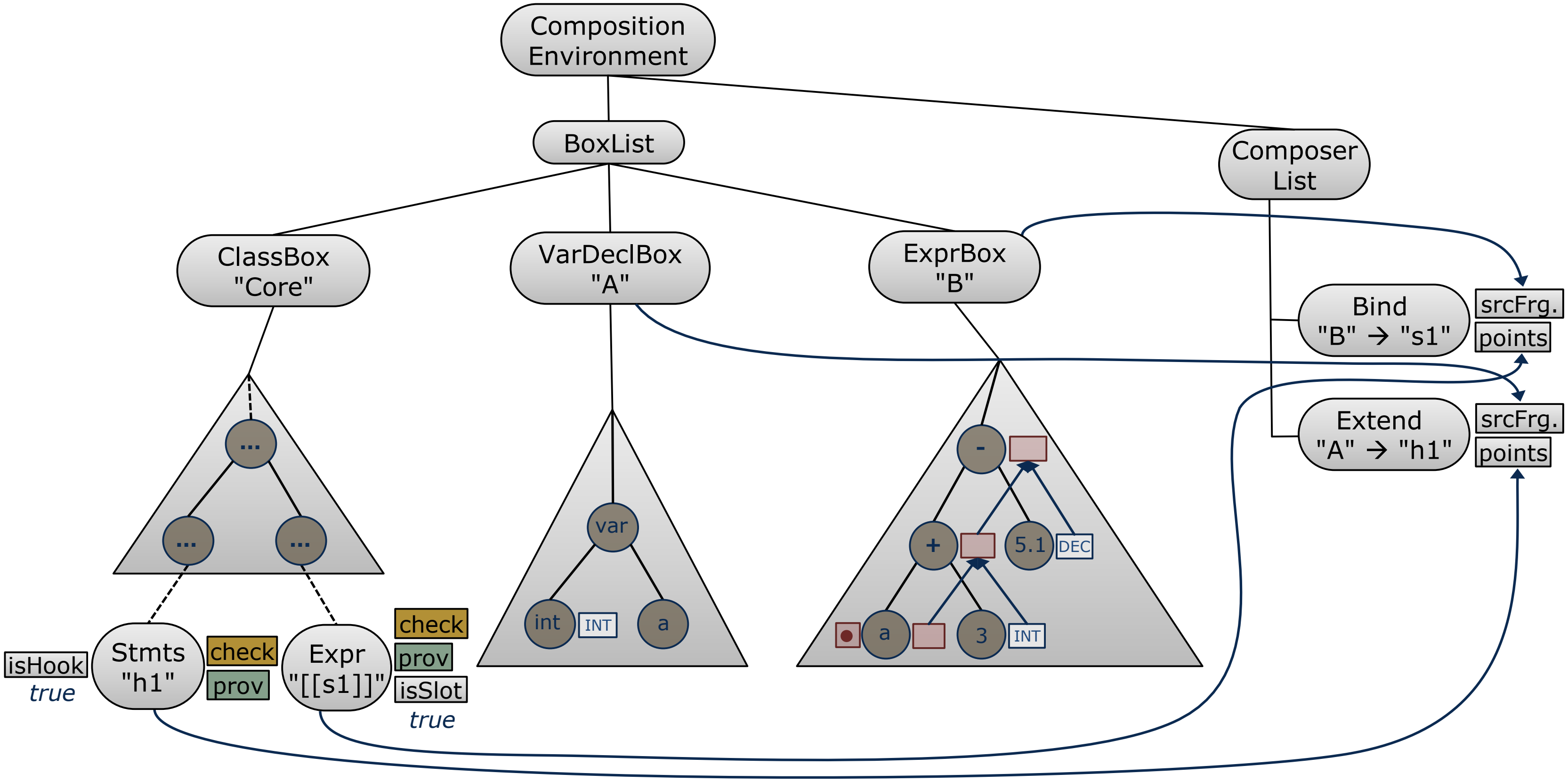
$$\textbf{fun}\ f\,.\,\mathit{prop} = \kappa\texttt{-}\mathit{expression}$$

Fragment assertion:

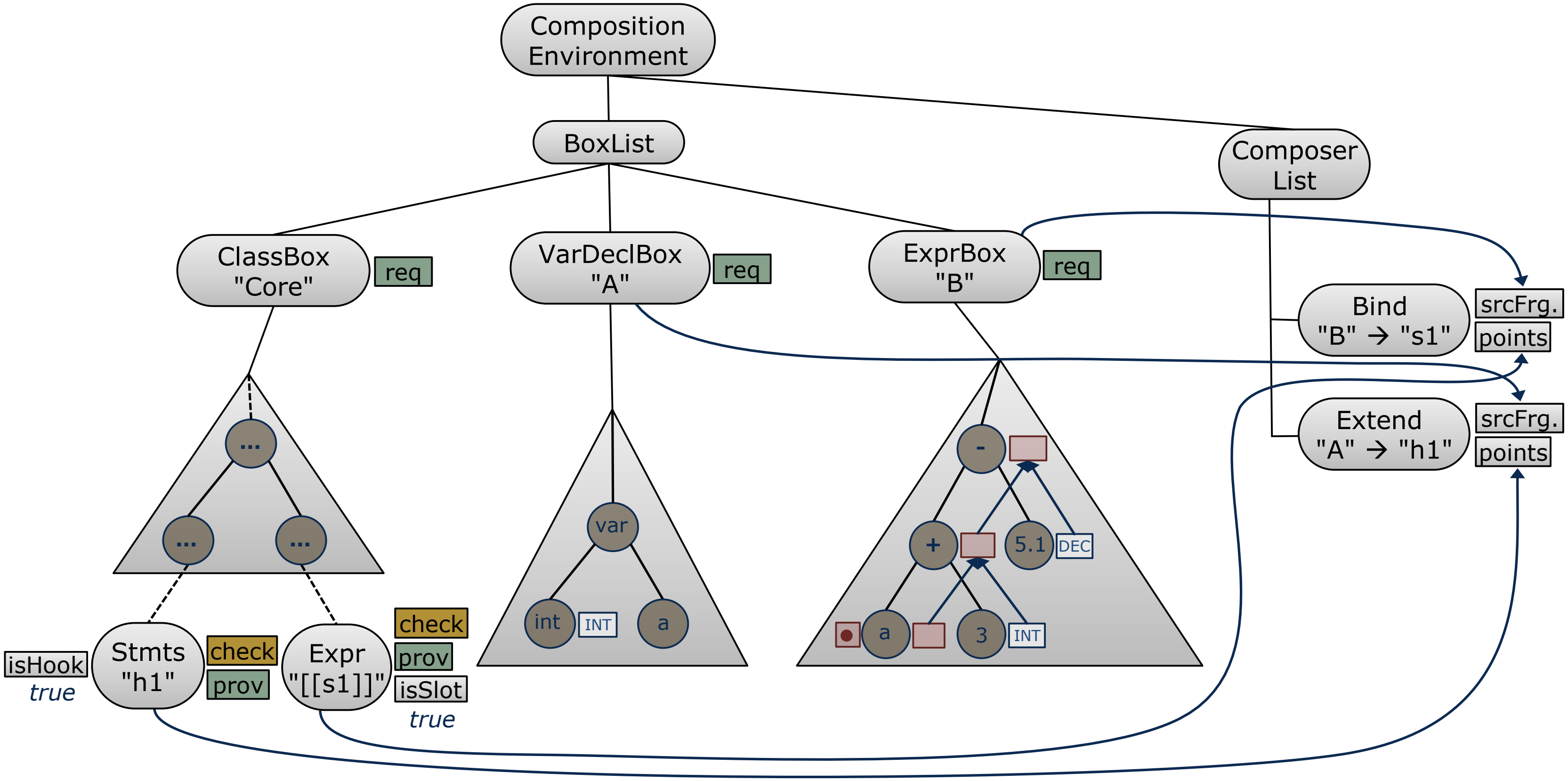
$$\textbf{syn}\ \texttt{bool}_\perp\ \{n \mid n \in \mathcal{S} \cup \mathcal{H}\}.\texttt{check}\,(\texttt{Node})$$

$$\textbf{fun}\ n_i.\texttt{check}\,(\texttt{fr}) = \begin{cases} \perp & \textbf{if}\ \texttt{isSlot} = \texttt{isHook} = \mathit{false}, \\ \mathit{true} & \textbf{if}\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) = \emptyset, \\ \mathit{bool\texttt{-}expr\ on} & \textbf{if}\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) \neq \emptyset. \\ \mathrm{Ass}_x(n) \cup \mathrm{Ass}_x(\mathrm{Lab}(\texttt{fr})) & \end{cases}$$

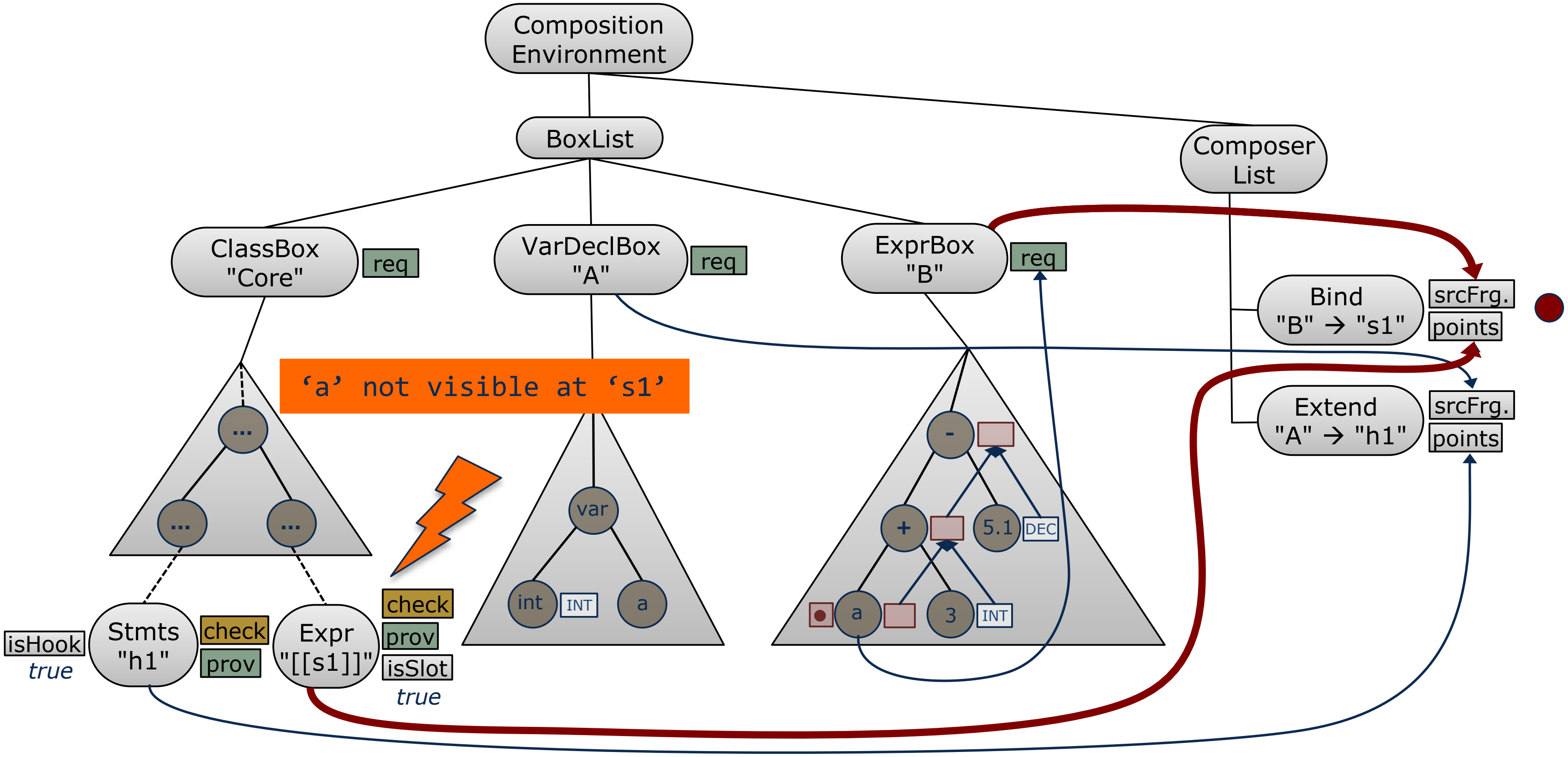
Example : operator-determined composition with contract



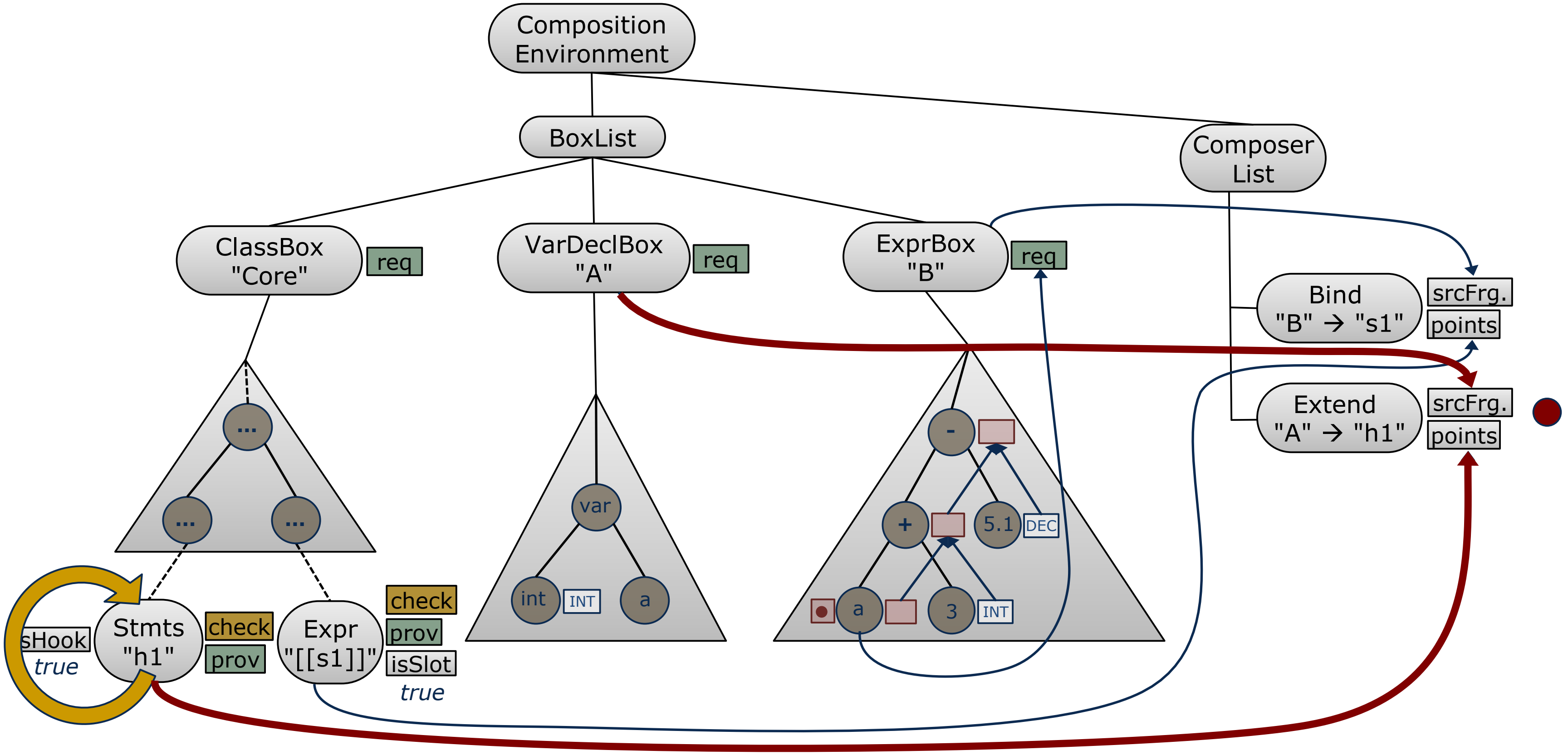
Example : operator-determined composition with contract



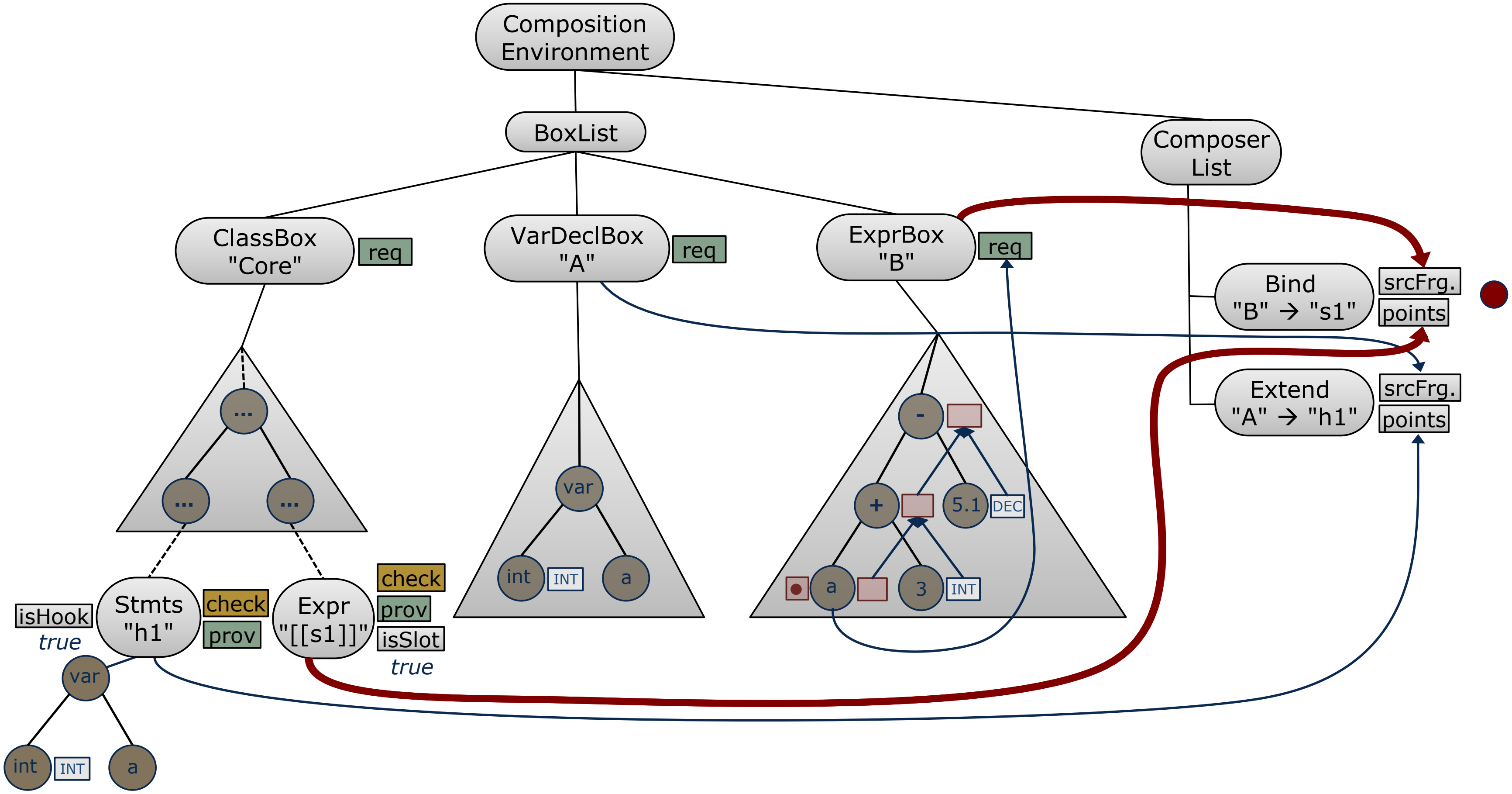
Example : operator-determined composition with contract



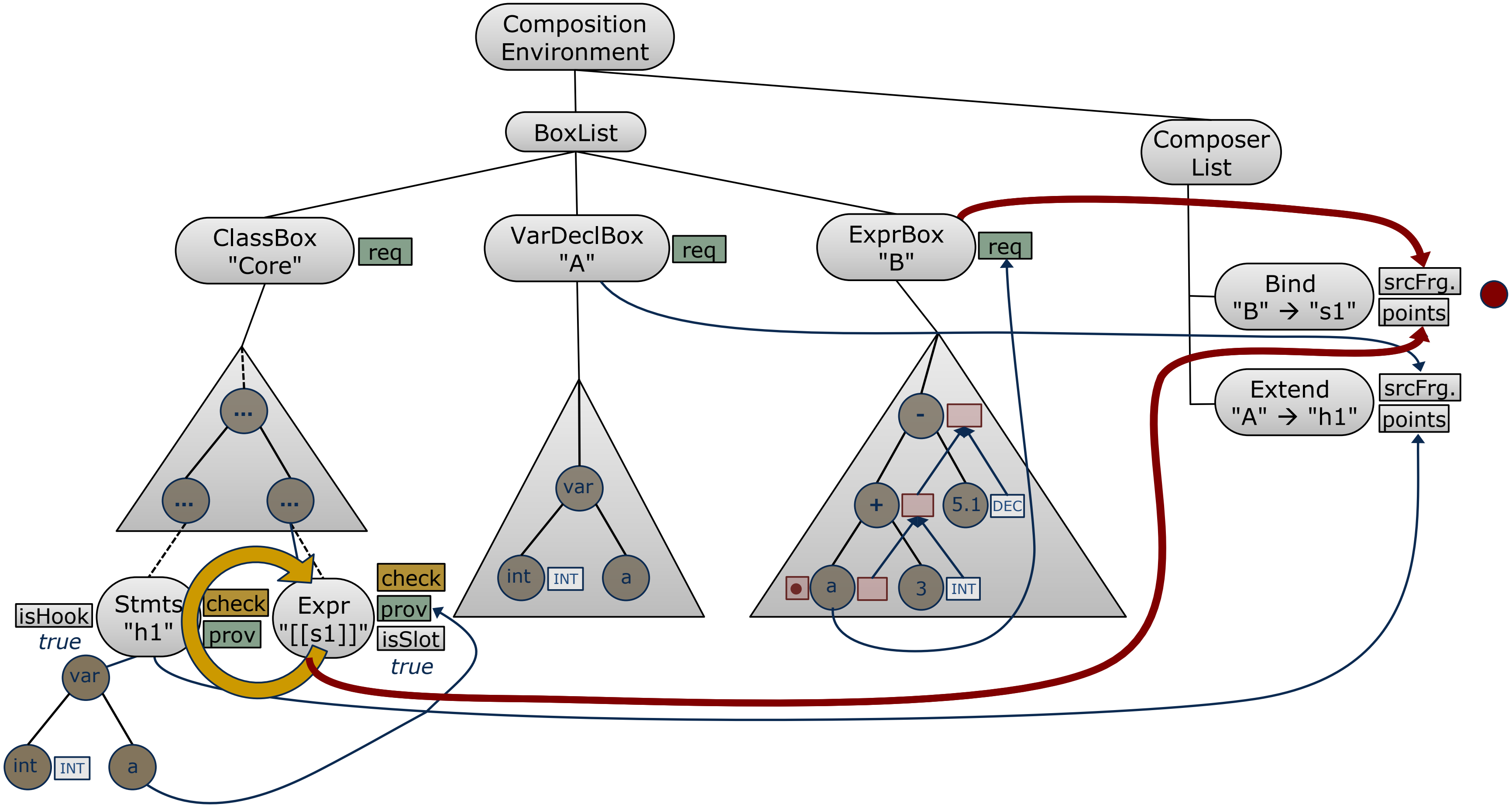
Example : operator-determined composition with contract



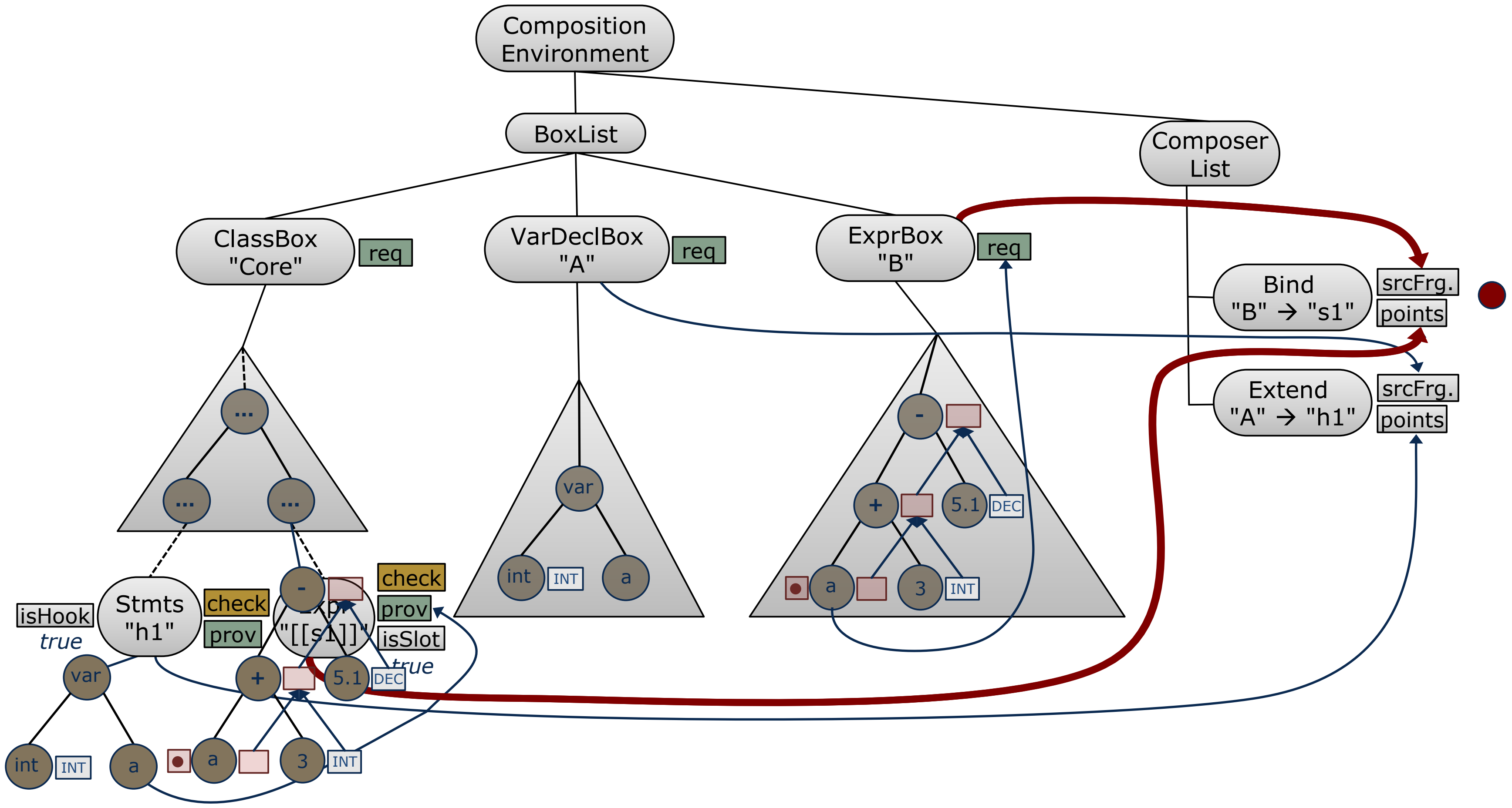
Example : operator-determined composition with contract



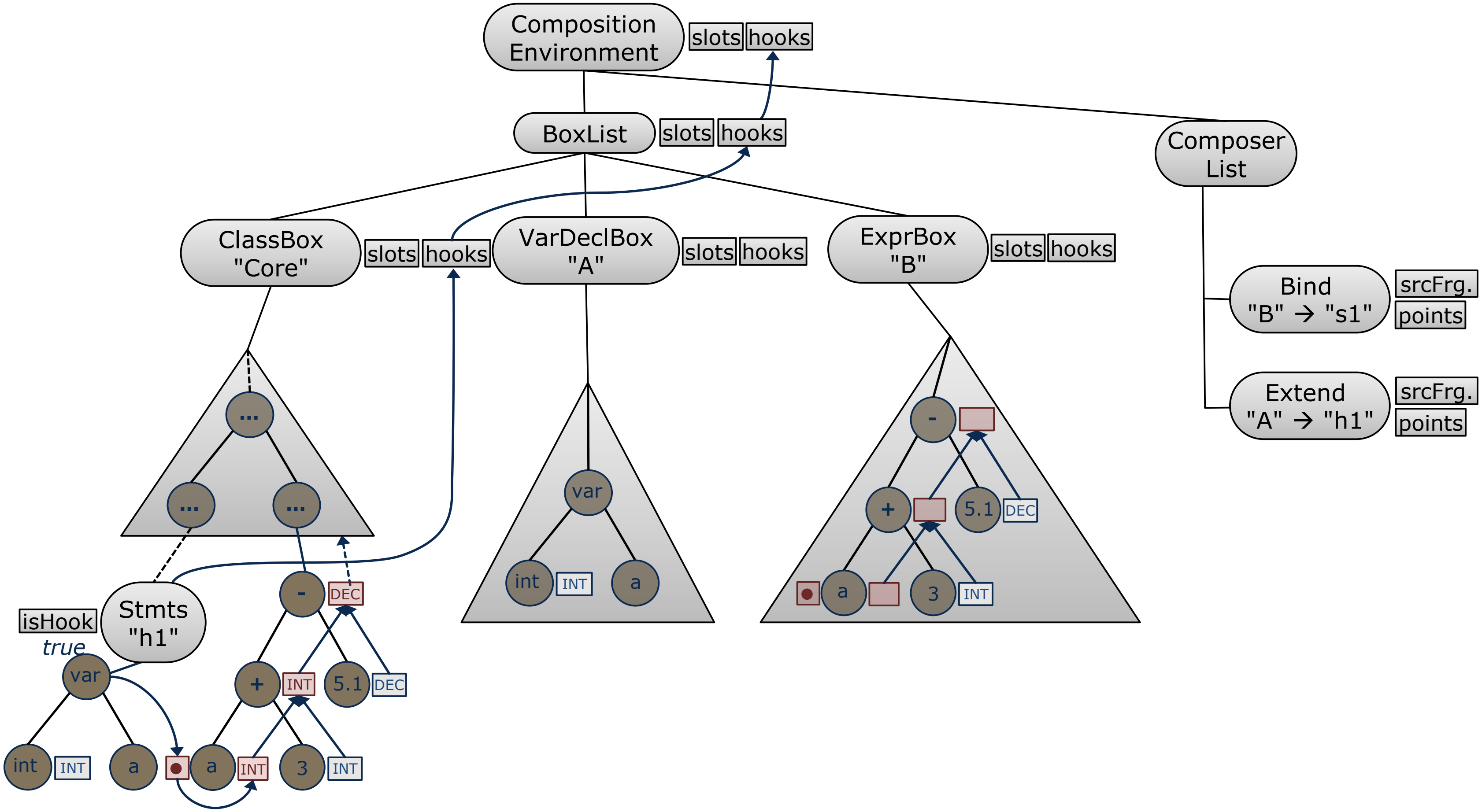
Example : operator-determined composition with contract



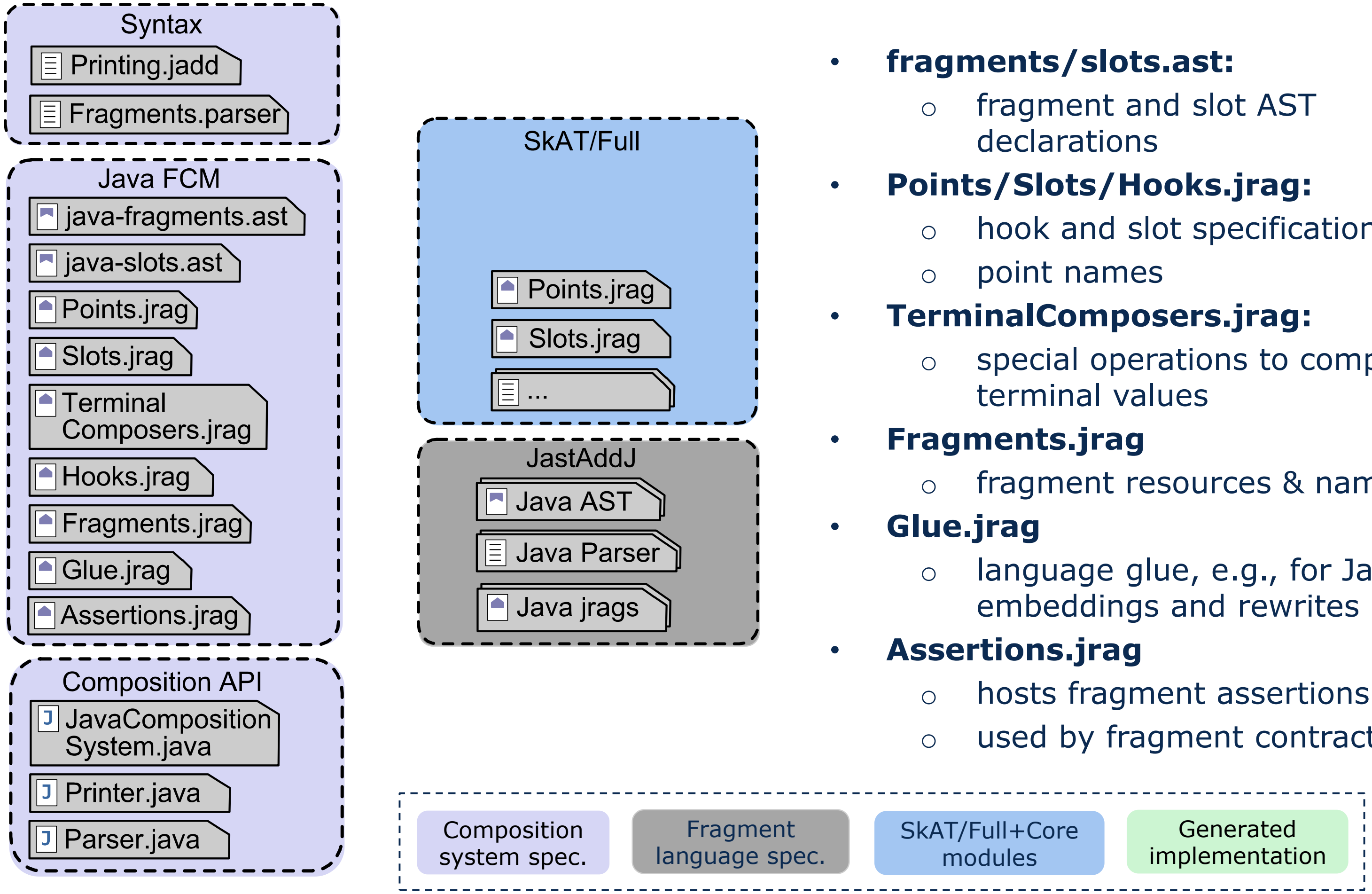
Example : operator-determined composition with contract



Example : operator-determined composition with contract



SkAT4J: a well-formed composition system for Java



Syntax
- Printing.jadd
- Fragments.parser

Java FCM
- java-fragments.ast
- java-slots.ast
- Points.jrag
- Slots.jrag
- Terminal Composers.jrag
- Hooks.jrag
- Fragments.jrag
- Glue.jrag
- Assertions.jrag

Composition API
- JavaComposition System.java
- Printer.java
- Parser.java

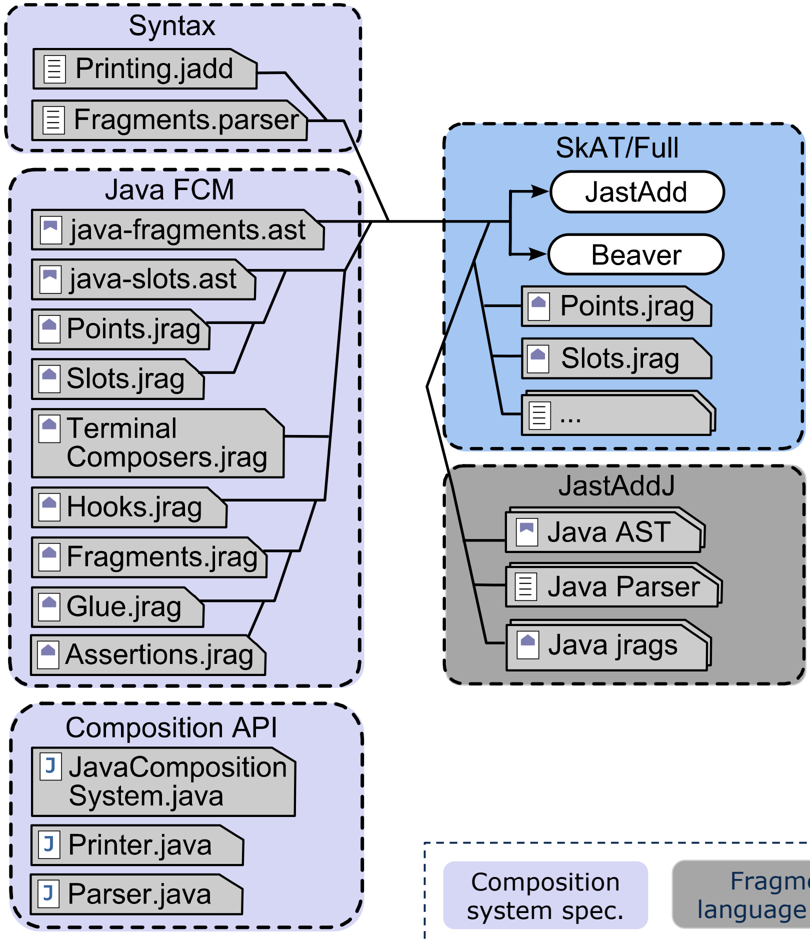
SkAT/Full
- Points.jrag
- Slots.jrag
- …

JastAddJ
- Java AST
- Java Parser
- Java jrags

Composition system spec. | Fragment language spec. | SkAT/Full+Core modules | Generated implementation

- **fragments/slots.ast:**
  - fragment and slot AST declarations
- **Points/Slots/Hooks.jrag:**
  - hook and slot specifications
  - point names
- **TerminalComposers.jrag:**
  - special operations to compose terminal values
- **Fragments.jrag**
  - fragment resources & naming
- **Glue.jrag**
  - language glue, e.g., for Java embeddings and rewrites
- **Assertions.jrag**
  - hosts fragment assertions
  - used by fragment contracts

SkAT4J: a well-formed composition system for Java



Syntax
- Printing.jadd
- Fragments.parser

Java FCM
- java-fragments.ast
- java-slots.ast
- Points.jrag
- Slots.jrag
- Terminal Composers.jrag
- Hooks.jrag
- Fragments.jrag
- Glue.jrag
- Assertions.jrag

Composition API
- JavaComposition System.java
- Printer.java
- Parser.java

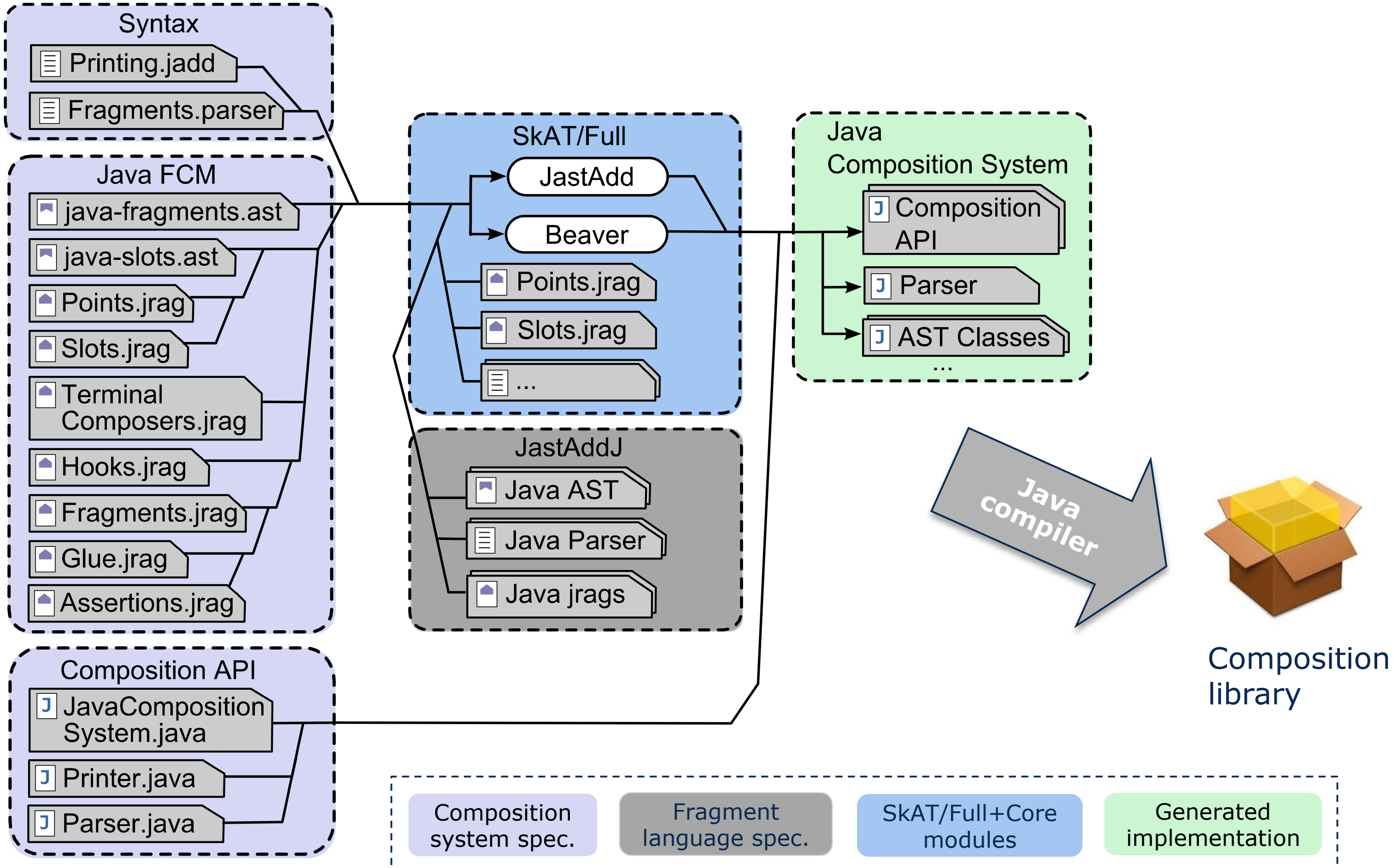
SkAT/Full
- JastAdd
- Beaver
- Points.jrag
- Slots.jrag
- ...

JastAddJ
- Java AST
- Java Parser
- Java jrags

- **fragments/slots.ast:**
  - fragment boxes and extra slot declarations
- **Points/Slots/Hooks.jrag:**
  - hook and slot specifications
  - point names
- **TerminalComposers.jrag:**
  - special operations to compose terminal values
- **Fragments.jrag**
  - fragment resources & naming
- **Glue.jrag**
  - language glue, e.g., for Java embeddings and rewrites
- **Assertions.jrag**
  - hosts fragment assertions
  - used by fragment contracts

Composition system spec.	Fragment language spec.	SkAT/Full+Core modules	Generated implementation

SkAT4J: a well-formed composition system for Java



🟪 Composition system spec.
▨ Fragment language spec.
🟦 SkAT/Full+Core modules
🟩 Generated implementation

SkAT4J: used as a library

```
private JavaCompositionSystem cSys = new JavaCompositionSystem(“baf/","in","out");

public void compositionProgram(Model bm) throws IOException {

    cSys.setCompositionStrategy(CompositionSystem.OP_ORDERED_COMPOSITION_FP);

    cSys.setRecoverMode(true);

    for(RoleDefinition role: bm.getRoleDefinitions()){
        String cuName = role.getName() + ".java";
        cSys.copyBox("Person.jbx",cuName);
        cSys.addBind(cuName+"#Type",role.getName());
        cSys.addBind(cuName+"#TypeName","\"" + role.getName() + "\"");
        cSys.addBind(cuName+"#ImplicitSuperClass","Person");
        cSys.addExtend(cuName+"#*.membersEntry","public [[Type]](){super();}");
        cSys.addBindTerminal(cuName+"#Pfx","\n");
    }

    ...
    cSys.triggerComposition();

    cSys.persistFragments();
}
```

setup fragment location

configure strategy

configure contract validation

declare compositions

trigger execution

write back to file system

SkAT/Full & SkAT4J: evaluation

- Supports the complete model of “classic” ISC
- Component model specification based on JastAdd RAGs
- Adds contracts (*redeclarations*, *expression type check*, *variables and methods in scope*)
- Adds extensibility of component models
- Adds composition strategies and supports command mode

Applications of well-formed ISC in SkAT4J

- Code generator for the BAF scenario
- Well-formed mixin composer
- Skeleton fragment library for parallel programming [Cole 04, Goswami+02]

Outline

Part I: Overview

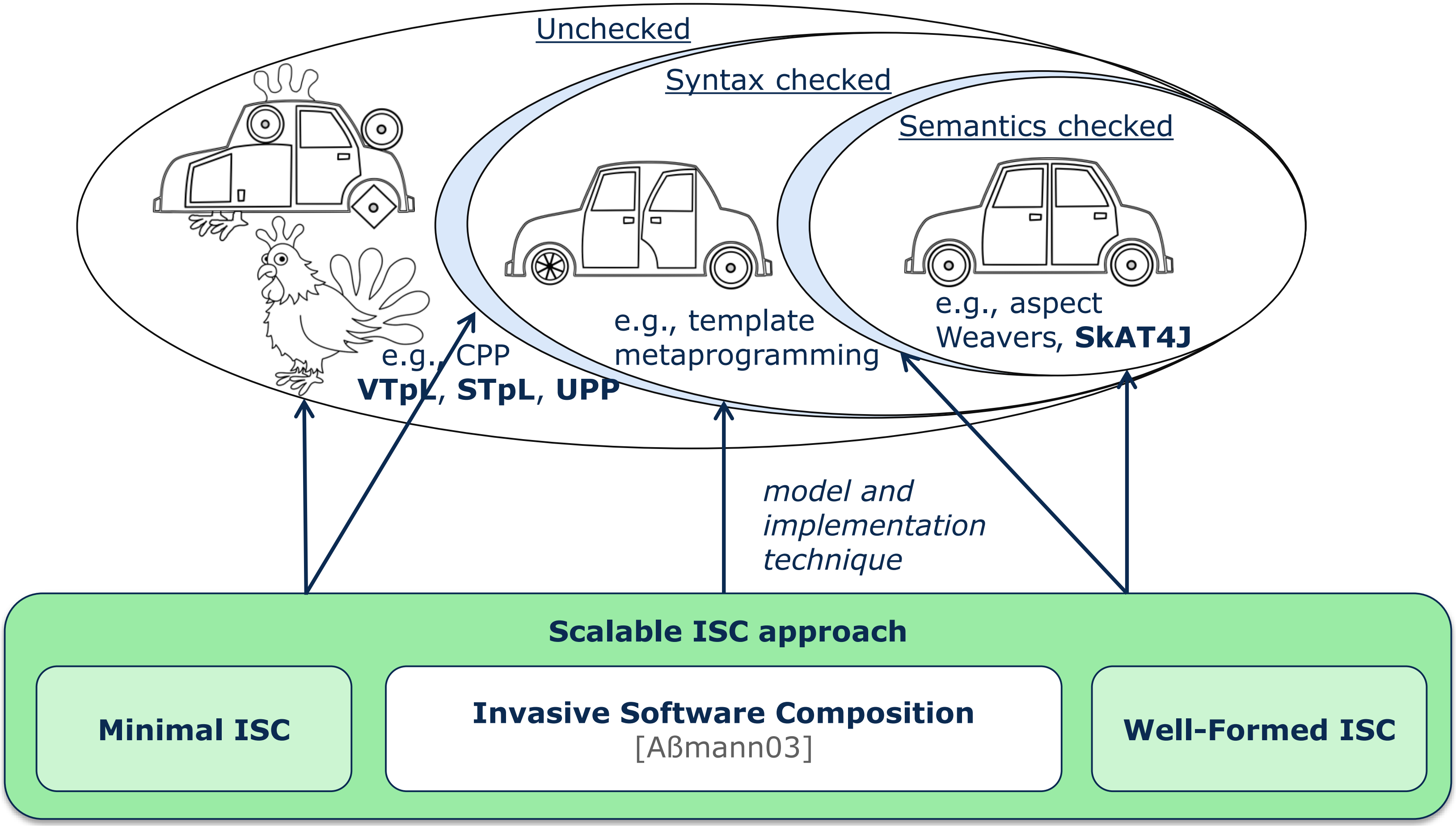
- Invasive Software Composition
- Problem Analysis
- Thesis Contributions

Part II: Well-Formed Invasive Software Composition

- Component Models with Attribute Grammars
- Composition Strategies
- Fragment Contracts
- Implementation in SkAT

Conclusion & Outlook

Thesis achievements – summary



Unchecked

Syntax checked

Semantics checked

e.g., CPP
VTpL, **STpL**, **UPP**

e.g., template metaprogramming

e.g., aspect Weavers, **SkAT4J**

model and implementation technique

Scalable ISC approach

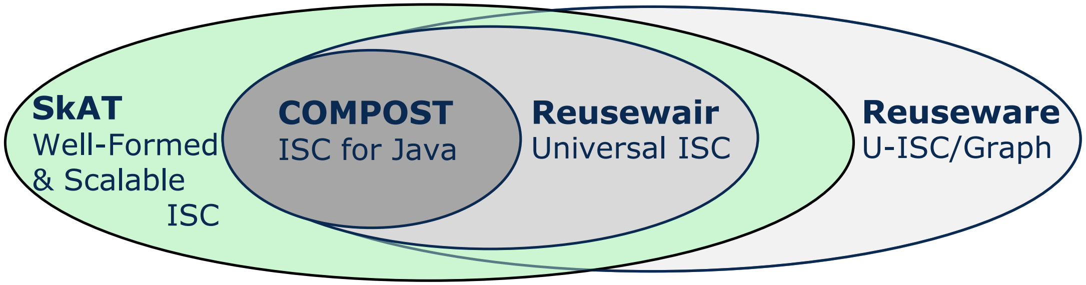
Minimal ISC

Invasive Software Composition
[Aßmann03]

Well-Formed ISC

Thesis achievements – summary

 # supported composition features



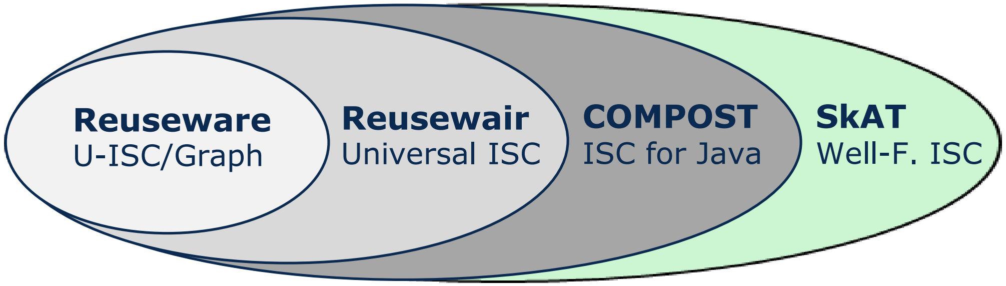
SkAT
Well-Formed & Scalable ISC

COMPOST
ISC for Java

Reusewair
Universal ISC

Reuseware
U-ISC/Graph

Support of ISC code generators (e.g., the ***BAF*** *scenario)*



Reuseware
U-ISC/Graph

Reusewair
Universal ISC

COMPOST
ISC for Java

SkAT
Well-F. ISC

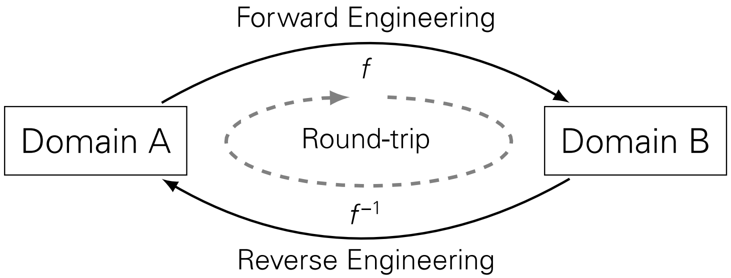
Part III: Conclusion & Outlook

Future work

- Support of model-based languages (e.g., using JastEMF [Bürger+11])
- High-level component-model specifications using DSLs
- Heterogeneous skeleton libraries (multiple languages and platforms)
- Performance optimizations (e.g., attribute caching, RACR [Bürger12])
- Support of custom contracts (e.g., based on annotations)
- Automatic interaction detection (cf. [Karol+11])

...

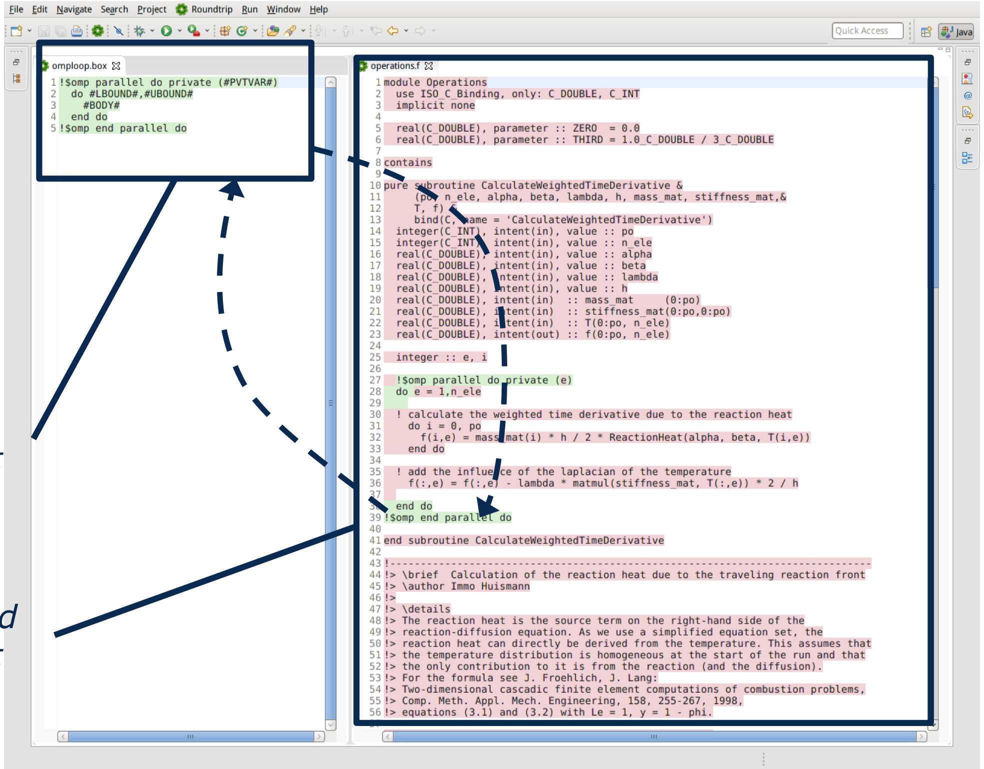
Recent work – round-trip for SkAT [Nett15]



- Propagate edits back to origin
- Maintain origin information
- Semi-automatic decision

origin fragment

composed fragment



omploop.box

```
!$omp parallel do private (#PVTVAR#)
  do #LBOUND#,#UBOUND#
    #BODY#
  end do
!$omp end parallel do
```

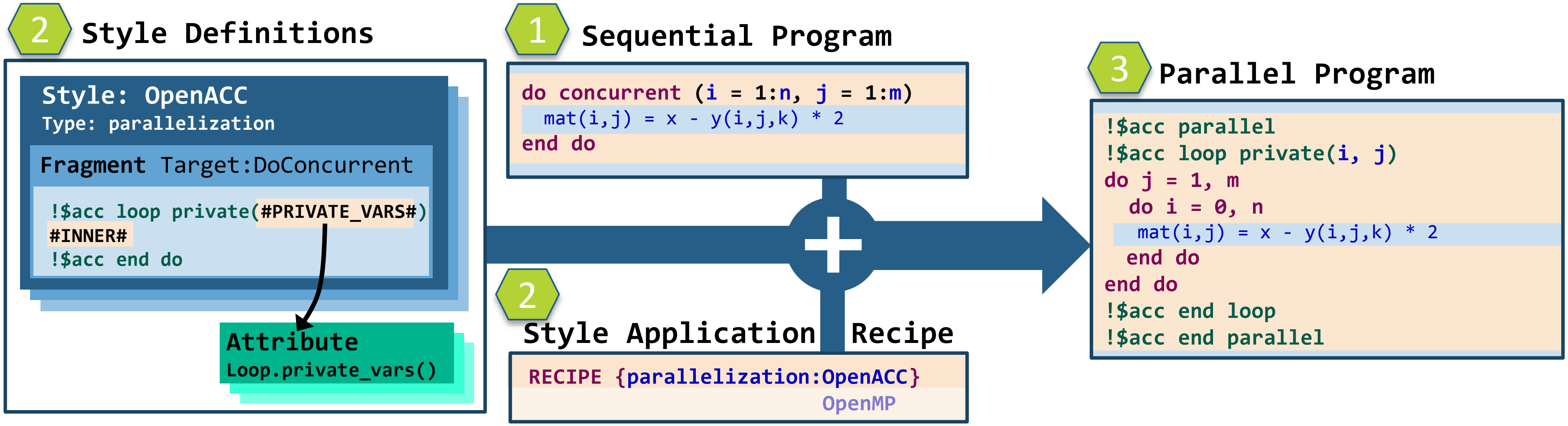
operations.f

```
module Operations
  use ISO_C_Binding, only: C_DOUBLE, C_INT
  implicit none

  real(C_DOUBLE), parameter :: ZERO  = 0.0
  real(C_DOUBLE), parameter :: THIRD = 1.0_C_DOUBLE / 3_C_DOUBLE

contains

pure subroutine CalculateWeightedTimeDerivative &
     (po, n_ele, alpha, beta, lambda, h, mass_mat, stiffness_mat,&
     T, f) &
     bind(C, name = 'CalculateWeightedTimeDerivative')
  integer(C_INT), intent(in), value :: po
  integer(C_INT), intent(in), value :: n_ele
  real(C_DOUBLE), intent(in), value :: alpha
  real(C_DOUBLE), intent(in), value :: beta
  real(C_DOUBLE), intent(in), value :: lambda
  real(C_DOUBLE), intent(in), value :: h
  real(C_DOUBLE), intent(in)  :: mass_mat     (0:po)
  real(C_DOUBLE), intent(in)  :: stiffness_mat(0:po,0:po)
  real(C_DOUBLE), intent(in)  :: T(0:po, n_ele)
  real(C_DOUBLE), intent(out) :: f(0:po, n_ele)

  integer :: e, i

  !$omp parallel do private (e)
  do e = 1,n_ele

  ! calculate the weighted time derivative due to the reaction heat
    do i = 0, po
      f(i,e) = mass_mat(i) * h / 2 * ReactionHeat(alpha, beta, T(i,e))
    end do

  ! add the influence of the laplacian of the temperature
    f(:,e) = f(:,e) - lambda * matmul(stiffness_mat, T(:,e)) * 2 / h

  end do
!$omp end parallel do

end subroutine CalculateWeightedTimeDerivative

!------------------------------------------------------------------------------
!> \brief  Calculation of the reaction heat due to the traveling reaction front
!> \author Immo Huismann
!>
!> \details
!> The reaction heat is the source term on the right-hand side of the
!> reaction-diffusion equation. As we use a simplified equation set, the
!> reaction heat can directly be derived from the temperature. This assumes that
!> the temperature distribution is homogeneous at the start of the run and that
!> the only contribution to it is from the reaction (and the diffusion).
!> For the formula see J. Froehlich, J. Lang:
!> Two-dimensional cascadic finite element computations of combustion problems,
!> Comp. Meth. Appl. Mech. Engineering, 158, 255-267, 1998,
!> equations (3.1) and (3.2) with Le = 1, y = 1 - phi.
```

(figure and screenshot from [Nett15])

Recent work – Orchestration Style Sheets [Mey+Aßmann15]

- **Idea:** annotate program with *parallelization styles*
- Compose different target-platform-specific variants from that program
- More flexible than preprocessor directives or macros
- Problem-specific and parallelization code get untangled



2 **Style Definitions**

Style: OpenACC
Type: parallelization

Fragment Target:DoConcurrent

```
!$acc loop private(#PRIVATE_VARS#)
#INNER#
!$acc end do
```

Attribute
Loop.private_vars()

1 **Sequential Program**

```
do concurrent (i = 1:n, j = 1:m)
  mat(i,j) = x - y(i,j,k) * 2
end do
```

2 **Style Application Recipe**

```
RECIPE {parallelization:OpenACC}
                        OpenMP
```

3 **Parallel Program**

```
!$acc parallel
!$acc loop private(i, j)
do j = 1, m
  do i = 0, n
    mat(i,j) = x - y(i,j,k) * 2
  end do
end do
!$acc end loop
!$acc end parallel
```

(figures by Johannes Mey)

EVOLUTION OF MEN DOING A PHD

End

TECHNISCHE
UNIVERSITÄT
DRESDEN

References

[Hedin00] Hedin, Görel. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, Nr. 3: 301–317.

[Boyland05] Boyland, John T. 2005. Remote attribute grammars. *Journal of the ACM* 52, Nr. 4 (July): 627–687.

[Johannes11] Johannes, Jendrik. 2011. Component-Based Model-Driven Software Development. Dissertation/Ph.D. thesis, Technische Universität Dresden, Januar.

[Henriksson09] Henriksson, Jakob. 2009. A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. Dissertation/Ph.D. thesis, Technische Universität Dresden, Januar.

[Aßmann03] Aßmann, Uwe. 2003. *Invasive Software Composition*. 1. Aufl. Springer.

[Bürger+11] Bürger, Christoff, Sven Karol, Christian Wende, und Uwe Aßmann. 2011. Reference Attribute Grammars for Metamodel Semantics. In *Software Language Engineering*. Springer Berlin / Heidelberg.

[Knuth68] Knuth, D. E. 1968. Semantics of context-free languages. *Theory of Computing Systems* 2, Nr. 2: 127–145.

[Vogt+89] Vogt, Harald H, Doaitse Swierstra, und Matthijs F Kuiper. 1989. Higher Order Attribute Grammars. In *PLDI '89*, 131–145. ACM.

[Ekman06] Ekman, Torbjörn. 2006. Extensible Compiler Construction. Dissertation/Ph.D. thesis, University of Lund.

[Grosch90] Grosch, Jan. 1990. *Object-Oriented Attribute Grammars*. Technical Report. Aachen: CoCoLab Datenverarbeitung, August 27.

[Hedin89] Hedin, Görel. An Object-Oriented Notation for Attribute Grammars. In *Proceedings ECOOP'89*, 329-345. BCS Workshop Series. Nottingham, U.K.: Cambridge University Press.

[Meyer92] Meyer, B. 1992. Applying `design by contract'. *Computer* 25, Nr. 10 (Oktober): 40-51.

[Klaeren+01] Klaeren, Herbert, Elke Pulvermüller, Awais Rashid, und Andreas Speck. 2001. Aspect Composition Applying the Design by Contract Principle. In *Generative and Component-Based Software Engineering,* 2177:57-69. LNCS. Springer Berlin / Heidelberg.

[Arnoldus11] Arnoldus, B. J. 2011. An Illumination of the Template Enigma: Software Code Generation with Templates. TU Eindhoven.

[Heidenreich+09] Heidenreich, Florian, Jendrik Johannes, Mirko Seifert, Christian Wende, und Marcel Böhme. 2009. Generating Safe Template Languages. In *Proc. of ACM 8th International Conference on Generative Programming and Component Engineering (GPCE'09)*. ACM Press.

[Bürger+10] Bürger, Christoff, Sven Karol, und Christian Wende. 2010. Applying attribute grammars for metamodel semantics. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, 1:1–1:5. FML '10. New York, NY, USA: ACM.

[Ekman+07] The jastadd extensible java compiler. In *SIGPLAN Not.*, 42(10), S.1—18.

[Kristensen+87] Kristensen, Bent Bruun, Ole Lehrmann Madsen, Birger Moller-Pedersen, und Kristen Nygaard. 1987. „The BETA programming language". In *Research directions in object-oriented programming*, 7–48. Cambridge, MA, USA: MIT Press.

[Cole 04] Cole, Murray. „Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". *Parallel Computing* 30, Nr. 3 (2004): 389–406. doi:10.1016/j.parco.2003.12.002.

[Goswami+02] Goswami, Dhrubajyoti, Ajit Singh, und Bruno R. Preiss. „From Design Patterns to Parallel Architectural Skeletons". *Journal of Parallel and Distributed Computing* 62, Nr. 4 (2002): 669 – 695. doi:10.1006/jpdc.2001.1809.

[Nett15] Nett, Tobias. 2015. Round-Trip Engineering of Template-Based Code Generation in SkAT. Großer Beleg, TU Dresden.

[Visser97] Visser, Eelco (1997). "Syntax Definition for Language Prototyping." PhD thesis. Amsterdam, Netherlands: University of Amsterdam

[Bürger12] Bürger, Christoff (2012). *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. TR TUD-FI12-09.

[Mey15] Mey, Johannes. 2015. *Orchestration Style Sheets for Heterogeneous Many-Core Architectures*. Poster/demo at the CfAED research festival March 2015, TU Dresden.

[Moonen01] Moonen, Leon (2001). "Generating Robust Parsers Using Island Grammars." In: *Proceedings of Reverse Engineering 2001*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 13–22.

[Tasic14] Marta Tasic (2014). *"Benchmarking Incremental Dynamic Attribute Evaluation - RACR/JastAdd Performance Case Study in the Domain of Invasive Software Composition."* Published: online available slide set. Dresden, Germany: TU Dresden.

[Laitinen+00] Laitinen, Mauri, Mohamed Fayad, und Robert P. Ward. „The problem with scalability." *Commun. ACM* 43, Nr. 9 (2000): 105–7.

References

[Brabrand+Schwartzbach02] Brabrand, Claus and Michael I. Schwartzbach (2002). “Growing Languages with Metamorphic Syntax Macros.” In: *Proceedings of PEPM ’02*. New York, NY, USA: ACM, pp. 31–40.

[Kästner+Apel09] Kästner, Christian and Sven Apel (2009). “Virtual Separation of Concerns - A Second Chance for Preprocessors.” In: *Journal of Object Technology* 8.6, pp. 59–78.

[Kästner+11] Kästner, Christian, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger (2011b). “Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation.” In: *OOPSLA ’11*. New York, NY, USA: ACM, pp. 805–824.

[Heumüller+14] Heumüller, Robert, Jochen Quante, and Andreas Thums (2014). “Parsing Variant C Code: An Evaluation on Automotive Software.” In: *Softwaretechnik-Trends* 34.2: *Berichte und Beiträge vom 16. Workshop “Software-Reengineering und -Evolution”*

[Kiczales+01] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). “An Overview of AspectJ.” In: *Proceedings ECOOP ’01*. Vol. 2072. LNCS. Berlin / Heidelberg: Springer, pp. 327–353.

[Avgustinov+06] Avgustinov, Pavel, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble (2006). “abc: An Extensible AspectJ Compiler.” In: *TAOSD I*. Vol. 3880. LNCS. Berlin / Heidelberg, Germany: Springer, pp. 293–334.

[Avgustinov+08] Avgustinov, Pavel, Torbjörn Ekman, and Julian Tibble (2008). “Modularity First: A Case for Mixing AOP and Attribute Grammars.” In: *Proceedings of AOSD ’08*. New York, NY, USA: ACM, pp. 25–35.

[Lohmann+04] Lohmann, Daniel, Georg Blaschke, and Olaf Spinczyk (2004). “Generic Advice: On the Com- bination of AOP with Generative Programming in AspectC++.” In: *Proceedings of GPCE 2004*. Vol. 3286. LNCS. Berlin / Heidelberg, Germany: Springer, pp. 55–74.

[Erwig+Walkingshaw11] Erwig, Martin and Eric Walkingshaw (2011). “The Choice Calculus: A Representation for Software Variation.” In: *TOSEM* 21.1, 6:1–6:27.

[Batory+04] Batory, Don, Jacob N. Sarvela, and Axel Rauschmayer (2004). “Scaling Step-Wise Refinement.” In: *IEEE Transactions on Software Engineering* 30.6, pp. 355–371.

[Chen+14] Chen, Sheng, Martin Erwig, and Eric Walkingshaw (2014). “Extending Type Inference to Varia- tional Programs.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.1, 1:1–1:54.

[Azurat07] Azurat, Ade (2007). “Mechanization of Invasive software Composition in F-Logic.” In: *Proc. of the CEA ’07*. Stevens Point, Wisconsin, USA: World Scientific, Engineering Academy, and Society (WSEAS), pp. 89–94.

[Kezadri+14] Kezadri Hamiaz, Mounira, Marc Pantel, Benoît Combemale, and Xavier Thirioux (2014). “Correct-by-Construction Model Composition: Application to the Invasive Software Composition Method.” In: *EPTC* 147: *Proc. of FESCA 2014*, pp. 108–122.

[Kezadri+12] Kezadri, Mounira, Benoît Combemale, Marc Pantel, and Xavier Thirioux (2012). “A Proof Assistant Based Formalization of MDE Components.” In: *Proceedings of FACS 2011*. Vol. 7253. LNCS. Berlin / Heidelberg, Germany: Springer, pp. 223–240.

[Kifer+95] Kifer, Michael, Georg Lausen, and James Wu (1995). “Logical Foundations of Object-oriented and Frame-based Languages.” In: *Journal of the ACM* 42.4, pp. 741–843.

[Bravenboer+08] Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser (2008). “Stratego/XT 0.17. A Language and Toolset for Program Transformation.” In: *SCP* 72.1-2: *Second Issue of EST,* pp. 52–70.

[Cordy06] Cordy, James R. (2006). “The TXL Source Transformation Language.” In: *SCP* 61.3: *Special Issue on (LDTA ’04)*, pp. 190–210.

[Baxter+04] Baxter, Ira D., Christopher Pidgeon, and Michael Mehlich (2004). “DMSR : Program Transformations for Practical Scalable Software Evolution.” In: *Proceedings of ICSE ’04*. Washington, DC, USA: IEEE, 625–634.

[Ford02] Ford, Bryan (2002). “Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl.” In: *Proc.of ICFP ’02*. New York, USA: ACM, pp. 36–47.

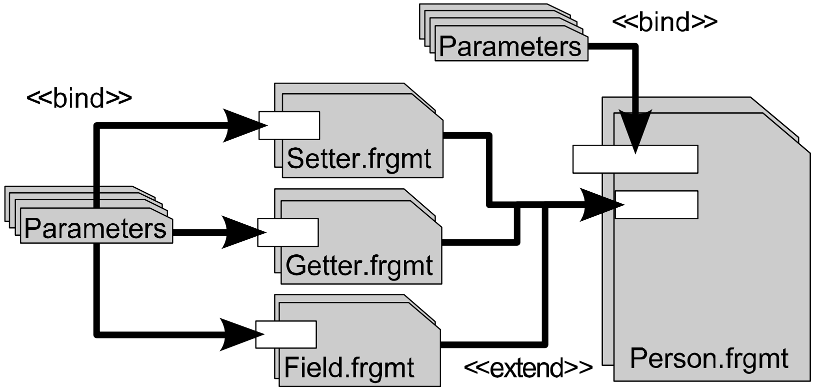
[Ford04] Ford, Bryan (2004). “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.” In: *Proc. of POPL ’04*. New York,USA: ACM, pp. 111–122.

[Hogeman14] Hogeman, Erik (2014). “Extending JastAddJ to Java 8.” Master Thesis. Lund, Sweden: University of Lund

[Karol+11] Karol, Sven, Matthias Niederhausen, Daniel Kadner, Uwe Aßmann, and Klaus Meißner (2011). “Detecting and Resolving Conflicts Between Adaptation Aspects in Multi-staged XML Transformations.” In: *DocEng’11)*. New York, NY, USA: ACM, pp. 229–238.

C4: Consolidating review of the state of the art in ISC

BAF scenario: generating code from a business domain model



```
//getter.frgmt
public [[Type]] [[GetSfx]]() {
  return [[Field]];
}
```

```
//person.frgmt
public class [[Type]] extends Person {
  public String asString(){
    String v = [[TypeName]];
    v+= "[[Pfx]] id:" + getID();
    v+= "[[Pfx]] name:" + getName();
    return v;
  }
}
```

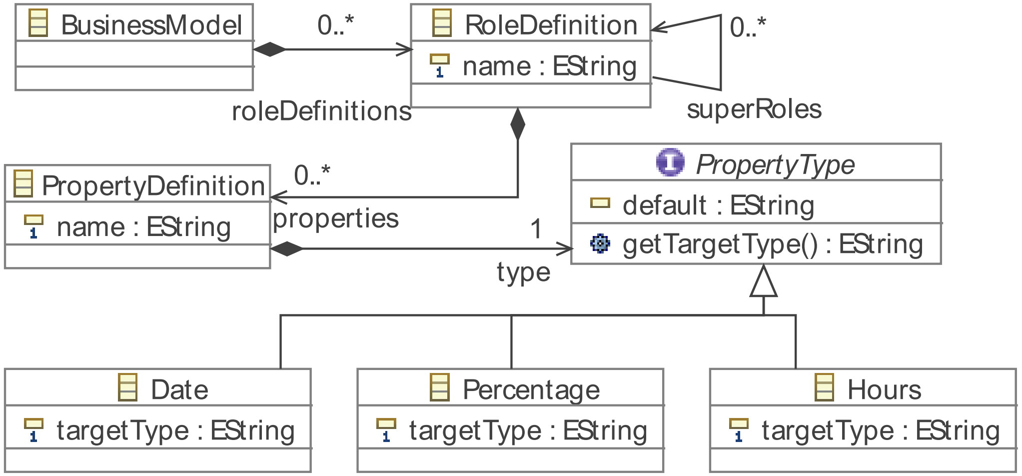
```
//field.frgmt
private [[Type]] [[Field]];
```

```
//setter.frgmt
public void [[SetSfx]]([[Type]] [[Field]]){
  this.[[Field]] = [[Field]];
}
```

C4: Consolidating review of the state of the art in ISC

BAF scenario: metamodel and grammar

BAF metamodel:



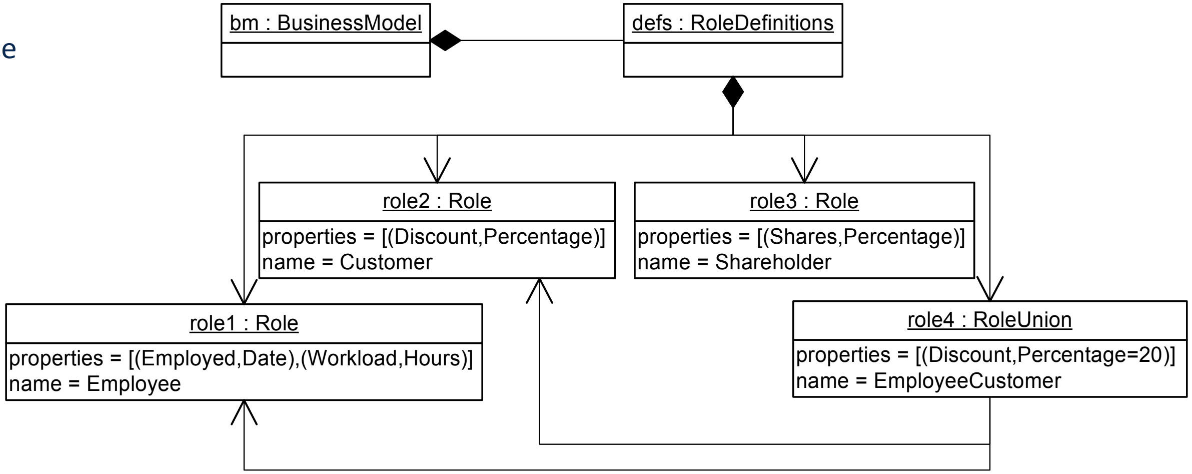
BAF grammar:

```
SYNTAXDEF bm
FOR <http://www.emftext.org/language/businessmodel> START BusinessModel
RULES {
  BusinessModel ::= "application" "roles" "{" roleDefinitions* "}";
  RoleDefinition ::= "object" name[] ("is_a" superRoles[] ("," superRoles[])* )? (":"
            properties+)?; PropertyDefinition ::= name[] ":" type;
  Date ::= "Date" ("[" default[] "]")?; Hours ::= "Hours" ("[" default[] "]")?;
  Percentage ::= "Percentage" ("[" default[] "]")?; }
```

C4: Consolidating review of the state of the art in ISC

BAF scenario: example model instance

Example BAF instance object graph:

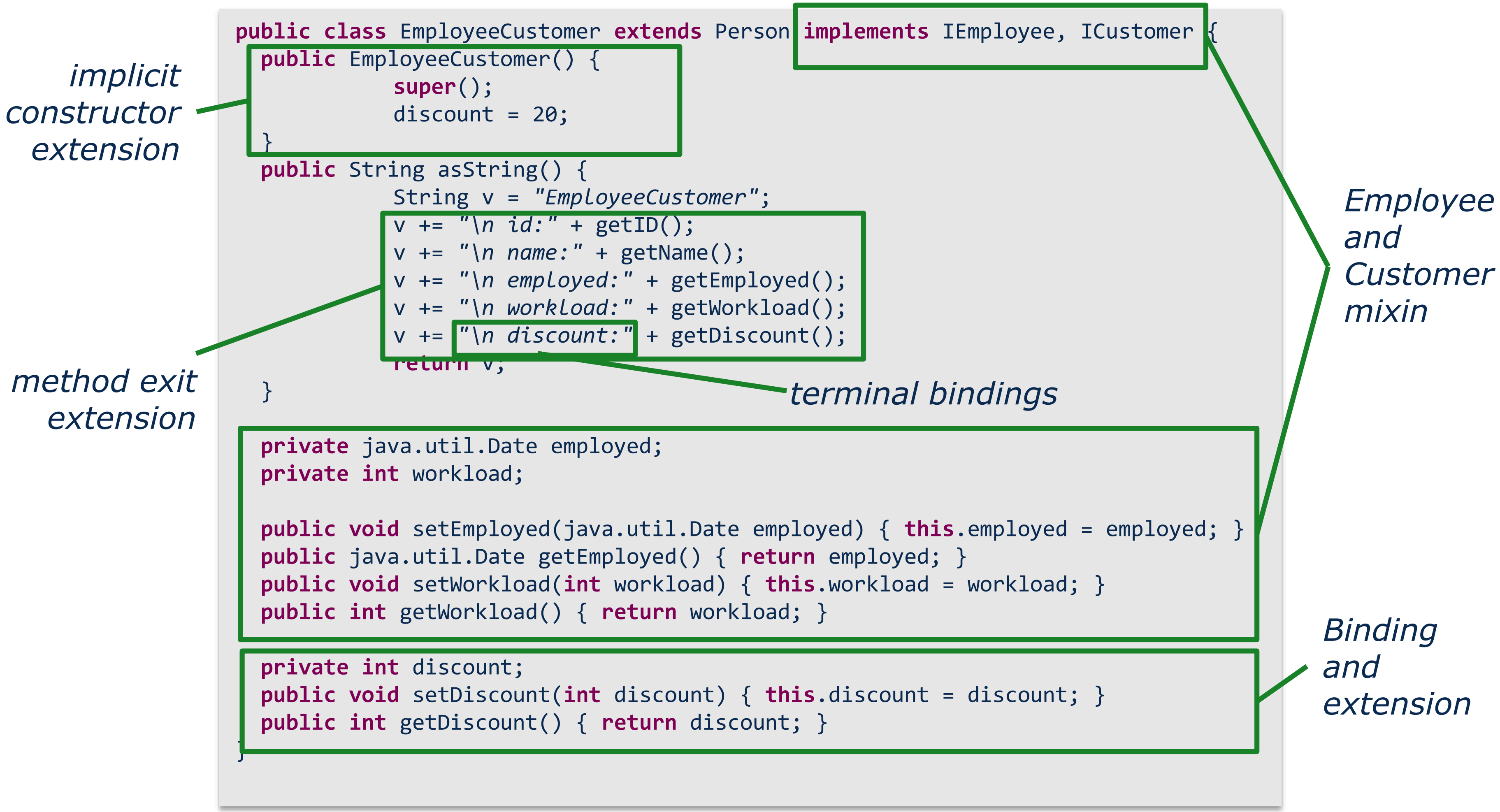


Example BAF instance specification:

```
roles {
   object Employee:
     employed : Date
     workload : Hours
   object Customer:
     discount : Percentage
   object Shareholder:
     shares : Percentage
   object EmployeeCustomer is_a Employee,Customer:
     discount : Percentage[20]
}
```

C4: Consolidating review of the state of the art in ISC

BAF scenario: ideal output



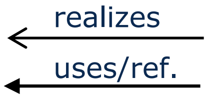
```
public class EmployeeCustomer extends Person implements IEmployee, ICustomer {
  public EmployeeCustomer() {
            super();
            discount = 20;
  }
  public String asString() {
            String v = "EmployeeCustomer";
            v += "\n id:" + getID();
            v += "\n name:" + getName();
            v += "\n employed:" + getEmployed();
            v += "\n workload:" + getWorkload();
            v += "\n discount:" + getDiscount();
            return v;
  }

  private java.util.Date employed;
  private int workload;

  public void setEmployed(java.util.Date employed) { this.employed = employed; }
  public java.util.Date getEmployed() { return employed; }
  public void setWorkload(int workload) { this.workload = workload; }
  public int getWorkload() { return workload; }

  private int discount;
  public void setDiscount(int discount) { this.discount = discount; }
  public int getDiscount() { return discount; }
}
```

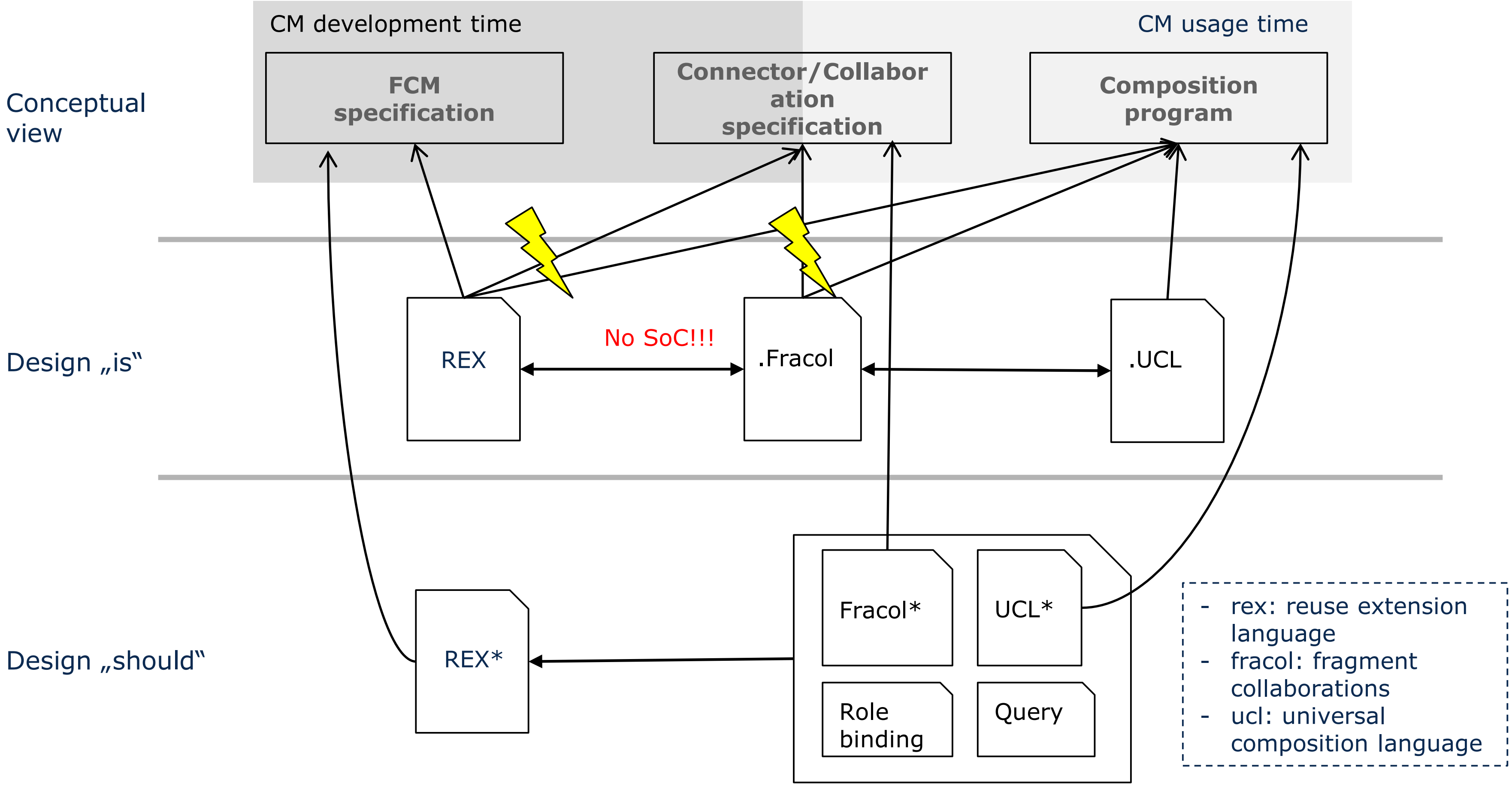
TECHNISCHE
UNIVERSITÄT
DRESDEN

Part I: Thesis Contributions

	COMPOST	**Reusew*air***	**Reuseware**	**SkAT**
Tree model	AST (Recoder)	EMOF (Ecore)	EMOF (Ecore)	AST (JastAdd)
Graph model	-	-	overlay graph, EMOF(Ecore)	overlay graph, RAG (JastAdd)
Tree FCM	slot, hook	slot	hook, prototype	slot, hook, rudiment
Tree operators	bind, extend, extract	bind, extend	bind, extend	bind, extend, extract
Graph FCM	-	-	slot, anchor	attribute declarations
Graph operators	-	-	slot ← anchor	attribute equations
Composition language	Turing complete (Java)	Turing complete (Java)	acyclic data-flow graph (DSL)	Turing complete (Java)
Composition strategies	-	-	-	fixpoint, point wise, ordered, attributes
FCM specification	class hierarchy	DSL	set of DSLs	attribute grammar
FCM extensibility	via inheritance	-	-*	RAG modules
Fragment roles	-	-	name-based	-
FCM manifestation	implementation	generative	interpretative	generative
Graphical languages	-	-	graphical EMOF languages	via JastEMF*
Well-formedness	-	-	-	contracts, assertions
Partial languages	-	-	-	island FCMs

Reuseware Issues





- rex: reuse extension language
- fracol: fragment collaborations
- ucl: universal composition language

Attribute grammars: shortened definition

Definition (attribute grammar):

An attribute grammar (AG) is an 8-tuple G=($\mathbf{G_0}$,**Syn**,**Inh**,$\mathbf{Syn_x}$,$\mathbf{Inh_x}$,**K**,**Ω**,**Φ**) with the following components:

- $\mathbf{G_0}$ = (N,Σ,P,S) a CFG,
- **Syn** and **Inh** the finite, disjoint sets of synthesized and inherited attributes,
- $\mathbf{Syn_x}$: N → P(**Syn**) a function that assigns a set of synthesized attributes to each nonterminal in G_0,
- $\mathbf{Inh_x}$: N → P(**Inh**) a function that assigns a set of inherited attributes to each nonterminal in G_0,
- **K** a set of types/sorts,
- **Ω** : **Inh** ∪ **Syn** → K a function assigning each attribute a $\kappa \in K$,
- **Φ** a set of semantic functions $\varphi_{(p,i,a)}$ with $p \in P$, $i \in \{0,\dots,n_p\}$, $a \in Syn_x(p_i) \cup Inh_x(p_i)$.

RAG-based component models: compositional points

- Hooks are identified using *inherited* attributes → context-dependent
- Hooks are list nodes

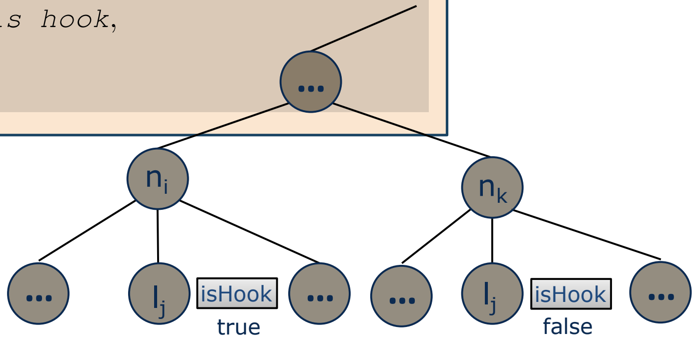
$$\textbf{inh}\ \texttt{string}_\perp\ \{n \mid n \in N\}.\texttt{hookName} \qquad \textbf{inh}\ \texttt{bool}_\perp\ \{n \mid n \in N\}.\texttt{isHook}$$

$$\textbf{fun}\downarrow\texttt{CompositionEnvironment}.\texttt{child}_{\texttt{all}}.\texttt{isHook} = \mathit{false}$$

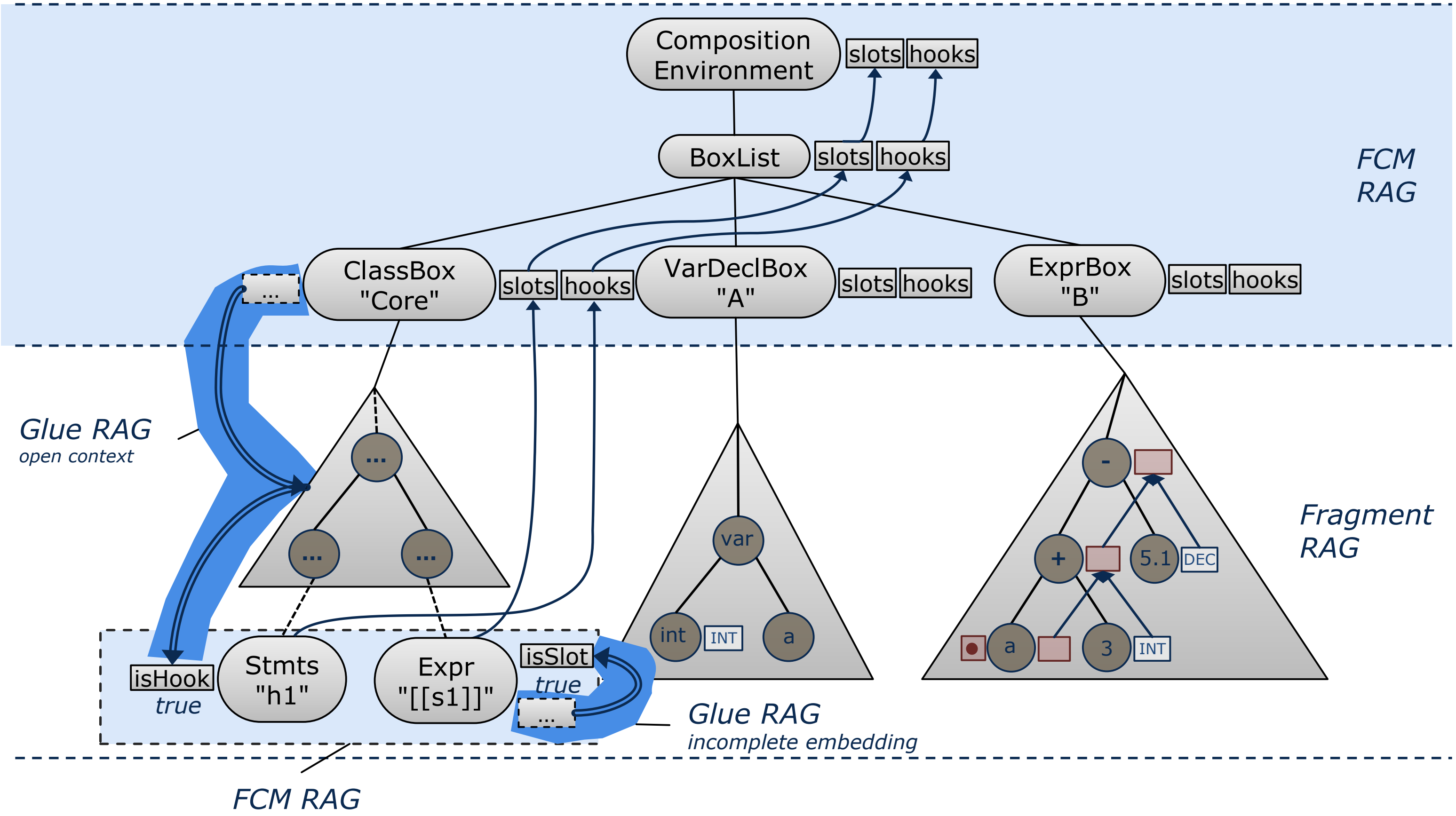
$$\textbf{fun}\downarrow\texttt{CompositionEnvironment}.\texttt{child}_{\texttt{all}}.\texttt{hookName} = \perp$$

$$\textbf{fun}\ n_i.l_j.\texttt{isHook} = \begin{cases} \mathit{true} & \textbf{if}\ \mathit{l\text{-}node\ matches\ pattern}, \\ \mathit{false} & \textbf{else}. \end{cases}$$

$$\textbf{fun}\ n_i.l_j.\texttt{hookName} = \begin{cases} \mathit{name\ from\ context} & \textbf{if}\ \mathit{node\ is\ hook}, \\ \perp & \textbf{else}. \end{cases}$$



RAG-based component models: glue RAG



RAG-based component models: generalized composers

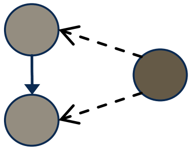
- composers are identified via composer-identification attributes
- any node in the AST can be a composer
- support embedded invasive software composition and macro languages

$$\begin{array}{l}
\textbf{syn}\ \texttt{Node}\star\ \{n\,|\,n\in N\}.\texttt{points}\\
\textbf{syn}\ \texttt{Node}\ \{n\,|\,n\in N\}.\texttt{srcFragment}\\
\\
\textbf{syn}\ \texttt{bool}_{\perp}\ \{n\,|\,n\in N\}.\texttt{isBind}\\
\textbf{syn}\ \texttt{bool}_{\perp}\ \{n\,|\,n\in N\}.\texttt{isExtend}\\
\textbf{syn}\ \texttt{bool}_{\perp}\ \{n\,|\,n\in N\}.\texttt{isExtract}
\end{array}$$

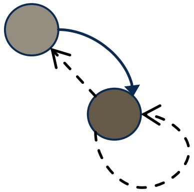
$$\begin{array}{ll}
\textbf{fun}\ \{n\,|\,n\in N\setminus\mathcal{B}\}.\texttt{isBind} & = \mathit{false}\\
\textbf{fun}\ \{n\,|\,n\in \mathcal{B}\}.\texttt{isBind} & = \mathit{true}\\
\textbf{fun}\ \{n\,|\,n\in N\setminus\mathcal{E}\}.\texttt{isExtend} & = \mathit{false}\\
\textbf{fun}\ \{n\,|\,n\in \mathcal{E}\}.\texttt{isExtend} & = \mathit{true}\\
\textbf{fun}\ \{n\,|\,n\in N\setminus\mathcal{D}\}.\texttt{isExtract} & = \mathit{false}\\
\textbf{fun}\ \{n\,|\,n\in \mathcal{D}\}.\texttt{isExtract} & = \mathit{true}
\end{array}$$

RAG-based component models: composer categories

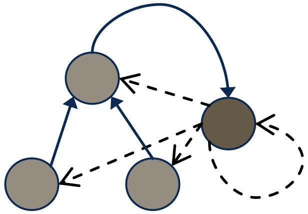
1. **primitive composers**
   - not part of the fragment language
   - do not own their source fragment



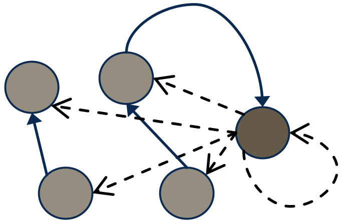
2. **primitive in-place composers (e.g., include)**
   - part of the fragment language
   - are compositional points at the same time
   - may produce their own fragments



3. **local in-place composers (e.g., macro call)**
   - like primitive composers, but perform additional local compositions

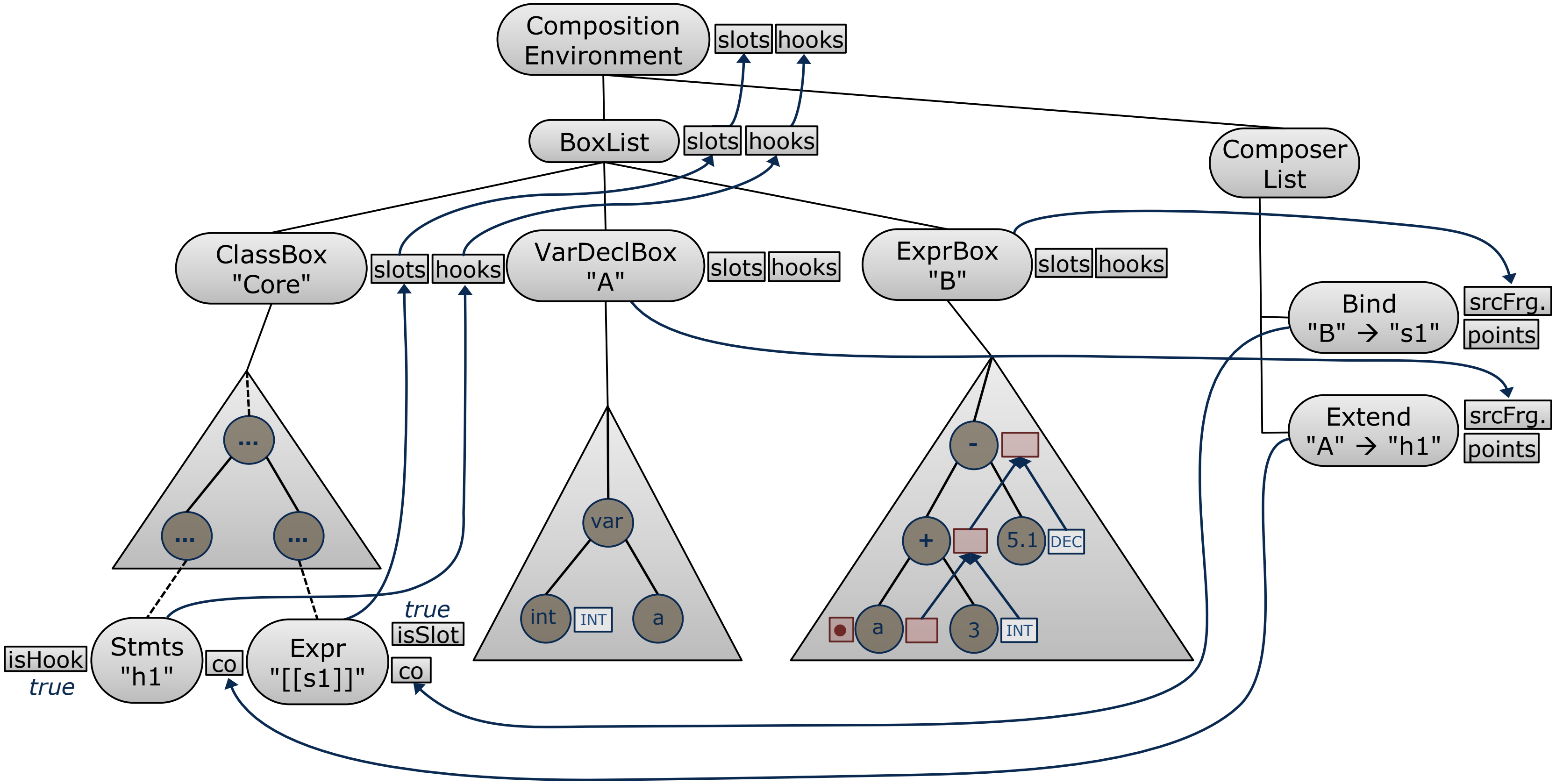


4. **non-local in-place composers (e.g., mixin composer)**
   - perform additional non-local compositions as side-effects

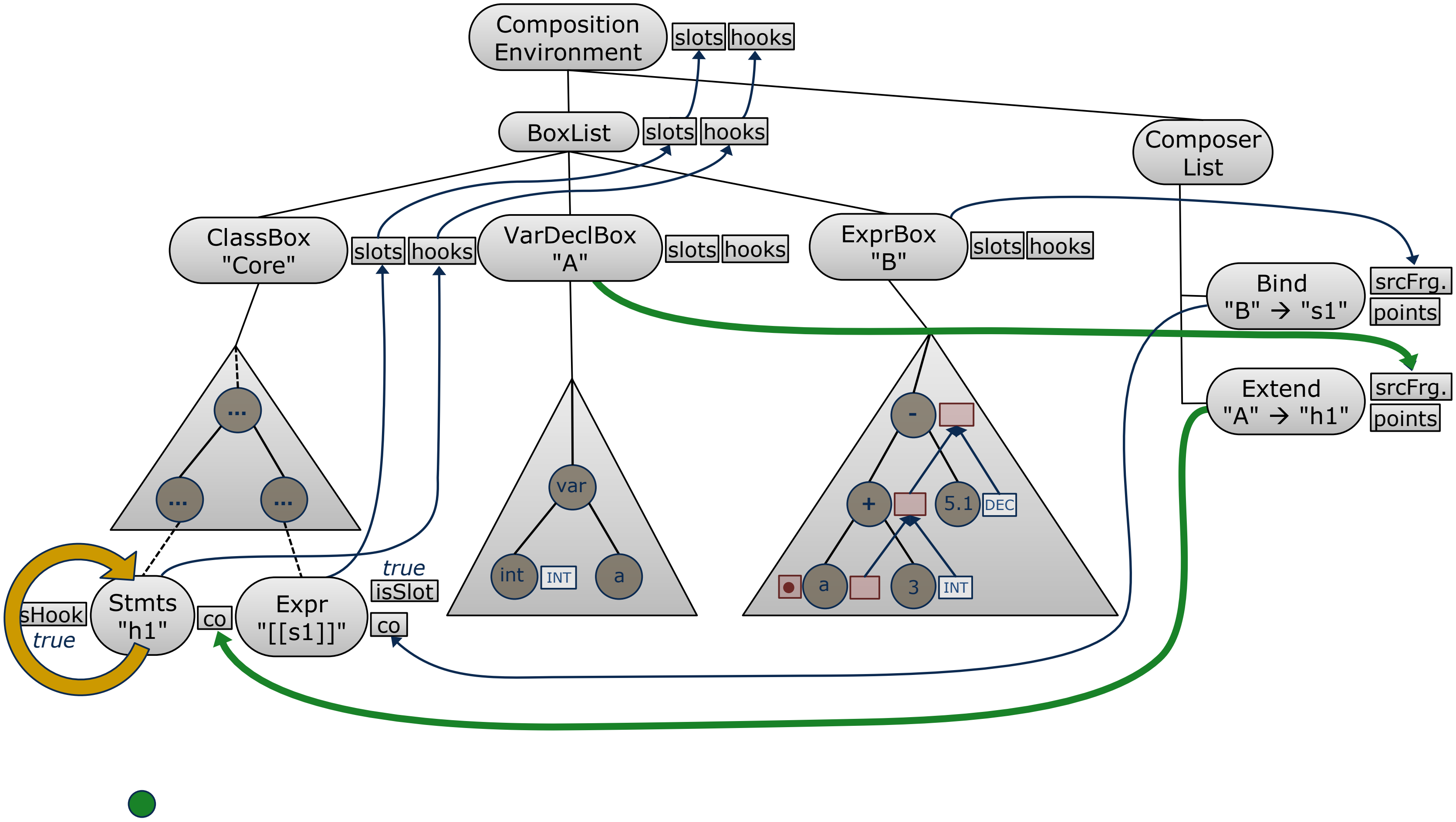


5. **external composers (e.g., aspects)**
   - are defined in an external composition language
   - declare their own embedded fragments
   - use the composition environment API

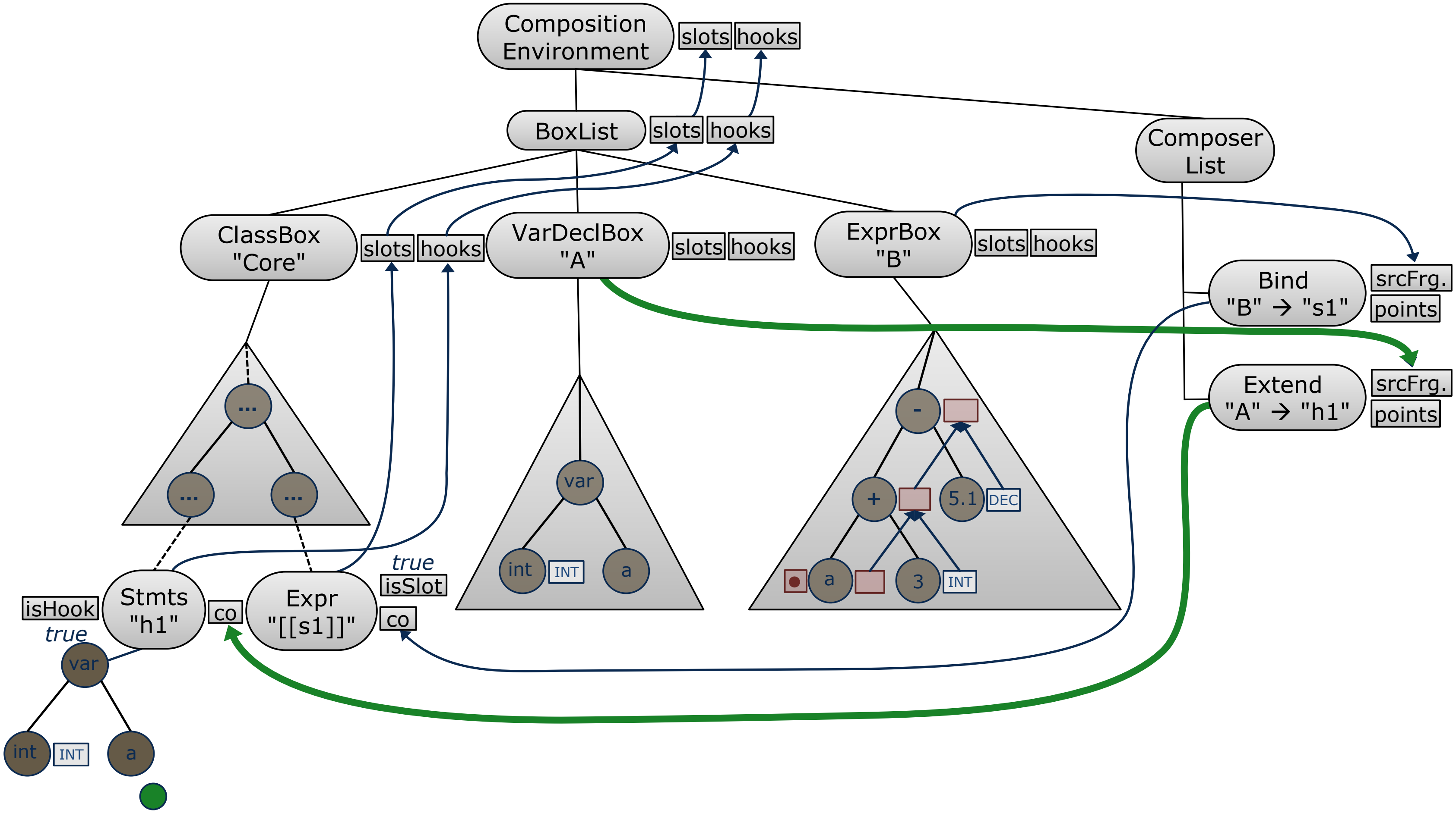
Composition strategies: point-determined composition



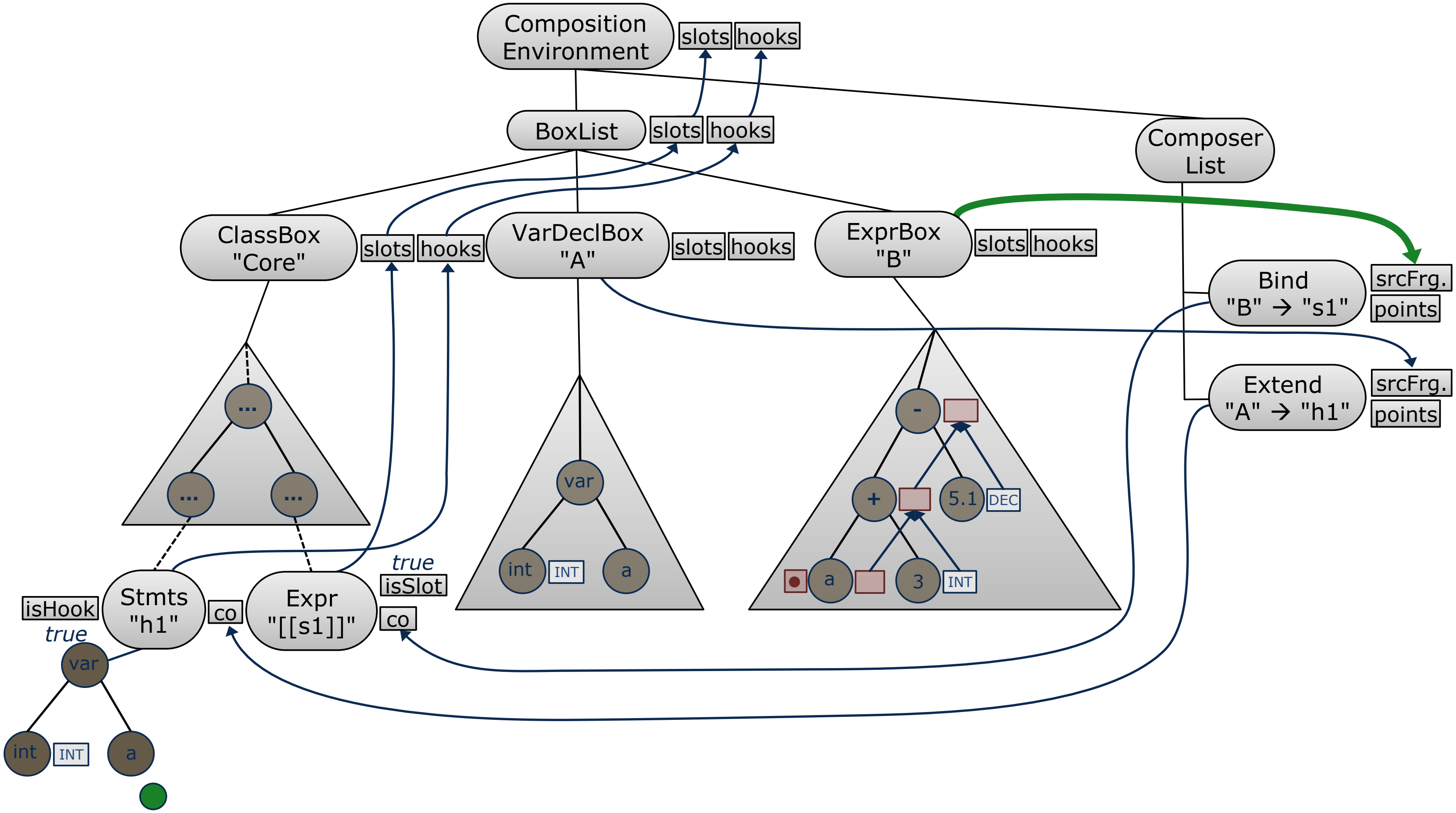
Composition strategies: point-determined composition



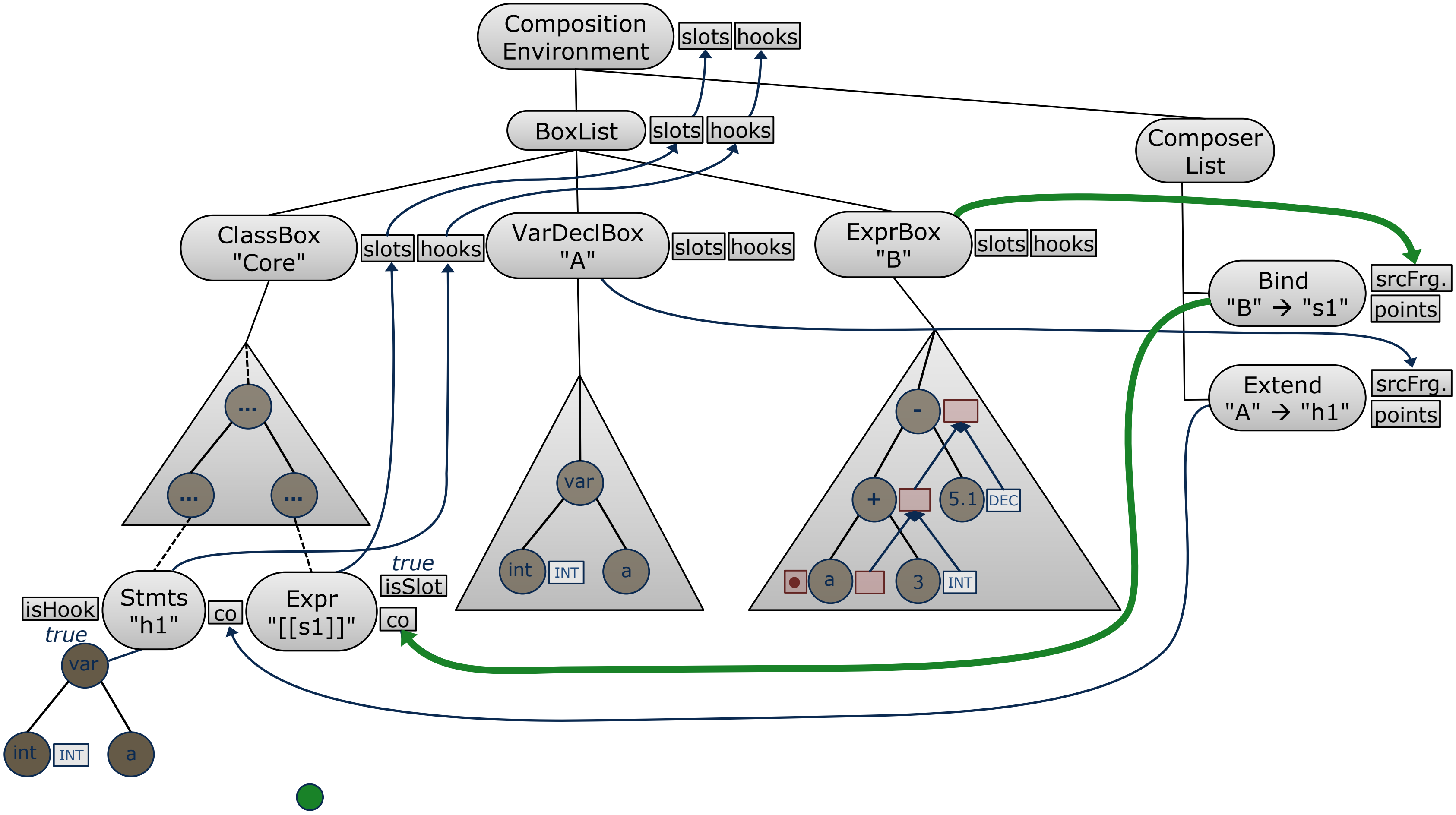
Composition strategies: point-determined composition



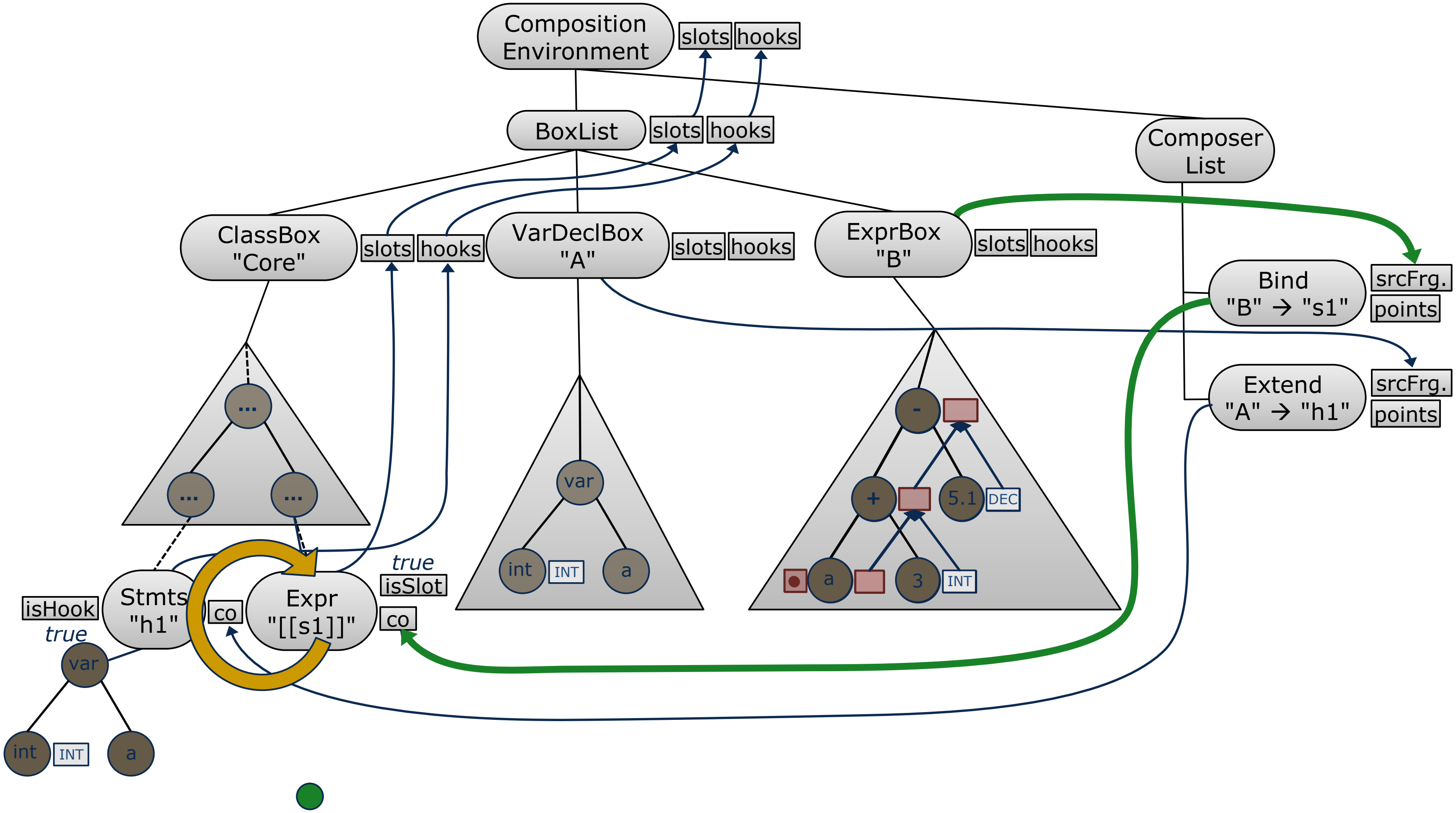
Composition strategies: point-determined composition



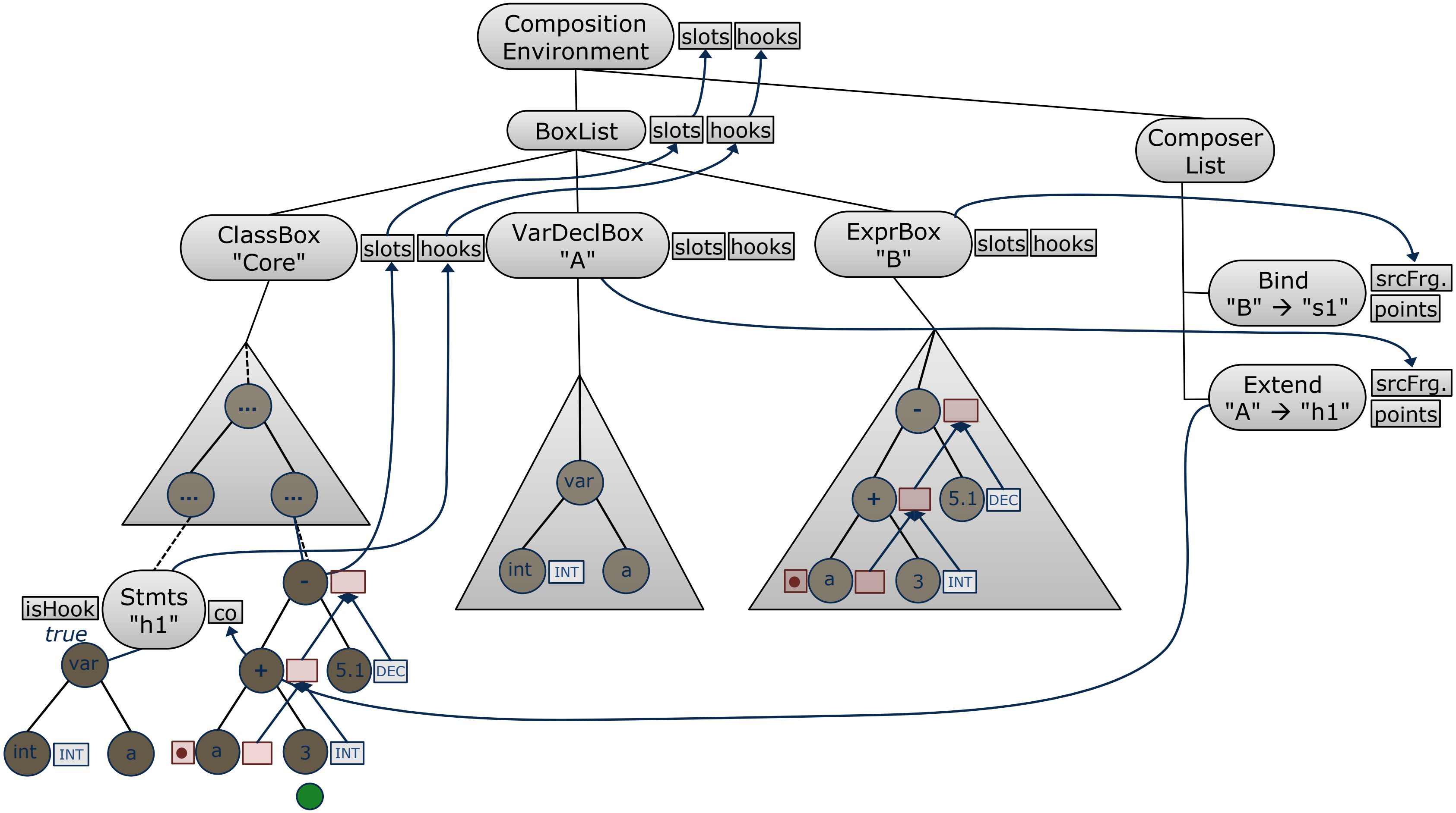
Composition strategies: point-determined composition



Composition strategies: point-determined composition



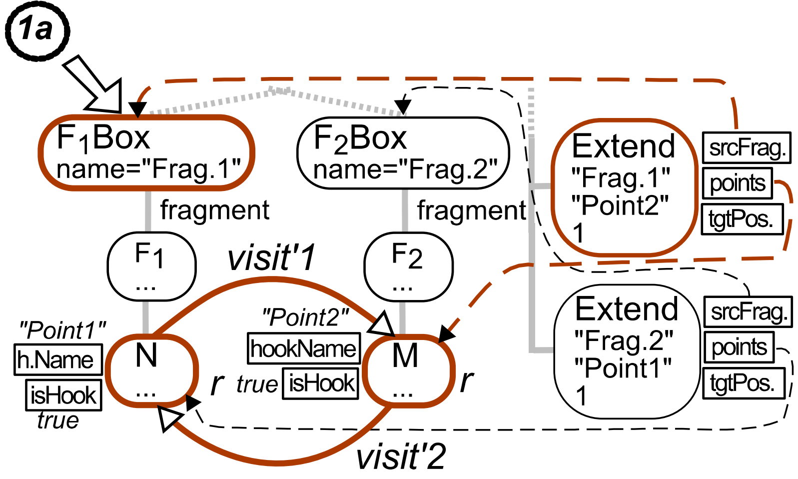
Composition strategies: point-determined composition



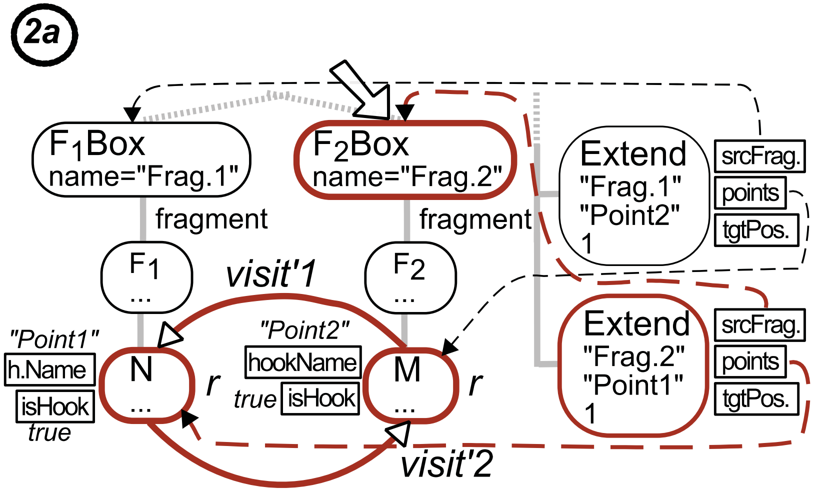
Composition strategies: attribute-determined composition

- different starting points may cause different composition results
- attribute dependencies are not obvious: composition may yield unintended results

Situation 1

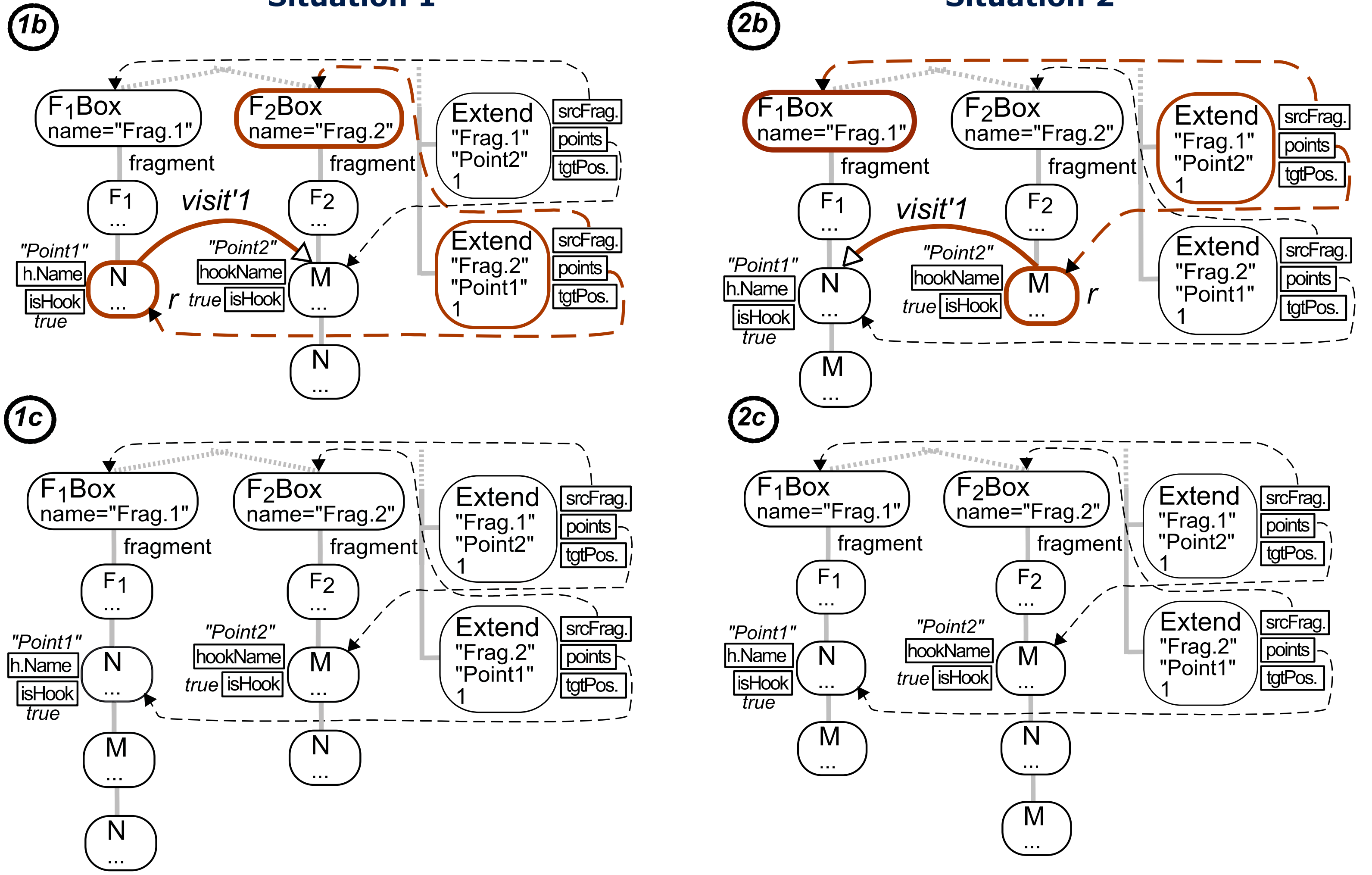


Situation 2

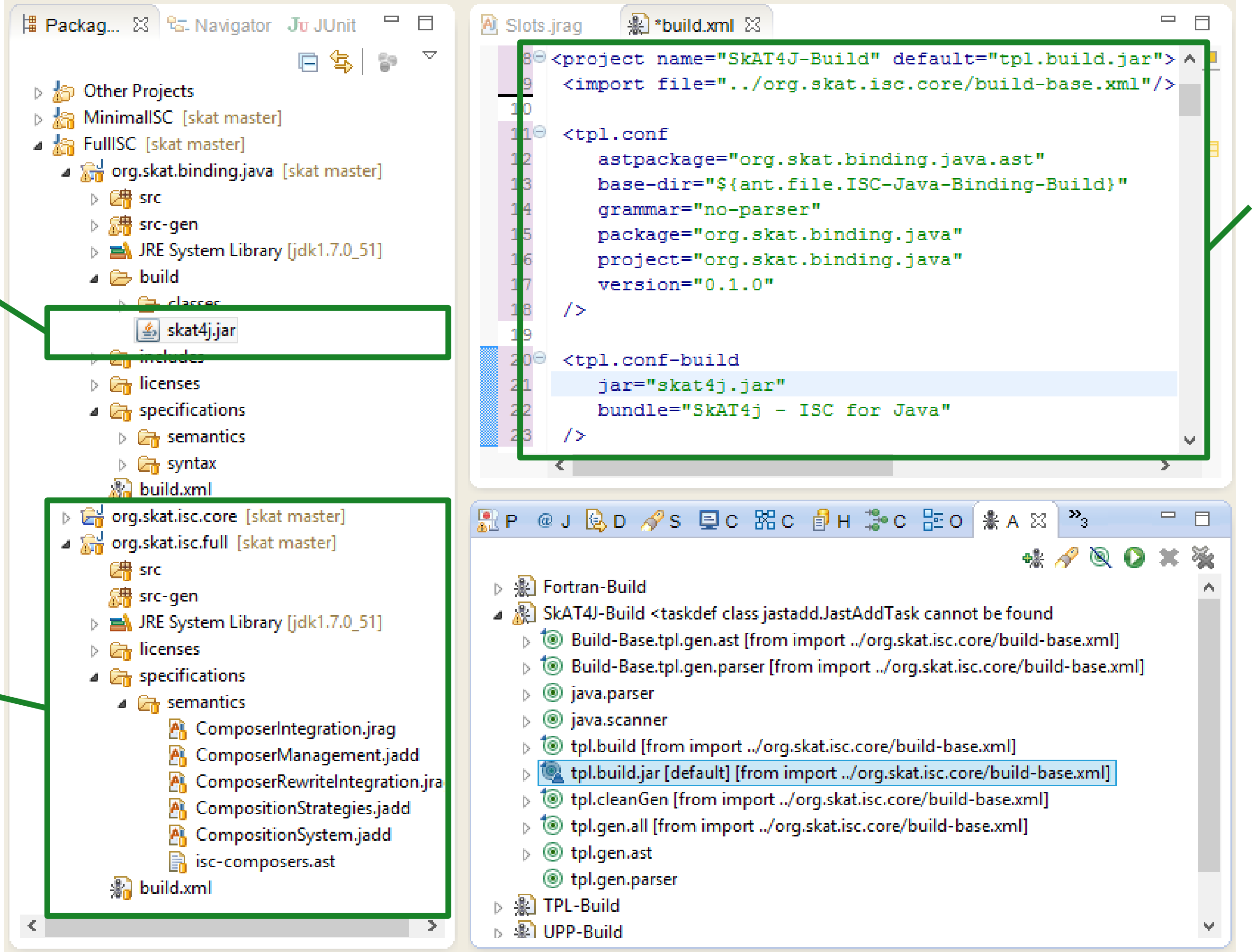


Composition strategies: attribute-determined composition

Situation 1 **Situation 2**



SkAT screenshot



```
▷ Other Projects
▷ MinimalISC [skat master]
◢ FullISC [skat master]
  ◢ org.skat.binding.java [skat master]
    ▷ src
    ▷ src-gen
    ▷ JRE System Library [jdk1.7.0_51]
    ◢ build
      ▷ classes
        skat4j.jar
    ▷ includes
    ▷ licenses
    ◢ specifications
      ▷ semantics
      ▷ syntax
      build.xml
  ▷ org.skat.isc.core [skat master]
  ◢ org.skat.isc.full [skat master]
      src
      src-gen
    ▷ JRE System Library [jdk1.7.0_51]
    ▷ licenses
    ◢ specifications
      ◢ semantics
          ComposerIntegration.jrag
          ComposerManagement.jadd
          ComposerRewriteIntegration.jra
          CompositionStrategies.jadd
          CompositionSystem.jadd
          isc-composers.ast
      build.xml
```

```
<project name="SkAT4J-Build" default="tpl.build.jar">
 <import file="../org.skat.isc.core/build-base.xml"/>

 <tpl.conf
    astpackage="org.skat.binding.java.ast"
    base-dir="${ant.file.ISC-Java-Binding-Build}"
    grammar="no-parser"
    package="org.skat.binding.java"
    project="org.skat.binding.java"
    version="0.1.0"
 />

 <tpl.conf-build
    jar="skat4j.jar"
    bundle="SkAT4j - ISC for Java"
 />
```

```
▷ Fortran-Build
◢ SkAT4J-Build <taskdef class jastadd.JastAddTask cannot be found
  ▷ Build-Base.tpl.gen.ast [from import ../org.skat.isc.core/build-base.xml]
  ▷ Build-Base.tpl.gen.parser [from import ../org.skat.isc.core/build-base.xml]
  ▷ java.parser
  ▷ java.scanner
  ▷ tpl.build [from import ../org.skat.isc.core/build-base.xml]
  ▷ tpl.build.jar [default] [from import ../org.skat.isc.core/build-base.xml]
  ▷ tpl.cleanGen [from import ../org.skat.isc.core/build-base.xml]
  ▷ tpl.gen.all [from import ../org.skat.isc.core/build-base.xml]
  ▷ tpl.gen.ast
    tpl.gen.parser
▷ TPL-Build
▷ UPP-Build
```

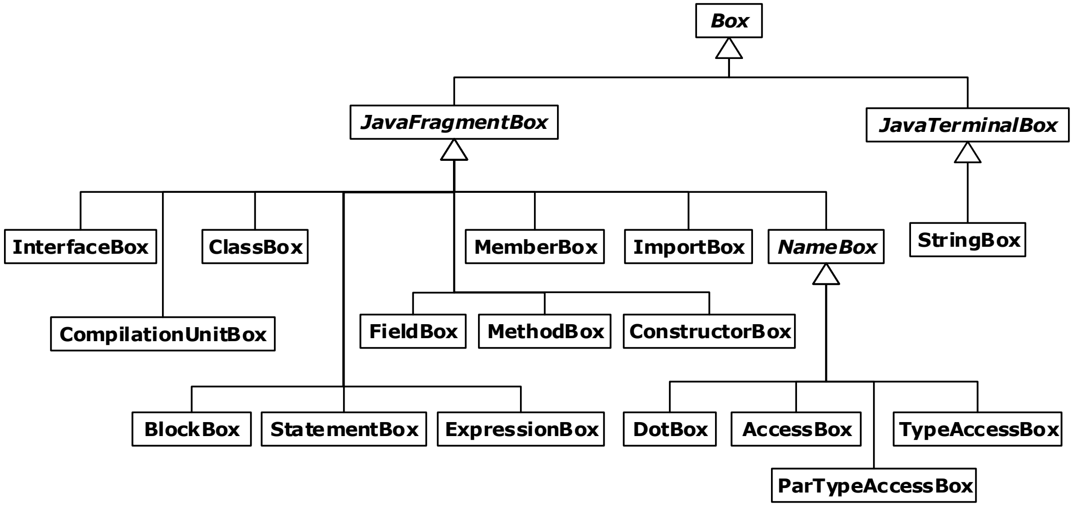
generated jar library

SkAT/Full and SkAT/Core packages

declarative build configuration

Boxology of SkAT4J

```
abstract JavaFragmentBox:Box ::= ;
// Top-level boxes.
CompilationUnitBox:JavaFragmentBox ::= Fragment:CompilationUnit;
ClassBox:JavaFragmentBox ::= Fragment:ClassDecl;
InterfaceBox:JavaFragmentBox ::= Frag

ImportBox:JavaFragmentBox ::= Fragmen
// Member-level boxes.
MethodBox:JavaFragmentBox ::= Fragmen
ConstructorBox:JavaFragmentBox ::= Fr
FieldBox:JavaFragmentBox ::= Fragment
MemberBox:JavaFragmentBox ::= Fragmen

// Block-level boxes.
StatementBox:JavaFragmentBox ::= Frag
ExpressionBox:JavaFragmentBox ::= Fra
BlockBox:JavaFragmentBox ::= Fragment

abstract JavaNameBox:JavaFragmentBox
DotBox:JavaNameBox ::= Fragment:Dot;
TypeAccessBox:JavaNameBox ::= Fragmen
ParTypeAccessBox:JavaNameBox ::= Frag
AccessBox:JavaNameBox ::= Fragment:Access;

abstract JavaTerminalBox:Box ::= ;
StringBox:JavaTerminalBox ::= Fragment:StringValue;
StringValue ::= <Value:String>;
```



SkAT4J: BAF example code generator

```java
private JavaCompositionSystem cSys = new JavaCompositionSystem("baf/","in","out");
public void compositionProgram(File modelFile) throws IOException {
        // Load the BusinessModel object with EMFText.
        BusinessModel bm = loadBusinessModel(modelFile);
        // Configure the composition system.
        cSys.setCompositionStrategy(CompositionSystem.OP_ORDERED_COMPOSITION_FP);
        cSys.setRecoverMode(true);
        // For each role definition, generate a Java class.
        for(RoleDefinition role: preOrder(bm.getRoleDefinitions())){
                // Instantiate template Person.jbx for each role.
                String cuName = role.getName() + ".java";
                cSys.copyBox("Person.jbx",cuName);
                cSys.addBindContent(cuName+"#Type",role.getName());
                cSys.addBindContent(cuName+"#TypeName","\"" + role.getName() + "\"");
                cSys.addBindContent(cuName+"#ImplicitSuperClass","Person");
                cSys.addExtendContent(cuName+"#*.membersEntry","public [[Type]](){super();}");
                cSys.addBindTerminal(cuName+"#Pfx","\n");
                cSys.triggerComposition();
                cSys.clearCompositionProgram();
                // Mix in code of super roles.
                for(RoleDefinition superRole:role.getSuperRoles()){
                        mixin(role.getName() + ".java",superRole.getName() + ".java");
                }
                // Generate code for PropertyDefinitions of the current role.
                for(PropertyDefinition def:concat(role.getProperties(),getSuperProps(role))){
                        // If there’s not already a mixed-in implementation, add members.
                        if(def.eContainer()==role && !isShadowed(def)){
                                cSys.addExtend(cuName+"#*.members","Setter.jbx");
                                cSys.addExtend(cuName+"#*.members","Getter.jbx");
                                cSys.addExtend(cuName+"#*.members","Field.jbx");
                                cSys.addBindContent(cuName+"#Type",def.getType().getTargetType());
                                cSys.addBindContent(cuName+"#SetSfx","set" + toFirstUpper(def.getName()));
                        }
                        // Extend asString().
                        if(def.eContainer()==role && !isShadowed(def) || def.eContainer()!=role){
                                String stmt = "v += \"\\n [[Field]]:\" + [[GetSfx]]();";
                                cSys.addExtendContent(cuName+"#*.asString.methodExit",stmt);
                                cSys.addBindContent(cuName+"#GetSfx","get" + toFirstUpper(def.getName()));
                                cSys.addBindContent(cuName+"#Field",def.getName());
                        }
                        // Extend constructor with defaults.
                        if(def.eContainer()==role && def.getType().getDefault()!=null){
                                String stmt = def.getName() + "=" + def.getType().getDefault() + ";";
                                cSys.addExtendContent(cuName+"#*." + role.getName() + ".statements",stmt);
                        }
                        cSys.triggerComposition();
                        cSys.clearCompositionProgram();
                }
        }
        cSys.persistFragments();}
```

SkAT4J: supported slots

Slot name	Example	Description
MethodDecl	`public class A { void MethodSlot(){}; }`	*A slot for method decls.*
MemberDeclSlot	`public class A{ [[MemberSlot]];}`	*A class-members slot, e.g., fields, methods.*
StmtSlot	`public void m(){ [[StmtSlot]]; }`	*A slot for statements, e.g., declarations, blocks.*
ExprSlot	`Object[] o = new Object[[[ExprSlot]]];`	*A slot for expressions, e.g., initializations.*
VarAccessSlot	`int a = this.[[VarName]];`	*A slot for accessing fields of an object.*
TypeAccessSlot	`[[TypeSlot]] t = new [[TypeSlot]]();`	*A slot for "real" generic types.*
ArrayTypeAccessSlot	`[[TypeSlot]][] t = new [[TypeSlot]][i];`	*A slot for generic arrays.*
GenericTypeAccessSlot	`public void m(A<[[ParSlot]]> arg){}`	*Convenience slot for type parameters.*
TypeVariableSlot	`public class A <[[ParSlot]]>{}`	*Convenience slot for type parameters.*
TypeDeclNameSlot	`public interface [[NameSlot]] {}`	*A slot for interface and class-declaration names.*
MethodDeclNameSlot	`public void [[NameSlot]](){}`	*A slot supporting generic method names.*
VarDeclNameSlot	`private String [[Name]];`	*Slots supporting generic field and variable names.*
MethodAccessNameSlot	`String a = [[NameSlot]]();`	*A slot for generic method calls.*
ConstructorDeclNameSlot	`public [[NameSlot]](){}`	*A slot supporting generic constructor names.*
StringValueSlot	`String s = "SkAT: [[Msg]] [[Msg]].“`	*A slot in string literals.*
ParameterDeclNameSlot	`public void m(String [[Name]]){}`	*A slot for method-parameter names.*

SkAT4J: excerpts from Slots.jrag

```
syn boolean ASTNode.isSlot() = false;
syn String ASTNode.slotName() = "";
eq List.isSlot() = false;
eq Opt.isSlot() = false;
inh boolean ASTNode.isInSlot();
eq CompositionEnvironment.getChild(int i).isInSlot() = false;
```

SkAT/Core

Slots.jrag

```
eq MethodDecl.isSlot() = name().endsWith("Slot");
eq MethodDecl.slotName() = isSlot()?name().substring(0,name().length()-4):"";
eq MethodDecl.getChild(int i).isInSlot() = isSlot();
```

```
public class A { void MethodSlot(){}; }
```

Java FCM

Slots.jrag

```
eq StmtSlot.isSlot() = true;
eq StmtSlot.slotName() = isSlot()?extract(getSlotName(),"[[","]]"):"";
eq StmtSlot.compatibleFragmentTypes() = new Class[]{Stmt.class};
```

```
public void m(){ [[StmtSlot]]; }
```

SkAT4J: supported hooks

Hook name	Example	Description

Block.stmts	`<scope>.statements`
	`.statementsEnd`
	`.methodEntry`
	`.methodExit`
		Hooks to extend the statement list of any block.

CompilationUnit.importDecls	`<scope>.imports`	A hook to extend the list of import declarations.

TypeDecl.bodyDecls	`<scope>.members`
	`.membersEntry`
	`.membersExit`
		Hooks to extend the members of a class or interface declaration.

ClassDecl.implements	`<scope>.implements`	A hook to add new interfaces to a class.
MethodDecl.parameters	`<scope>.parameters`	A hook to add new parameters to a method.

SkAT4J: excerpts from Hooks.jrag

```
syn boolean ASTNode.isHook(List hook) = false;
syn String ASTNode.hookName(List hook) = "";
syn String[] ASTNode.hookAliases(List hook) = new String[0];
syn int ASTNode.hookIndex(List hook, String hookName) = 0;
```

SkAT/Core

Hooks.jrag

```
eq Block.isHook(List hook) = hook == getStmtList() && !isInSlot();
eq Block.hookName(List hook) = "statements";
eq Block.hookAliases(List hook) {
    if(isMethodRootBlock()){
      return new String[] {"methodEntry","methodExit","statementsEnd"}; }
    else if(!isMethodRootBlock() && endsWithReturn()){
      return new String[] {"methodExit" + numReturns(),"statementsEnd"};
    }
    else return new String[] {"statementsEnd"};
}
eq Block.hookIndex(List hook, String hookName) {
    if("methodEntry".equals(hookName) || "statements".equals(hookName)){
      return 0;
    }
    else if (hookName.startsWith("methodExit") && endsWithReturn()){
      return hook.numChildren()-1;
    }
    else return hook.numChildren();
}
```

Java FCM

Hooks.jrag

SkAT4J: supported fragment assertions

Attributes	**Description**

`syn TypeDecl.assertNotDeclared(MethodDecl)`
`// Characteristic Attributes:`
`syn MethodDecl.signature()`
`syn TypeDecl.localMethodsSignatureMap()`
	Checks method redeclarations

`syn TypeDecl.assertNotDeclared(FieldDeclaration)`
`// Characteristic Attributes:`
`syn TypeDecl.localFieldsMap()`
	Checks field redeclarations

`syn TypeDecl.assertVariablesProvided(ASTNode)`
`// Characteristic Attributes:`
`*syn ASTNode.danglingVars()`
`syn TypeDecl.memberFields(String)`
`inh TypeDecl.lookupVariable(String)`
	Checks variables in scope

`syn TypeDecl.assertMethodsProvided(ASTNode)`
`// Characteristic Attributes:`
`*syn MethodDecl.danglingCalls()`
`syn TypeDecl.memberMethods(String)`
`inh TypeDecl.lookupMethod(String)`
	Checks methods in scope

`syn ExprSlot.assertCompatibleType(Expr expr)`
`// Characteristic Attributes:`
`syn Expr.type()`
`syn TypeDecl.wideningConversionTo(TypeDecl)`
	Checks expression types

SkAT4J: excerpts from Assertions.jrag

```
eq ExprSlot.checkContractPre(Object fragment) = true;
eq ExprSlot.checkContractPost(Object fragment) {
     if(fragment instanceof Expr){
          Object result = assertCompatibleType((Expr)fragment);
          if(result!=Boolean.TRUE)
               return result;
     }
     return true;
}
```

Java FCM

Slots.jrag

```
syn Object ExprSlot.assertCompatibleType(Expr fgmt){
     ...
     AssignExpr parent = = (AssignExpr)expr.getParent() ;
     if(parent.getSource() == expr){
          TypeDecl sourceType = fgmt.type();
          TypeDecl destType = parent.getDest().type();
          if(sourceType == expr.unknownType() || destType==expr.unknownType()
               || !sourceType.wideningConversionTo(destType))
          return "Type of Expr fragment ’" + sourceType.getID()
               + "’ does not fit left-hand type ’" + destType.getID() + "’.";
     }
     ...
     return true;
}
```

Java FCM

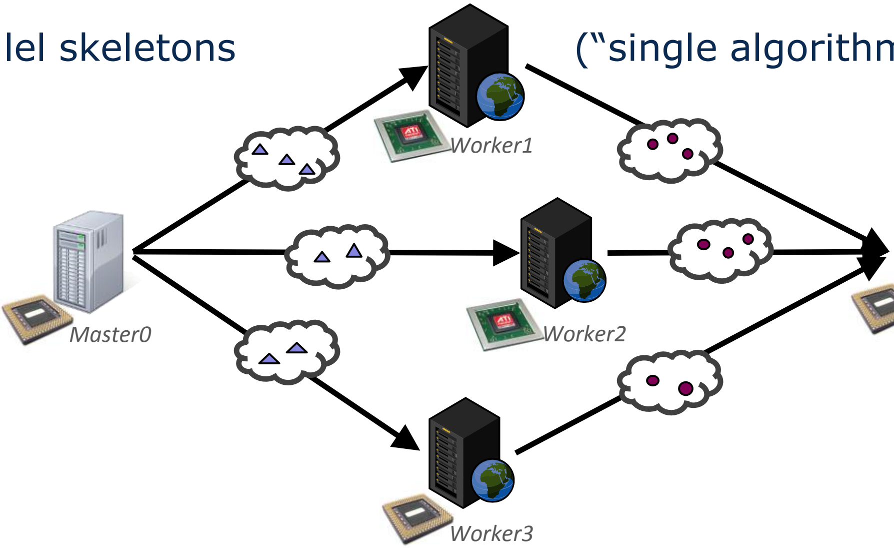
Assertions.jrag

Fragment contracts – potentials

- **Error detection:** contracts are checked before/after a composition → problematic composition steps can be detected.
  - Alternatively a compiler could afterwards find it via some trace links
- **Composition control:** if a contract is not fulfilled at the beginning of the composition, it might still be fulfilled later.
- **Efficiency:** Caching mechanisms of AGs can make the approach more efficient than a complete re-evaluation/re-compilation
- **Expressivity:** contract conditions can contain more information than just the information derived from the fragments (characteristic vs. assertion).
  - Example: access restriction to a certain variable (assertion: provided={a,b,…,z}, required={a,b,…,z}, condition: fits(A) if A.required ⊆ S.provided and not b ϵ required)
- **Fragment selection/conditional composition:** select fragment components fitting to a certain condition or assertion

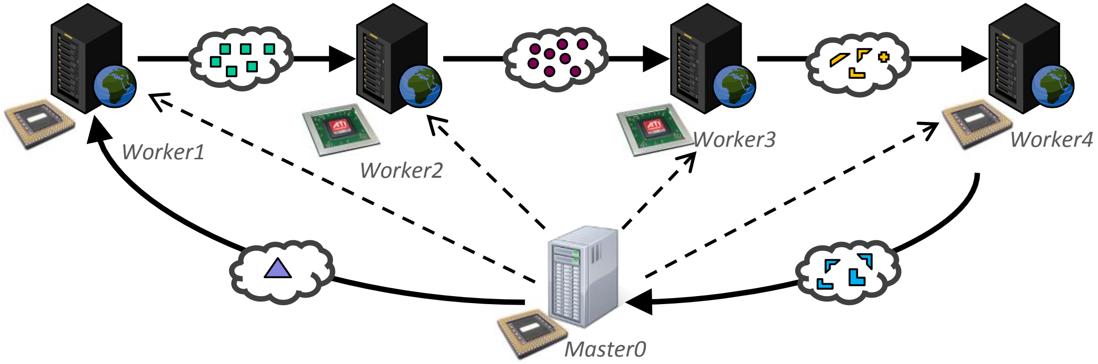
Skeletons: basic forms

→ data-parallel skeletons (“single algorithm multiple nodes”)



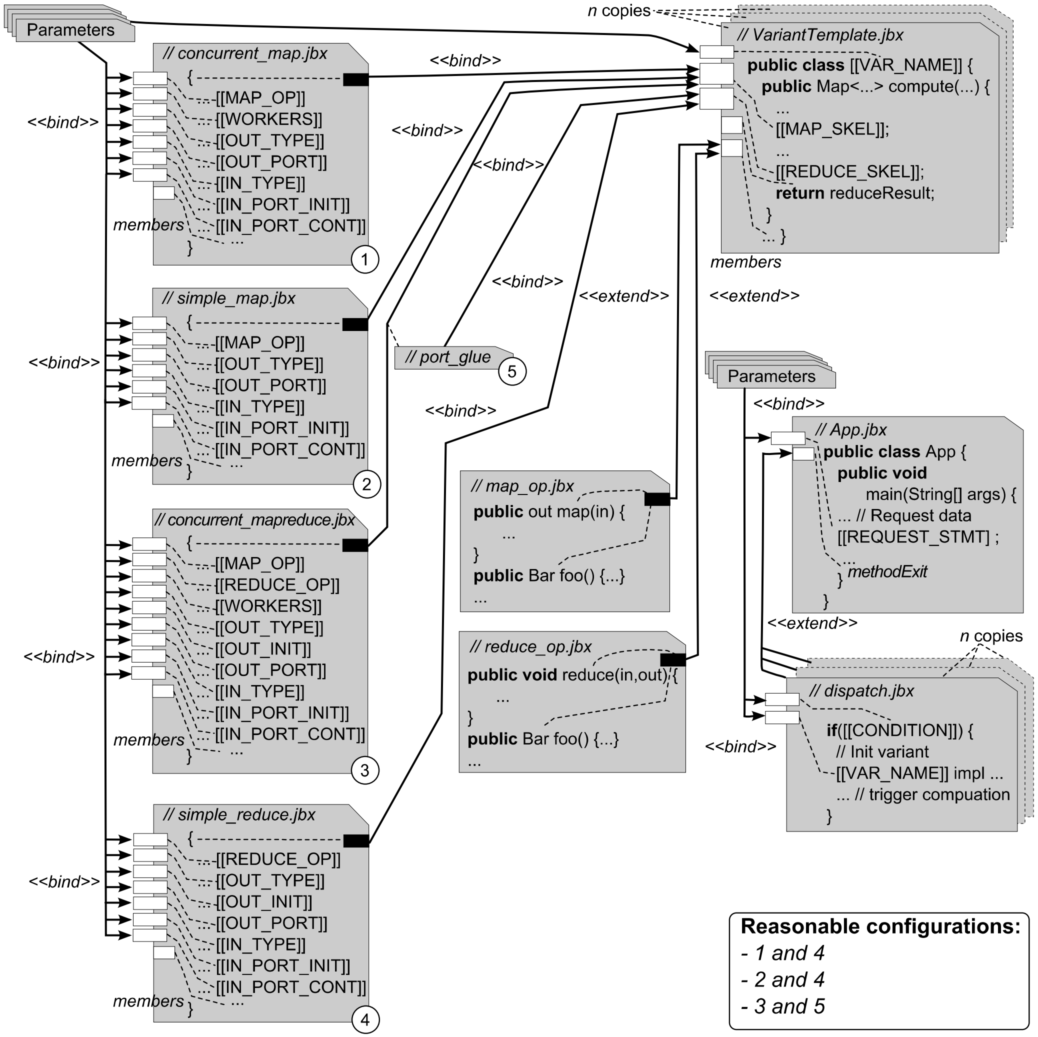
(e.g. Map)

→ task-parallel skeletons (“multiple algorithms multiple nodes”)



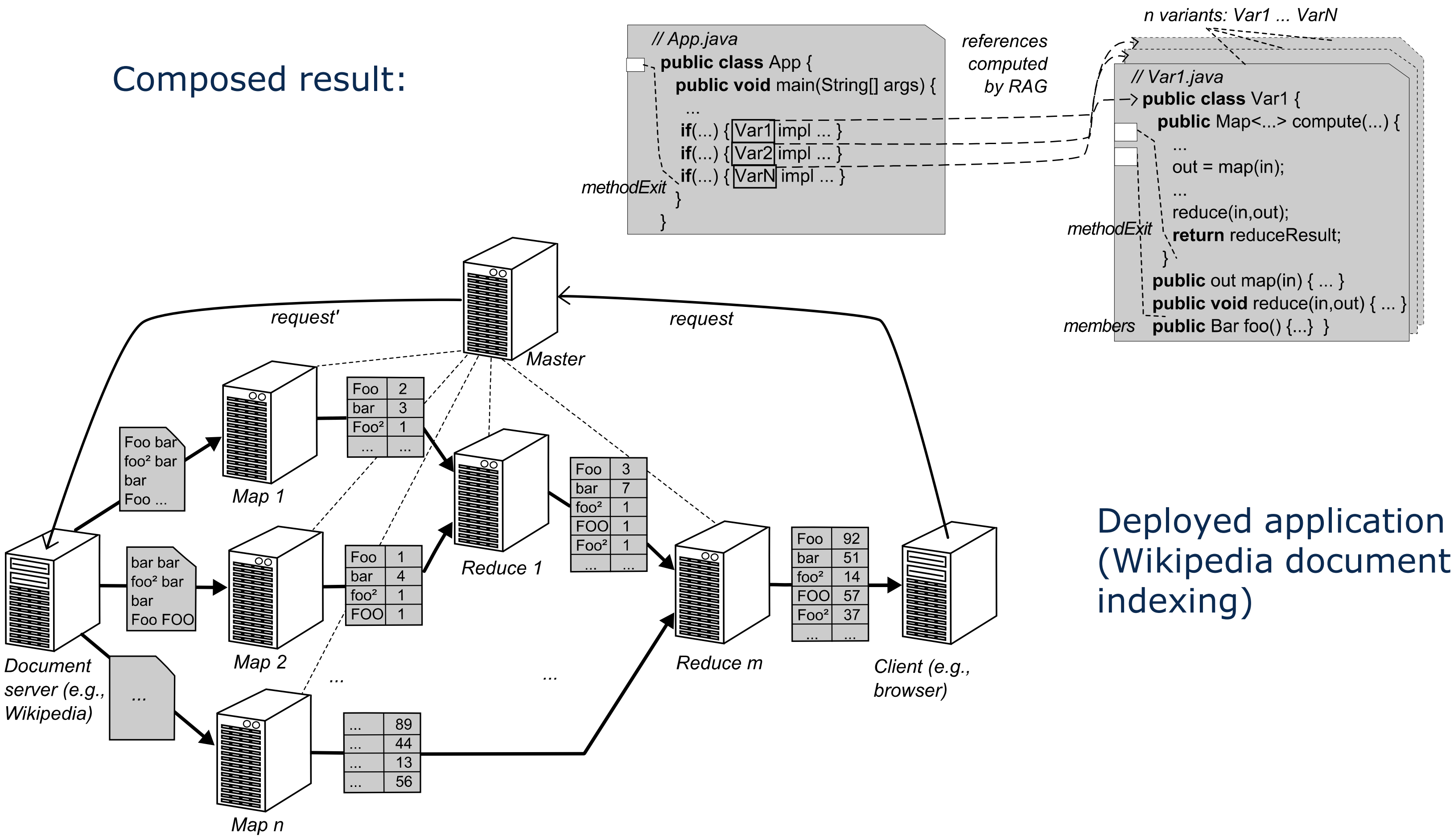
(e.g. Pipe)

SkAT4J: a skeleton library



Reasonable configurations:
- *1 and 4*
- *2 and 4*
- *3 and 5*

SkAT4J: a skeleton library



Composed result:

```
// App.java
public class App {
  public void main(String[] args) {
   ...
   if(...) {Var1 impl ... }
   if(...) {Var2 impl ... }
   if(...) {VarN impl ... }
  }
}
```

```
// Var1.java
public class Var1 {
  public Map<...> compute(...) {
    ...
    out = map(in);
    ...
    reduce(in,out);
    return reduceResult;
  }
 public out map(in) { ... }
 public void reduce(in,out) { ... }
 public Bar foo() {...}  }
```

Foo bar
foo^2 bar
bar
Foo ...

bar bar
foo^2 bar
bar
Foo FOO

...

Foo	2
bar	3
Foo^2	1
...	...

Foo	1
bar	4
foo^2	1
FOO	1

...	89
...	44
...	13
...	56

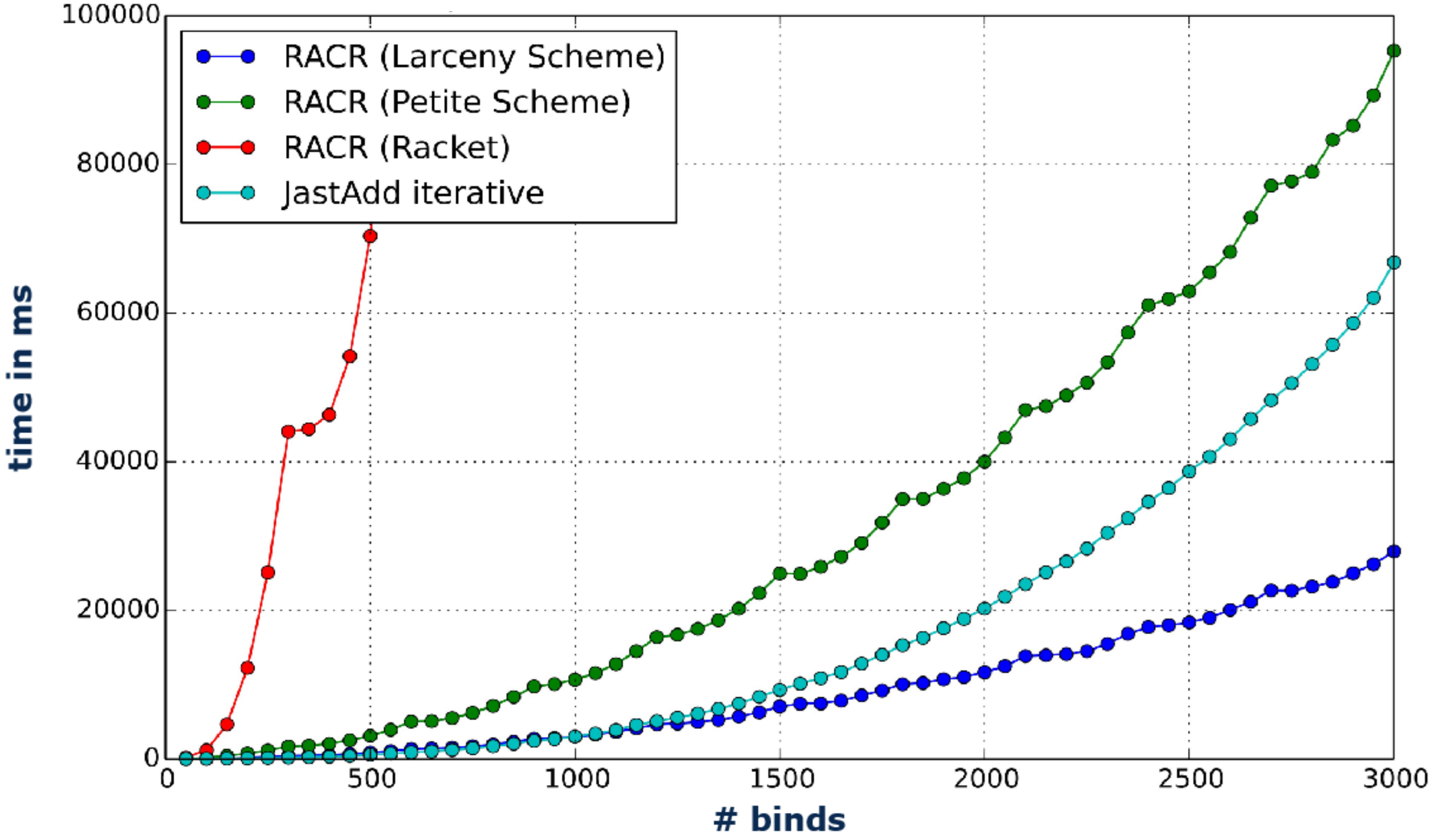
Foo	3
bar	7
foo^2	1
FOO	1
Foo^2	1
...	...

Foo	92
bar	51
foo^2	14
FOO	57
Foo^2	37
...	...

Deployed application (Wikipedia document indexing)

Performance: ISC in JastAdd and RACR

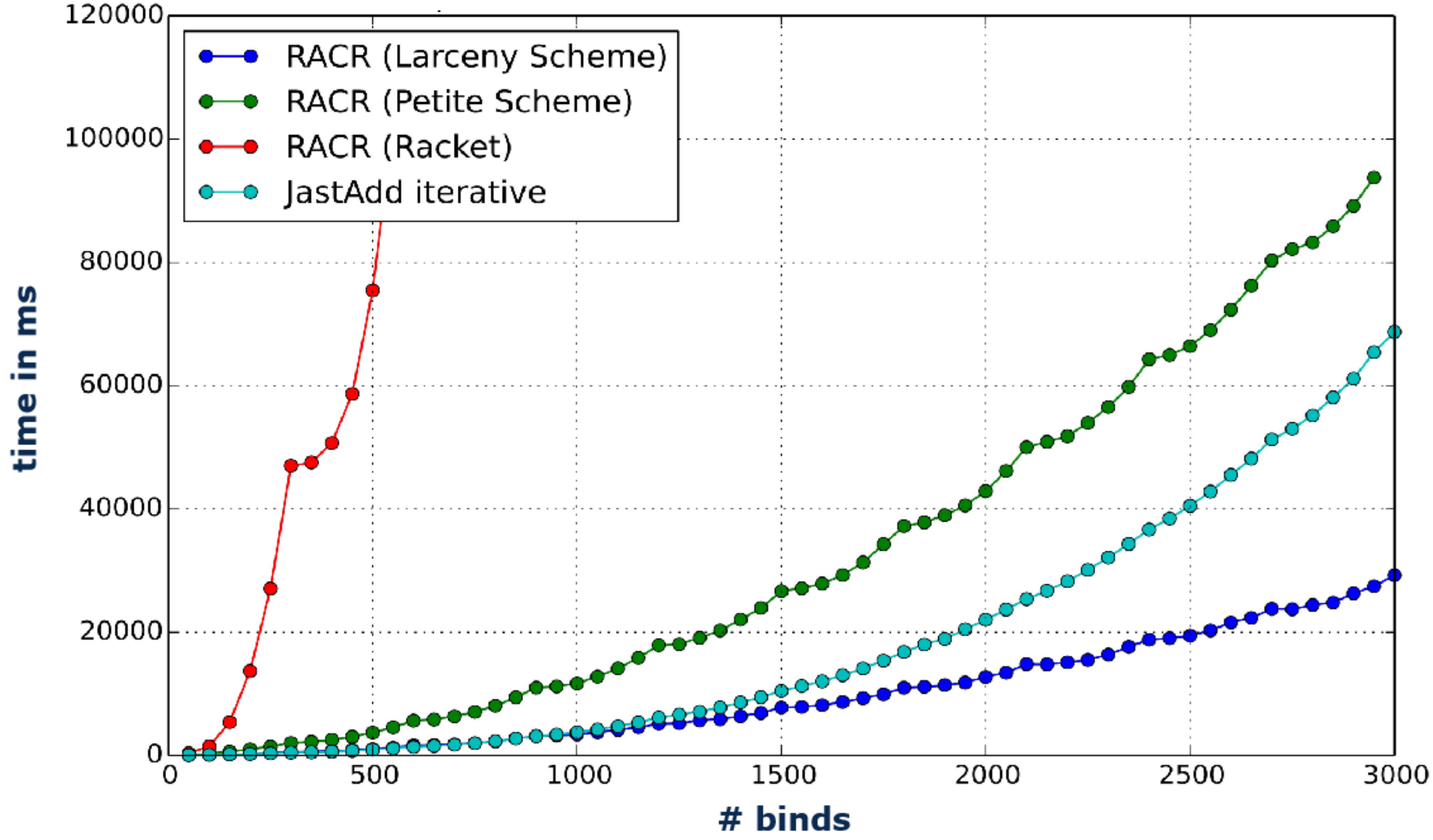
Composing Java classes: no well-formedness check; deep tree



(experiments by [Tasic14])

Performance: ISC in JastAdd and RACR

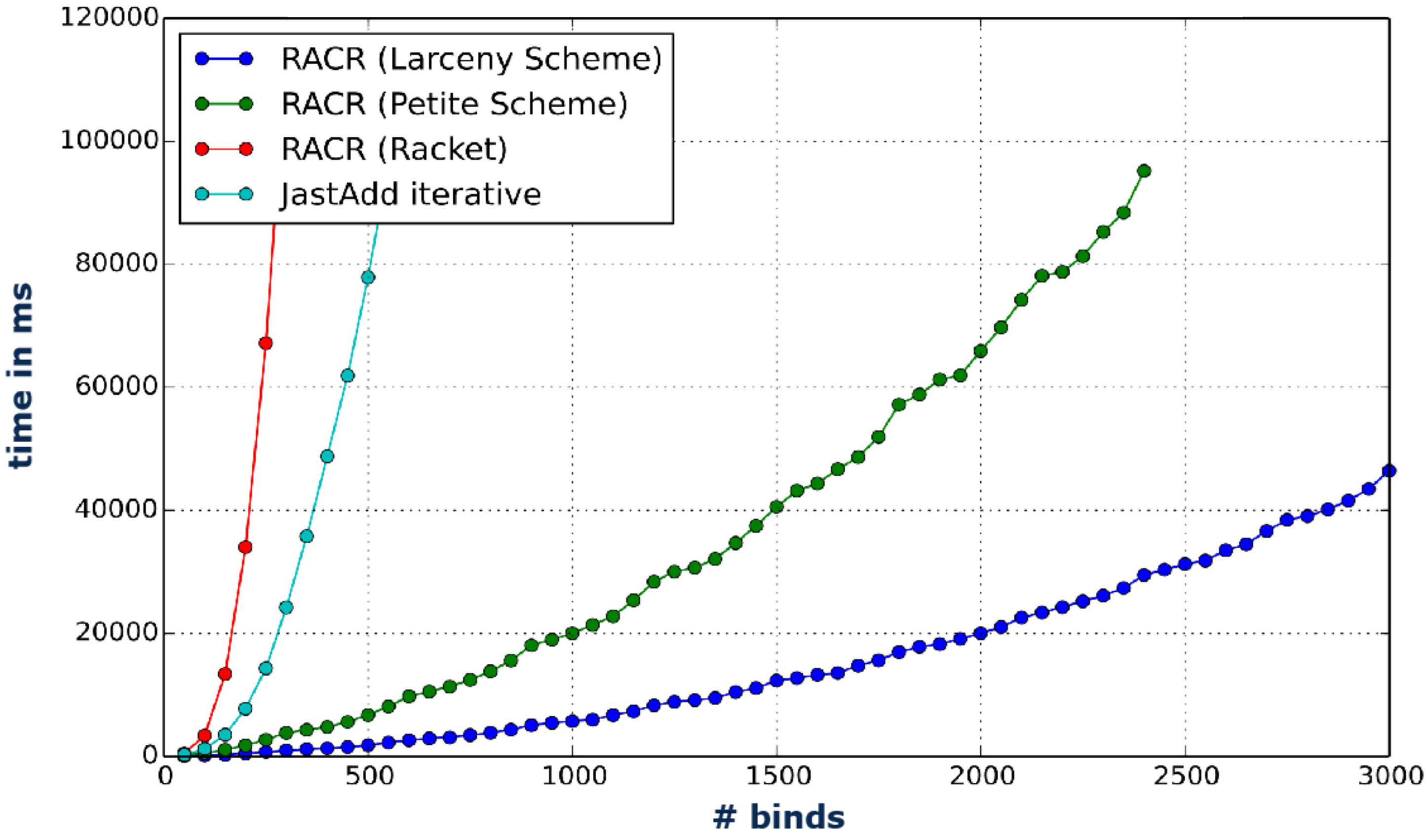
Composing Java classes: 1x well-formedness check; deep tree



(experiments by [Tasic14])

Performance: ISC in JastAdd and RACR

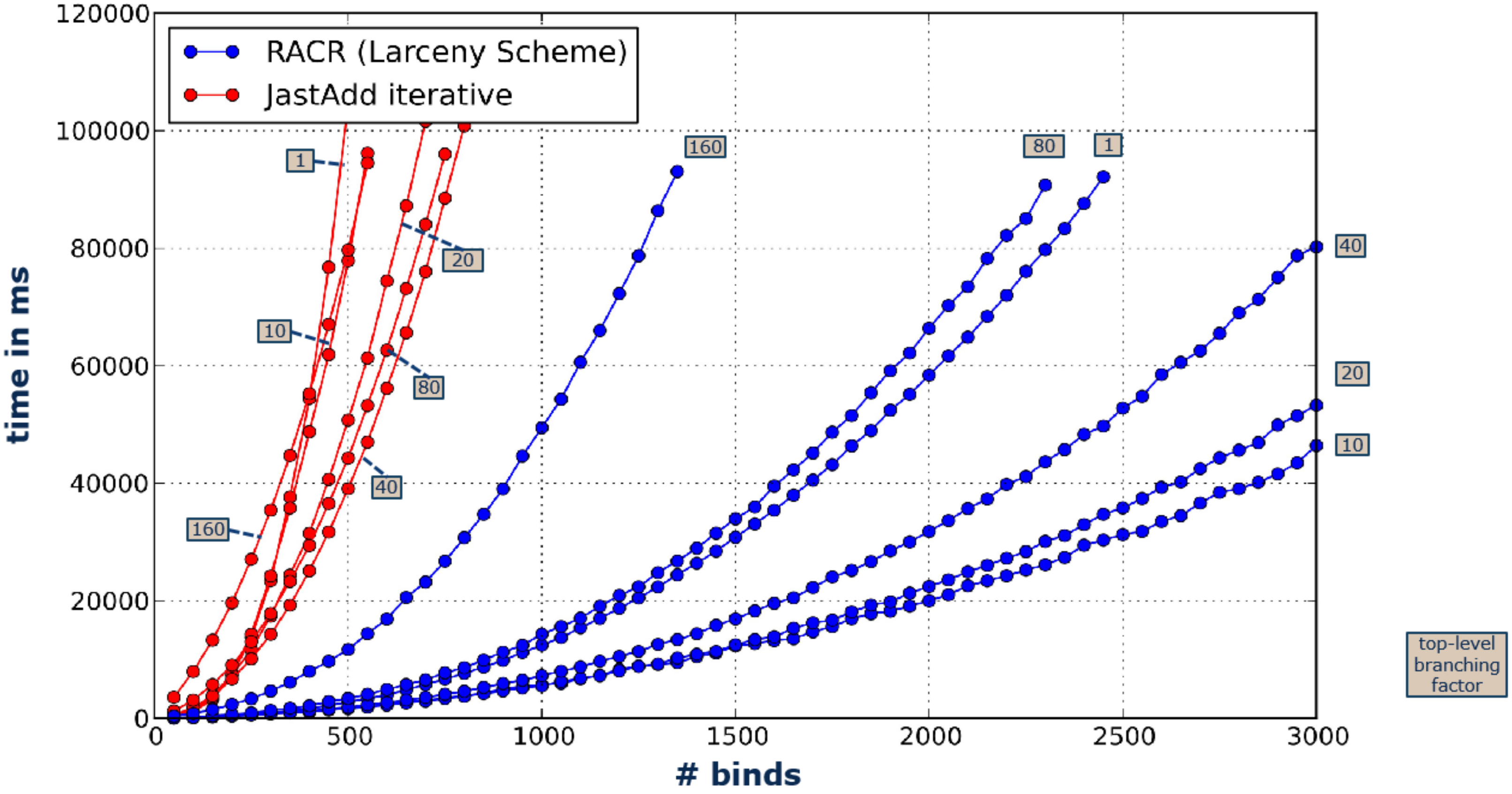
Composing classes: rep. well-formedness check; deep tree



(experiments by [Tasic14])

Performance: ISC in JastAdd and RACR

Composing classes: rep. well-formedness check; diff. trees



(experiments by [Tasic14])

Scalability in Software Engineering

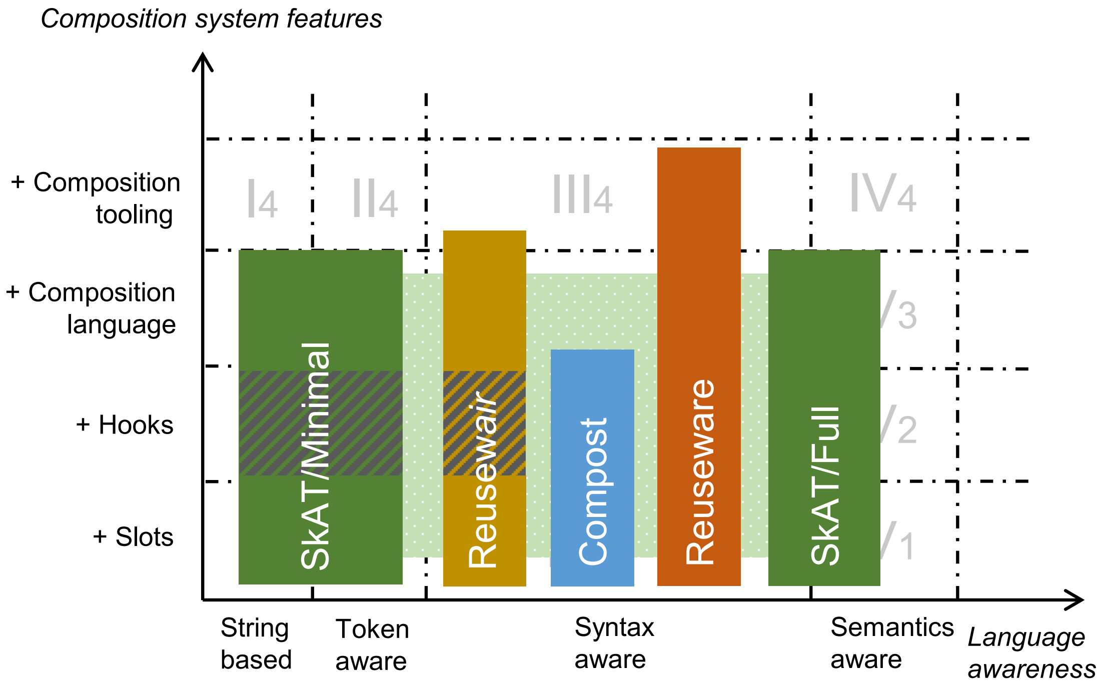
“In order to make the term [scalability] meaningful, it has to be understood within a particular context and then regarded as variation within a range. Scalability in the context of software engineering is the property of reducing or increasing the scope of methods, processes, and management according to the problem size” [Laitinen+00].

“Functional scalability: The ability to enhance the system by adding new functionality at minimal effort” [Wikipedia/Scalability].

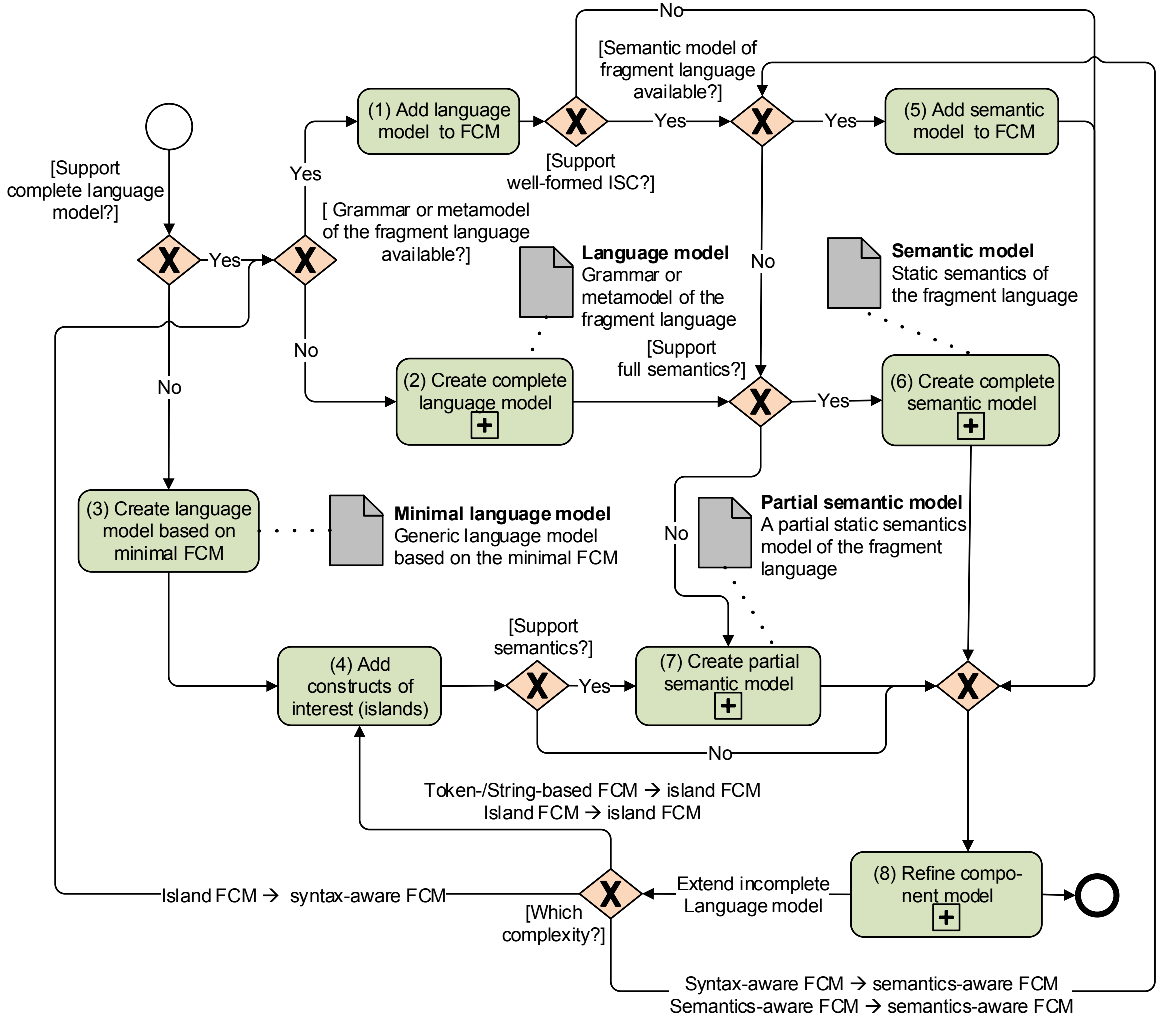
[Laitinen+00] Laitinen, Mauri, Mohamed Fayad, und Robert P. Ward. „The problem with scalability.“ *Communications of the ACM* 43, Nr. 9 (2000): 105–7.

Supported features of composition systems

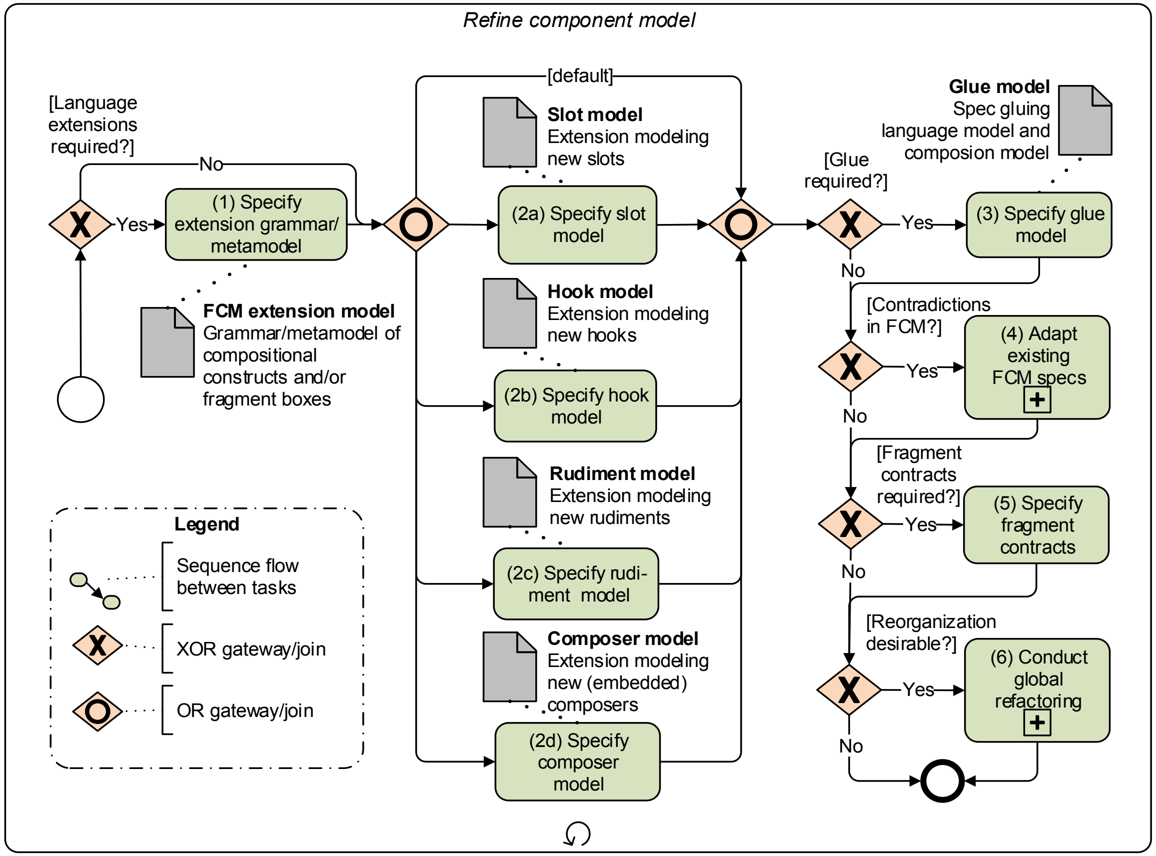
- Existing frameworks are syntax-aware
- SkAT scales over the whole range of language awareness



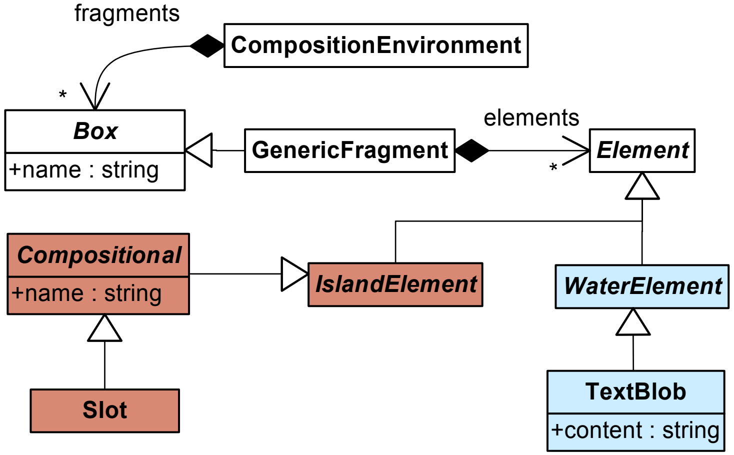
BPMN workflow of agile composition-system development



BPMN workflow of complex refinement activity



Minimal fragment component model



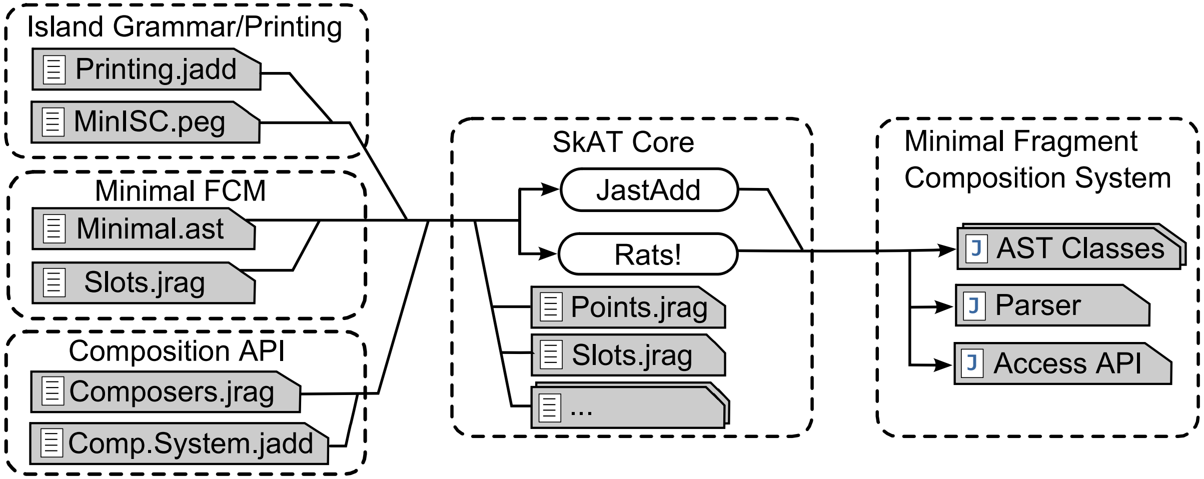
```
CompositionEnvironment ::= fragments:Box*
@Box ::= name:<string>
GenericFragment ▷ Box ::= elements:Element*
@Element ::=
@WaterElement ▷ Element ::=
TextBlob ▷ WaterElement ::= content:<string>
@IslandElement ▷ Element ::=
@Compositional ▷ IslandElement ::=
                          name:<string>
Slot ▷ Compositional ::=
```

$$\begin{aligned} \textbf{fun}\ \texttt{Slot.isSlot} &= \mathit{true}\\ \textbf{fun}\ \texttt{Slot.slotName} &= \mathit{name}\\ \textbf{fun}\ \{n \mid n \in N - \mathcal{S}\}\texttt{.isSlot} &= \mathit{false}\\ \textbf{fun}\ \{n \mid n \in N - \mathcal{S}\}\texttt{.slotName} &= \perp \end{aligned}$$

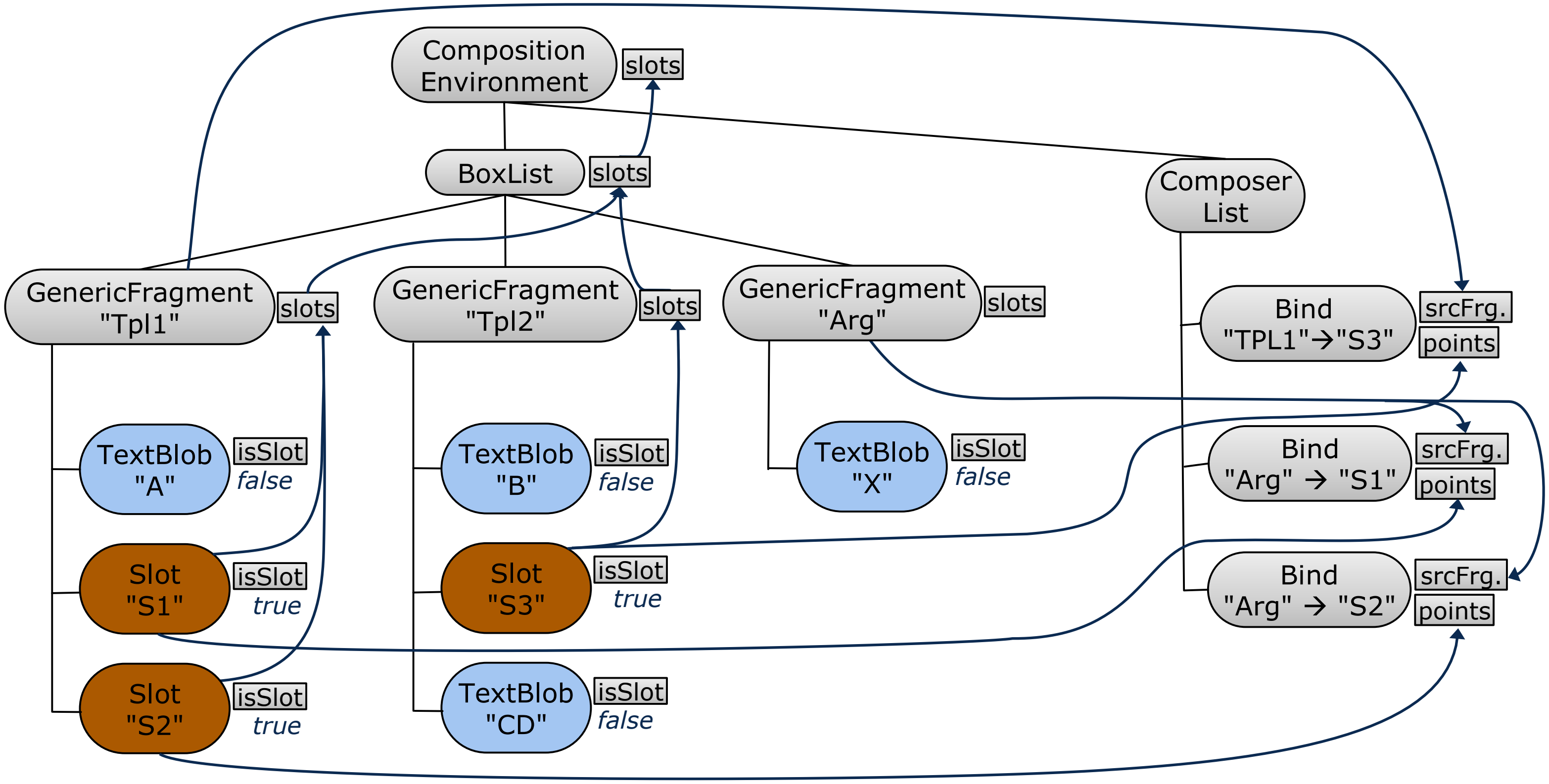
$$\begin{aligned} \textbf{fun}\ \{n \in N\}\texttt{.child}_{\texttt{all}}\texttt{.isHook} &= \mathit{false}\\ \textbf{fun}\ \{n \in N\}\texttt{.child}_{\texttt{all}}\texttt{.hookName} &= \perp \end{aligned}$$

Minimal ISC code generation

- includes the minimal component model
- includes an extensible parser module based on PEGs

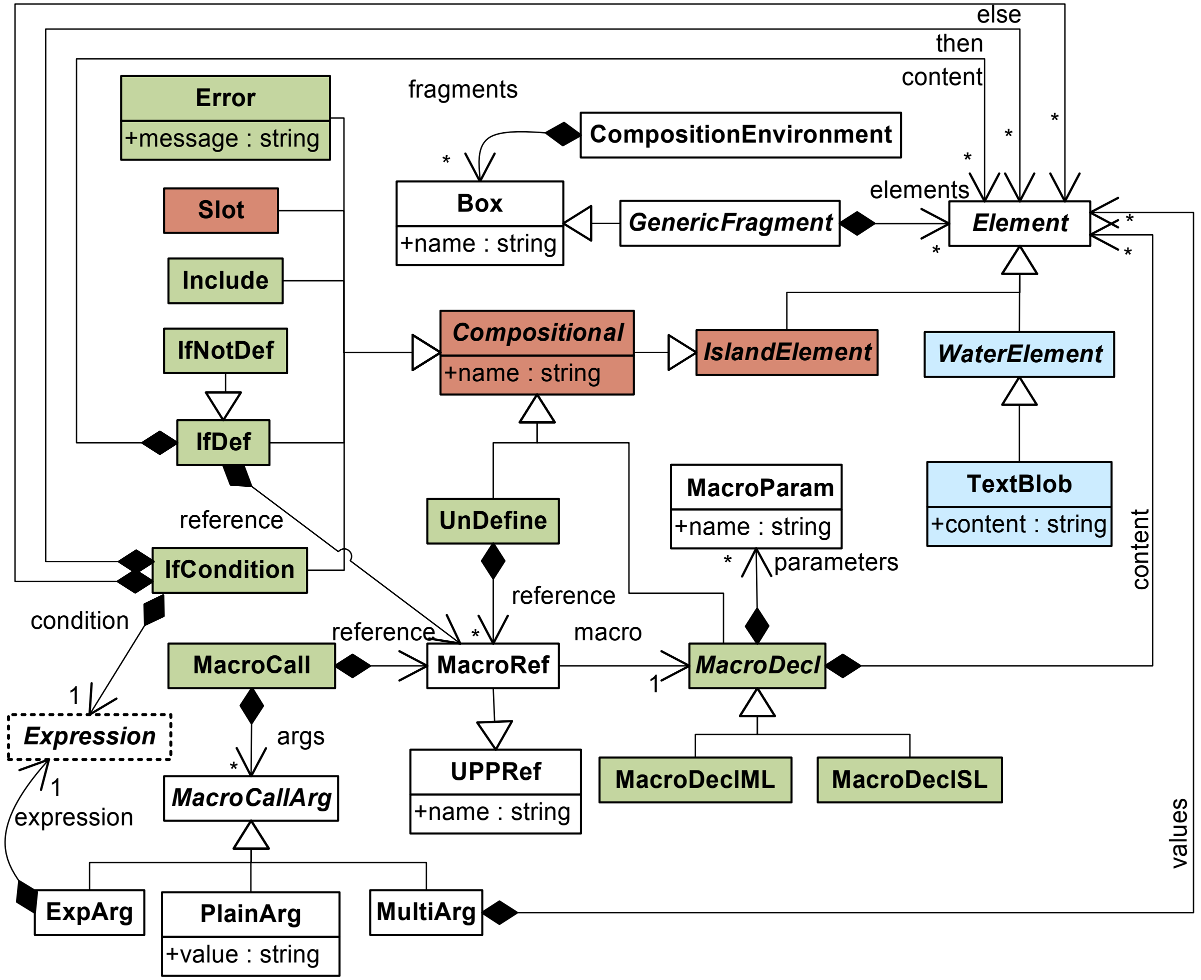


Minimal fragment component model



A[[S1]][[S2]] **B[[S3]]CD** **X** **BAXXCD**

UPP: fragment component model

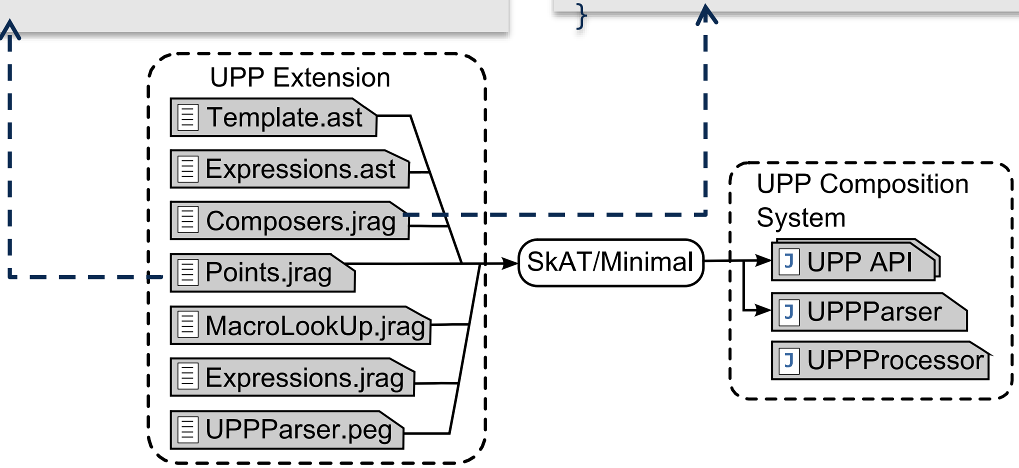


UPP: fragment component model + embedded composers

embedded composition operators

```
aspect UPPPoints {
    // identifying slots
    eq Include.isSlot() = true;
    eq IfDef.isSlot() = true;
    eq IfNotDef.isSlot() = true;
    eq IfCondition.isSlot() = true;
    eq MacroCall.isSlot() = true;

    // identifying rudiments
    eq Error.isRudiment() = true;
    eq MacroDecl.isRudiment() = true;
    eq UnDefine.isRudiment() = true;

    // determining point names
    eq Compositional.pointName() = getName();
}
```

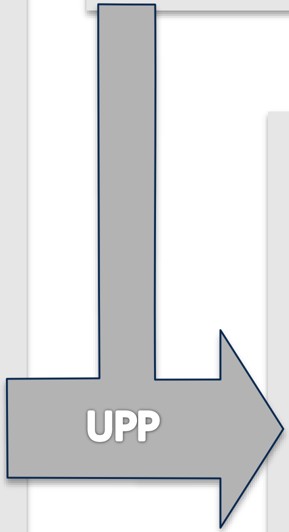
```
aspect IncludeComposer {
    syn GenericFragment Include.associatedFragment();
    eq Include.isBind() = true;
    eq Include.targetPoint() = this;
    eq Include.composer() = this;
    eq Include.associatedFragment() =
                  findFragment(getName());
    eq Include.srcFragment() {
        GenericFragment fragment =
associatedFragment();
        if(fragment!=null){
            return fragment.getElements().fullCopy();
        }
        return null;
    }
}
```



UPP example: 2D and 3D vectors in Java

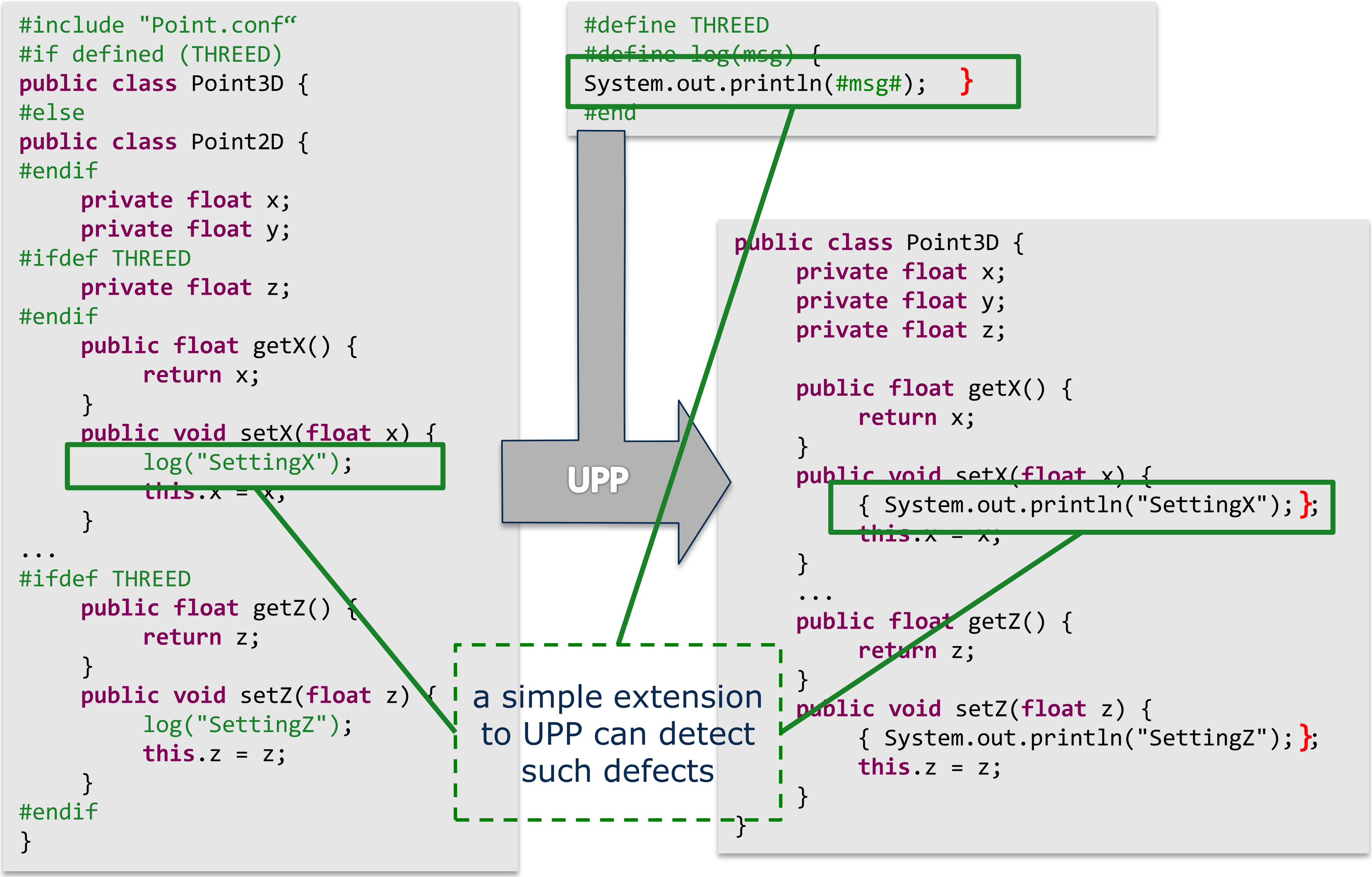
```
#include "Point.conf“
#if defined (THREED)
public class Point3D {
#else
public class Point2D {
#endif
     private float x;
     private float y;
#ifdef THREED
     private float z;
#endif
     public float getX() {
          return x;
     }
     public void setX(float x) {
          log("SettingX");
          this.x = x;
     }
...
#ifdef THREED
     public float getZ() {
          return z;
     }
     public void setZ(float z) {
          log("SettingZ");
          this.z = z;
     }
#endif
}
```

```
#define THREED
#define log(msg) {
System.out.println(#msg#);
#end
```



```
public class Point3D {
     private float x;
     private float y;
     private float z;

     public float getX() {
          return x;
     }
     public void setX(float x) {
          { System.out.println("SettingX"); ;
          this.x = x;
     }
     ...
     public float getZ() {
          return z;
     }
     public void setZ(float z) {
          { System.out.println("SettingZ"); ;
          this.z = z;
     }
}
```

UPP example: 2D and 3D vectors in Java



```
#include "Point.conf“
#if defined (THREED)
public class Point3D {
#else
public class Point2D {
#endif
      private float x;
      private float y;
#ifdef THREED
      private float z;
#endif
      public float getX() {
            return x;
      }
      public void setX(float x) {
            log("SettingX");
            this.x = x;
      }
...
#ifdef THREED
      public float getZ() {
            return z;
      }
      public void setZ(float z) {
            log("SettingZ");
            this.z = z;
      }
#endif
}
```

```
#define THREED
#define log(msg) {
System.out.println(#msg#);   }
#end
```

```
public class Point3D {
      private float x;
      private float y;
      private float z;

      public float getX() {
            return x;
      }
      public void setX(float x) {
            { System.out.println("SettingX"); };
            this.x = x;
      }
      ...
      public float getZ() {
            return z;
      }
      public void setZ(float z) {
            { System.out.println("SettingZ"); };
            this.z = z;
      }
}
```

a simple extension to UPP can detect such defects

Part IV: Related Work

Safe template languages

- **Repleo** [Arnoldus11]
  - declarative approach to specify syntax-safe template languages
  - placeholders, iterations, control flow
  - based on SDF [Visser97]
- **Model-aware templates** [Heidenreich+09]
  - automatic metamodel transformation to add template constructs
  - semantics via generated stubs (not declarative)
  - based on EMFText
- **SafeGen** [Huang+11]
  - language for safe template-metaprogramming in Java
  - correctness is ensured via theorem proving
  - 1st order axioms specify well-formedness
  - less expressive than RAGs

Part IV: Related Work

Macros and preprocessors

- **Metamorphic syntax macros** [Brabrand+Schwartzbach02]
  - macro constructs are transformed ("metamorphed") into host language syntax
  - related to embedded ISC [Henriksson09]
  - supported in SkAT via generalized composer definitions (e.g., in the UPP)

- **Language-aware preprocessors**
  - preprocessor directives have bad impact on program quality [Kästner+Apel09]
  - variability-aware code analysis helps to detect errors early
  - tools like TypeChef help with that [Kästner+11]
  - [Heumüller+14] suggest island grammars as a potential approach
  - SkAT/UPP is based on island grammars and attribute grammars

Aspect languages

- **Static aspects in JastAdd**
  - inter-type declarations are a basic composition system
  - partially uses attributes for analysis
  - backend could realized as a well-formed ISC system (bootstrap)

- **AspectBench Compiler** [Avgustinov+06]
  - extensible component model for AspectJ [Kiczales+01]
  - based on delegation and visitor pattern
  - non-declarative
  - case study with JastAdd as backend in [Avgustinov+08]

- **Generic Aspects** [Lohmann+04]
  - aspects are extended with template features
  - can access generic type information

Part IV: Related Work

General approaches to software composition

- **Algebraic Hierarchical Equations for Application Design (AHEAD)** [Batory+04]
  - refining containment hierarchies based on composition equations
  - generalization of mixin-based inheritance
  - refinement composition operators could well be realized using ISC
    - *future work*

- **Choice calculus** [Erwig+Walkingshaw11]
  - formal representation of software variation as "choices"
  - may represent a subset of ISC
  - recent extension to lambda calculus [Chen+14]
    - considers object language and type system

Part IV: Related Work

Formal models of ISC

- **F-Logic** [Azurat07]
  - combines OO-features with resolution-based reasoning [Kifer+95]
  - enables automatic reasoning over component models and compositions
  - only sketch of an approach
    - exemplary modeling of components, hooks and composers

- **Theorem proving** [Kezardi+12,Kezardi+14]
  - formalization of basic ISC multigraphs and set theory
  - semi-automatic verification using theorem prover
  - first version checks syntactic integrity only
  - second version respects metamodel properties
  - pre- and post conditions are suggested to check well-formedness constraints [Kezardi+14]

Part IV: Related Work

Metaprogramming and rewriting

- **Stratego/XT** [Bravenboer+08] **and TXL** [Cordy06]
  - based on *term rewriting*
  - rules with matching patterns and replacement patterns
  - dynamic rewrite rules enable context-sensitive rewriting (Stratego)
  - programmable rewrite strategies (Stratego)
  - first-order functional programming for traversals and rewrite guards (TXL)

- **Design Maintenance System (DMS)** [Baxter+04]
  - commercial program transformation and analysis toolkit
  - support for attribute grammars and term rewriting
  - more than 30 languages supported
    - *potential implementation platform for well-formed ISC*

- **Reference attribute grammar controlled rewriting** [Bürger12]
  - RAG library for scheme that supports rewriting
  - alternating phases of term rewriting and attribute evaluation
  - sophisticated automatic caching of attributes under presence of rewriting
    - *potential implementation platform for well-formed ISC*