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
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 ...
How can we support all kinds of composition tools?

**Invasive Software Composition** [Aßmann03]
(and related approaches)
Example: Composing Java fragments

Fragment „Car“

```java
public class Car {
    private double speed;
    public double getSpeed()
        return 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;
}
```

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

08.10.2015 Well-Formed and Scalable ISC Slide 7 of 85
Example: Composing Java fragments

Fragment "Car"

```java
public class Car {
    private double speed;
    public double getSpeed(){
        return speed;
    }
    [[brakeSlot]]
}
```

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;
}
```

**Fragment component model:** interface definition
- **Slot:** *explicitly* declared variation point in a fragment
- **Hook:** *implicitly* extension point in a fragment
Example: Composing Java fragments

Fragment „Car“

```java
public class Car {
    private double speed;
    public double getSpeed()
        return speed;
    }

[brakeSlot]
```

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;
}
```

**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, ...)

Part I: Invasive Software Composition (ISC)
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“

```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;
}
```
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;
}
```
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;
}
```
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;
}
```
Example: Composing Java fragments

Fragment „Car“

```java
public class Car {
    private double s Piet;
    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;
    }
```

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“

```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;
}
```

Problem(s) in composition program:
1. line 1: cannot bind ‘brake’ to slot ‘brakeSlot’ – ‘speed’ not visible at ‘brakeSlot’
2. 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.)

Well-Formed and Scalable ISC
Example: Composing Java fragments (cont.)

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;
        }
        speed = speed * 3;
        Log.log("Speed is "+ speed);
        return speed;
    }

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

Fragment „log“

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

Fragment „badbrakes“

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

speed = 180:
> Speed is 108
Example: Composing Java fragments (cont.)

**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;
        Log.log(“Speed is ” + speed);
        speed = speed * 3;
        return speed;
    }
    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

**Fragment „log“**
```java
Log.log(“Speed is “ + speed);
```

**Fragment „badbrakes“**
```java
-- 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.
Part I: Problem Analysis

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

Well-formed ISC
### Part I: Thesis Contributions

#### C1: Well-formed invasive software composition

<table>
<thead>
<tr>
<th>Compartment</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C1a</strong>: Fragment component models based on <em>reference attribute grammars</em> (RAGs)</td>
<td>P1 (validation)</td>
</tr>
<tr>
<td>- declarative specification of component models</td>
<td>P3 (dev. costs)</td>
</tr>
<tr>
<td>- supports context-sensitive constraints</td>
<td></td>
</tr>
<tr>
<td>- extensible specification mechanism</td>
<td></td>
</tr>
<tr>
<td><strong>C1b</strong>: Advanced composition technique based on strategies and rewrites</td>
<td>P2 (interaction)</td>
</tr>
<tr>
<td><strong>C1c</strong>: Fragment contracts</td>
<td></td>
</tr>
<tr>
<td>- integrate static semantics into composition</td>
<td>P1 (validation)</td>
</tr>
<tr>
<td>- produce cause-related error messages</td>
<td></td>
</tr>
</tbody>
</table>
Part I: Thesis Contributions

Scalable ISC

Well-formed ISC
Part I: Thesis Contributions

C2: Scalable invasive software composition

<table>
<thead>
<tr>
<th>Compartment</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C2a</strong>: Minimal invasive software composition</td>
<td>P3 (dev. costs)</td>
</tr>
<tr>
<td>- string-based ISC</td>
<td></td>
</tr>
<tr>
<td>- minimal fragment component model</td>
<td></td>
</tr>
<tr>
<td><strong>C2b</strong>: Island fragment component models</td>
<td></td>
</tr>
<tr>
<td><strong>C2c</strong>: Agile composition system development</td>
<td></td>
</tr>
<tr>
<td>- development in small iterations</td>
<td></td>
</tr>
<tr>
<td>- rapid prototyping</td>
<td></td>
</tr>
<tr>
<td>- minimal, classic and well-formed ISC in one process</td>
<td></td>
</tr>
</tbody>
</table>
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

Minimal composition system 1

Extended/Refined island 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-agnostic FCM

Extended/Refined island composition system 1

Syntax-aware & some semantics

Syntax-agnostic FCM

Extended/Refined island composition system 1

Syntax-aware & some semantics

Syntax-agnostic FCM

Extended/Refined island composition system 1

Syntax-aware & some semantics

Syntax-agnostic FCM

Extended/Refined island composition system 1

Syntax-aware & some semantics

Syntax-agnostic FCM

Extended/Refined island composition system 1

Syntax-aware & some semantics

Syntax-agnostic FCM
Part I: Thesis Contributions

SkAT framework

Scalable ISC

Well-formed ISC
C3: SkAT composition framework

<table>
<thead>
<tr>
<th>Compartment</th>
<th>Implements</th>
</tr>
</thead>
</table>
| **C3a**: Generic implementation of well-formed ISC | C1a (FCM RAGs)  
C1b (strategies)  
C1c (contracts) |
| **C3b**: SkAT4J – a well-formed composition system for Java | C1 demonstrator |
| **C3c**: A library of parallel algorithmic skeletons | C2a (minimal ISC)  
C2b (island FCMs) |
| **C3d**: Implementation of scalable ISC | C2 demonstrators |
| **C3e**: Three extensible island composition tools | |
| o STpL – Slot Template Language      | |
| o VTpL – Variability Template Language | |
| o UPP – Universal Extensible Preprocessor | |
C3: SkAT composition framework – modular architecture

- "Sk
eleons and Application Templates”
- Implementation in Java
  - using JastAdd reference attribute grammars (RAGs) [Hedin00]
  - following the principles of extensible compiler construction [Ekman06]

---

**Composition Tools**

- Applications
  - BAF Code Generator, JastAdd Backend, Skeleton Library
  - Other Applications

- Composition Abstractions
  - VTP: Variability Templates
  - UPP: Universal Extensible Preprocessor
  - Other Abstractions

- Functional Composition Systems
  - STP: Slot Templates
  - SkAT4J: SkAT for Java
  - Other Systems

- Composition Frameworks & Languages
  - SkAT/Minimal, SkAT/Core, SkAT/Full, SkAT/Full
  - JastAddJ, Other Languages

---

JastAdd Reference Attribute Grammars
C3: SkAT composition framework

- **Specific composition system specs**
  - Component model based on RAG
  - Fragment component grammars
  - 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

- **Preprocessor tool**
- **JastAdd RAG compiler**
- **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**

**Implementations:**
- **COMPOST** [Aßmann03]
- **Reusewair** [Henriksson09]
- **Reuseware** [Johannes11]
- **SkAT**

- typed by a metamodel
- textual specification
- implies various requirements on composition systems
C4: Consolidating review – results in brief

**Supported composition features**

- 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)

**Support of the BAF scenario**

**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
  o Invasive Software Composition
  o Problem Analysis
  o Thesis Contributions

Part II: Well-Formed Invasive Software Composition
  o Component Models with Attribute Grammars
  o Composition Strategies
  o Fragment Contracts
  o 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
  o static: ordered attribute grammars (OAGs)
  o 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 = ...
```

- **Input** example:
  - `var: int a + 3.1 DEC`
  - `int 3 INT` (AST node)

- **Result** example:
  - `var: int a + 3.1 DEC`
  - `int 3 INT`

Data flow and reference connections are shown in the diagram.
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])

**SimpAG specification language**
- flat EBNF for context-free grammars
- abstract nonterminals: @
- nonterminal inheritance: ▲
RAG-based component models: composition environments

SimpAG specification language
- flat EBNF for context-free grammars
- abstract nonterminals: @@
- nonterminal inheritance: \[\]

external ClassDecl, VarDecl, Expr

ClassBox \Box Box ::= fragment:ClassDecl
VarDeclBox \Box Box ::= fragment:VarDecl
ExprBox \Box Box ::= fragment:Expr

Composition Environment

BoxList

ClassBox "Core"

VarDeclBox "A"

ExprBox "B"

Stmts "h1"

Expr "[[s1]]"

var

int INT a

- 3 INT

08.10.2015
Well-Formed and Scalable ISC
Slide 48 of 85
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**

- \( \text{syn bool}_{\{n \mid n \in N\}} \cdot \text{isSlot} \)
- \( \text{syn string}_{\{n \mid n \in N\}} \cdot \text{slotName} \)

\[
\begin{align*}
\text{fun } n_i \cdot \text{isSlot} &= \begin{cases} 
\text{true} & \text{if node matches pattern,} \\
\text{false} & \text{else.}
\end{cases} \\
\text{fun } n_i \cdot \text{slotName} &= \begin{cases} 
\text{name from } n_i \text{-context} & \text{if node is a slot,} \\
\bot & \text{else.}
\end{cases}
\end{align*}
\]

**Slot collection**

- \( \text{syn Node*}_{\{n \mid n \in N\}} \cdot \text{slots} \)

\[
\begin{align*}
\text{fun } \{n \mid n \in N\} \cdot \text{slots} &= \begin{cases} 
\{\text{node}\} & \text{if isSlot} = \text{true,} \\
\bigcup_{c \in \text{child}_{\text{all}}} c \cdot \text{slots} & \text{else.}
\end{cases}
\end{align*}
\]
RAG-based component models: compositional points

```kotlin
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 BlockStmts.isHook = true;
fun BlockStmts.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

Part II: Well-Formed Invasive Software Composition
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 $n$**:

$$\begin{align*}
\text{syn } \kappa \{n \mid \prop \in \Ass_x(n)\}.\prop \\
\text{fun } n_i.\prop = \begin{cases} 
\bot & \text{if } \isSlot = \isHook = \false \\
\kappa\text{-expression} & \text{else.}
\end{cases}
\end{align*}$$

**Characteristic attributes**:

$$\begin{align*}
\text{syn } \kappa \{f \mid \prop \in \Ass_x(f)\}.\prop \\
\text{fun } f.\prop = \kappa\text{-expression}
\end{align*}$$

**Fragment assertion**:

$$\begin{align*}
\text{syn } \bool_{\bot} \{n \mid n \in S \cup H\}.\check\text{(Node)} \\
\text{fun } n_i.\check(f) = \begin{cases} 
\bot & \text{if } \isSlot = \isHook = \false, \\
\true & \text{if } \Ass_x(n) \cup \Ass_x(\Lab(f)) = \emptyset, \\
\bool\text{-expr} \text{ on } \Ass_x(n) \cup \Ass_x(\Lab(f)) & \text{if } \Ass_x(n) \cup \Ass_x(\Lab(f)) \neq \emptyset.
\end{cases}
\end{align*}$$
Example: operator-determined composition with contract
Part II: Well-Formed Invasive Software Composition

Example: operator-determined composition with contract
Example: operator-determined composition with contract

- ClassBox "Core"
- VarDeclBox "A"
- ExprBox "B"
- BoxList
- Composer List
- Composition Environment

`'a' not visible at 's1'`
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

- **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

- **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
Part II: Well-Formed Invasive Software Composition

SkAT4J: a well-formed composition system for Java

Diagram showing the architecture of SkAT4J, including components like Syntax, Java FCM, Composition API, SkAT/Full, and JastAddJ, along with the interactions between them.
SkAT4J: used as a library

```java
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();
}
```
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
  o Invasive Software Composition
  o Problem Analysis
  o Thesis Contributions

Part II: Well-Formed Invasive Software Composition
  o Component Models with Attribute Grammars
  o Composition Strategies
  o Fragment Contracts
  o Implementation in SkAT

Conclusion & Outlook
Part III: Conclusion & Outlook

Thesis achievements – summary

Scalable ISC approach

Minimal ISC

Invasive Software Composition
[Aßmann03]

Well-Formed ISC

Unchecked

Syntax checked

e.g., template metaprogramming

Semantics checked

e.g., aspect Weavers, SkAT4J

model and implementation technique

e.g., CPP, VTpL, STpL, UPP
Part III: Conclusion & Outlook

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
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

(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

**Style Definitions**

- **Style:** OpenACC
  - **Type:** parallelization
  - **Fragment Target:** DoConcurrent
    - `$acc loop private(#PRIVATE_VARS#)
      #INNER#
    $acc end do`

**Sequential Program**

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

**Parallel Program**

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

**Style Application Recipe**

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

(figures by Johannes Mey)
End
References


08.10.2015 Well-Formed and Scalable ISC Slide 86 of 85
References


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

**BAF scenario:** generating code from a business domain model

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

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

//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:**

```
BusinessModel
  \n  0..* roleDefinitions
  \n  RoleDefinition
    \n    name : EString
    \n    0..* superRoles
    \n    PropertyDefinition
      \n      name : EString
      \n      0..* properties
      \n      PropertyType
        \n        default : EString
        \n        Date
          \n          targetType : EString
          \n        Percentage
          \n          targetType : EString
          \n        Hours
          \n          targetType : EString
```

**BAF grammar:**

```java
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:

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

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

**BAF scenario:** ideal output

```java
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;
    }
}
```

- **implicit constructor extension**
- **method exit extension**
- **terminal bindings**
- **Employee and Customer mixin**
- **Binding and extension**
### Part I: Thesis Contributions

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

- **FCM specification**
- **Connector/Collaboration specification**
- **Composition program**

**Conceptual view**

- **CM development time**
- **CM usage time**

**Design „is“**

- **REX**
- **.Fracol**
- **.UCL**

**Design „should“**

- **REX**
- **Fracol**
- **UCL**

- **realizes**
- **uses/ref.**

No SoC!!!

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

08.10.2015

Well-Formed and Scalable ISC
Attribute grammars: shortened definition

**Definition (attribute grammar):**
An attribute grammar (AG) is an 8-tuple \( G = (G_0, \text{Syn}, \text{Inh}, \text{Syn}_x, \text{Inh}_x, K, \Omega, \Phi) \) with the following components:

- \( G_0 = (N, \Sigma, P, S) \) a CFG,
- \( \text{Syn} \) and \( \text{Inh} \) the finite, disjoint sets of synthesized and inherited attributes,
- \( \text{Syn}_x : N \rightarrow P(\text{Syn}) \) a function that assigns a set of synthesized attributes to each nonterminal in \( G_0 \),
- \( \text{Inh}_x : N \rightarrow P(\text{Inh}) \) a function that assigns a set of inherited attributes to each nonterminal in \( G_0 \),
- \( K \) a set of types/sorts,
- \( \Omega : \text{Inh} \cup \text{Syn} \rightarrow K \) a function assigning each attribute \( a \in K \),
- \( \Phi \) a set of semantic functions \( \varphi(p,i,a) \) with \( p \in P \), \( i \in \{0, \ldots, n_p\} \), \( a \in \text{Syn}_x(p_i) \cup \text{Inh}_x(p_i) \).
RAG-based component models: compositional points

- Hooks are identified using *inherited* attributes → context-dependent
- Hooks are list nodes
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{verbatim}
syn Node* {n | n \in N}.points
syn Node {n | n \in N}.srcFragment
syn bool↓ {n | n \in N}.isBind
syn bool↓ {n | n \in N}.isExtend
syn bool↓ {n | n \in N}.isExtract

fun {n | n \in N \setminus B}.isBind = false
fun {n | n \in B}.isBind = true
fun {n | n \in E \setminus \varepsilon}.isExtend = false
fun {n | n \in \varepsilon}.isExtend = true
fun {n | n \in N \setminus D}.isExtract = false
fun {n | n \in D}.isExtract = true
\end{verbatim}
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
Part II: Well-Formed Invasive Software Composition

Composition strategies: point-determined composition
Composition strategies: point-determined composition
Part II: Well-Formed Invasive Software Composition

Composition strategies: point-determined composition
Composition strategies: point-determined composition
Composition strategies: point-determined composition

Composition Environment
BoxList
ClassBox "Core"
VarDeclBox "A"
ExprBox "B"
Composer List

Stmts "h1"
Expr "[[s1]]"

isHook true
isSlot true
var
Int
var
Int
Int

true
"B" \rightarrow "s1"
"A" \rightarrow "h1"
srcFrg. points
srcFrg. points

ExprBox "B"

Slots hooks
Slots hooks
Slots hooks
Slots hooks

ExprBox "B"

Bind "B" \rightarrow "s1"
Extend "A" \rightarrow "h1"
Composition strategies: point-determined composition

Composition Environment

BoxList

ClassBox "Core"

VarDeclBox "A"

ExprBox "B"

Stmts "h1"

isHook true

co

ExprBox

Int

var

Int

a

Int

a

Int

3

slots hooks

slots hooks

slots hooks

slots hooks

Bind "B" \rightarrow "s1"

Extend "A" \rightarrow "h1"

Composer List

SrcFrg. points

SrcFrg. points

Slots hooks

Slots hooks

Slots hooks

Slots hooks

Composer List

Slots hooks

Slots hooks

Slots hooks

Slots hooks

Slots hooks
Composition strategies: attribute-determined composition

- different starting points may cause different composition results
- attribute dependencies are not obvious: composition may yield unintended results
Part II: Well-Formed Invasive Software Composition

Composition strategies: attribute-determined composition

**Situation 1**

1b

**Situation 2**

2b

08.10.2015

Well-Formed and Scalable ISC
SkAT screenshot

- generated jar library
- declarative build configuration
- SkAT/Full and SkAT/Core packages
Part II: Well-Formed Invasive Software Composition

Boxology of SkAT4J

abstract JavaFragmentBox:Box ::= ;
// Top-level boxes.
CompilationUnitBox:JavaFragmentBox ::= Fragment:CompilationUnit;
ClassBox:JavaFragmentBox ::= Fragment:ClassDecl;
InterfaceBox:JavaFragmentBox ::= Frag
ImportBox:JavaFragmentBox ::= Fragment
// Member-level boxes.
MethodBox:JavaFragmentBox ::= Fragment
ConstructorBox:JavaFragmentBox ::= Fr
FieldBox:JavaFragmentBox ::= Fragment
MemberBox:JavaFragmentBox ::= Fragment
// Block-level boxes.
StatementBox:JavaFragmentBox ::= Frag
ExpressionBox:JavaFragmentBox ::= Fra
BlockBox:JavaFragmentBox ::= Fragment
abstract JavaNameBox:JavaFragmentBox
DotBox:JavaNameBox ::= Fragment:Dot;
TypeAccessBox:JavaNameBox ::= Fragment
ParTypeAccessBox:JavaNameBox ::= Frag
AccessBox:JavaNameBox ::= Fragment:Access;

abstract JavaTerminalBox:Box ::= ;
StringBox:JavaTerminalBox ::= Fragment:StringValue;
StringValue ::= <Value: String>;
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.addExtendContent(cuName+#Type,def.getType().getTargetType());
                cSys.addBindTerminal(cuName+#SetSfx,"set" + toFirstUpper(def.getName()));
                String stmt = "v += \
 [Field]:\" + [GetSfx]();";
                cSys.addExtendContent(cuName+#AsString.methodExit,stmt);
                cSys.addExtendContent(cuName+#GetSfx,"get" + toFirstUpper(def.getName()));
                cSys.addExtendContent(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

<table>
<thead>
<tr>
<th>Slot name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MethodDecl</td>
<td><code>public class A { void MethodSlot(); }</code></td>
<td>A slot for method decls.</td>
</tr>
<tr>
<td>MemberDeclSlot</td>
<td><code>public class A { [MemberSlot]; }</code></td>
<td>A class-members slot, e.g., fields, methods.</td>
</tr>
<tr>
<td>StmtSlot</td>
<td><code>public void m(){ [StmtSlot]; }</code></td>
<td>A slot for statements, e.g., declarations, blocks.</td>
</tr>
<tr>
<td>ExprSlot</td>
<td><code>Object[] o = new Object[[ExprSlot]];</code></td>
<td>A slot for expressions, e.g., initializations.</td>
</tr>
<tr>
<td>VarAccessSlot</td>
<td><code>int a = this.[VarName];</code></td>
<td>A slot for accessing fields of an object.</td>
</tr>
<tr>
<td>TypeAccessSlot</td>
<td><code>[[TypeSlot]] t = new [[TypeSlot]];</code></td>
<td>A slot for “real” generic types.</td>
</tr>
<tr>
<td>ArrayTypeAccessSlot</td>
<td><code>[[TypeSlot]][] t = new [[TypeSlot]][i];</code></td>
<td>A slot for generic arrays.</td>
</tr>
<tr>
<td>GenericTypeAccessSlot</td>
<td><code>public void m(A&lt;[[ParSlot]&gt; arg){}</code></td>
<td>Convenience slot for type parameters.</td>
</tr>
<tr>
<td>TypeVariableSlot</td>
<td><code>public class A &lt;[[ParSlot]&gt;{}</code></td>
<td>Convenience slot for type parameters.</td>
</tr>
<tr>
<td>TypeDeclNameSlot</td>
<td><code>public interface [[NameSlot]] {}</code></td>
<td>A slot for interface and class-declaration names.</td>
</tr>
<tr>
<td>MethodDeclNameSlot</td>
<td><code>public void [[NameSlot]]{}{}</code></td>
<td>A slot supporting generic method names.</td>
</tr>
<tr>
<td>VarDeclNameSlot</td>
<td><code>private String [[Name]];</code></td>
<td>Slots supporting generic field and variable names.</td>
</tr>
<tr>
<td>MethodAccessNameSlot</td>
<td><code>String a = [[NameSlot]]();</code></td>
<td>A slot for generic method calls.</td>
</tr>
<tr>
<td>ConstructorDeclNameSlot</td>
<td><code>public [[NameSlot]]{}{}</code></td>
<td>A slot supporting generic constructor names.</td>
</tr>
<tr>
<td>StringValueSlot</td>
<td><code>String s = &quot;SkAT: [[Msg]] [[Msg]].&quot;</code></td>
<td>A slot in string literals.</td>
</tr>
<tr>
<td>ParameterDeclNameSlot</td>
<td><code>public void m(String [[Name]]){}</code></td>
<td>A slot for method-parameter names.</td>
</tr>
</tbody>
</table>
SkAT4J: excerpts from Slots.jrag

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

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(); }

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

SkAT/Core

Java FCM

08.10.2015

Well-Formed and Scalable ISC

Slide 112 of 85
### SkAT4J: supported hooks

<table>
<thead>
<tr>
<th>Hook name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block.stmts</td>
<td>`&lt;scope&gt;.statements</td>
<td>Hooks to extend the statement list of any block.</td>
</tr>
<tr>
<td></td>
<td>.statementsEnd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.methodEntry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.methodExit</td>
<td></td>
</tr>
<tr>
<td>CompilationUnit.importDecIs</td>
<td>`&lt;scope&gt;.imports</td>
<td>A hook to extend the list of import declarations.</td>
</tr>
<tr>
<td>TypeDecl.bodyDecIs</td>
<td>`&lt;scope&gt;.members</td>
<td>Hooks to extend the members of a class or interface declaration.</td>
</tr>
<tr>
<td></td>
<td>.membersEntry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.membersExit</td>
<td></td>
</tr>
<tr>
<td>ClassDecl.implements</td>
<td>`&lt;scope&gt;.implements</td>
<td>A hook to add new interfaces to a class.</td>
</tr>
<tr>
<td>MethodDecl.parameters</td>
<td>`&lt;scope&gt;.parameters</td>
<td>A hook to add new parameters to a method.</td>
</tr>
</tbody>
</table>
SkAT4J: excerpts from Hooks.jrag

```java
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;

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();
}
```
### SkAT4J: supported fragment assertions

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn TypeDecl.assertNotDeclared(MethodDecl)</td>
<td>Checks method redeclarations</td>
</tr>
<tr>
<td>*syn MethodDecl.signature()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.localMethodsSignatureMap()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.assertNotDeclared(FieldDeclaration)</td>
<td>Checks field redeclarations</td>
</tr>
<tr>
<td>*syn TypeDecl.localFieldsMap()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.assertVariablesProvided(ASTNode)</td>
<td>Checks variables in scope</td>
</tr>
<tr>
<td>*syn ASTNode.danglingVars()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.memberFields(String)</td>
<td></td>
</tr>
<tr>
<td>inh TypeDecl.lookupVariable(String)</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.assertMethodsProvided(ASTNode)</td>
<td>Checks methods in scope</td>
</tr>
<tr>
<td>*syn MethodDecl.danglingCalls()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.memberMethods(String)</td>
<td></td>
</tr>
<tr>
<td>inh TypeDecl.lookupMethod(String)</td>
<td></td>
</tr>
<tr>
<td>syn ExprSlot.assertCompatibleType(Expr expr)</td>
<td>Checks expression types</td>
</tr>
<tr>
<td>*syn Expr.type()</td>
<td></td>
</tr>
<tr>
<td>syn TypeDecl.wideningConversionTo(TypeDecl)</td>
<td></td>
</tr>
</tbody>
</table>
SkAT4J: excerpts from Assertions.jrag

```java
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
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;
}
```
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 ∈ required)
- **Fragment selection/conditional composition**: select fragment components fitting to a certain condition or assertion
Skeletons: basic forms

→ data-parallel skeletons

(e.g. Map)

→ task-parallel skeletons ("multiple algorithms multiple nodes")

(e.g. Pipe)
Well-Formed and Scalable ISC

Part II: Well-Formed Invasive Software Composition

SkAT4J: a skeleton library
SkAT4J: a skeleton library

Composed result:

Deployed application (Wikipedia document indexing)
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])
Composing classes: rep. well-formedness check; deep tree

(experiments by [Tasic14])
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].

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

1. Add language model to FCM
   - [Support complete language model?]
     - Yes
       - [Grammar or metamodel of the fragment language available?]
         - Yes
           - (1) Add language model to FCM
         - No
           - (5) Add semantic model to FCM
   - No
     - (4) Add constructs of interest (islands)

2. Create complete language model
   - [Support well-formed ISC?]
     - Yes
       - (6) Add semantic model to FCM
     - No
       - (3) Create language model based on minimal FCM

3. Create language model based on minimal FCM
   - [Minimal language model available?]
     - Yes
       - (2) Create complete language model
     - No
       - (4) Add constructs of interest (islands)

4. Add constructs of interest (islands)
   - [Support semantics?]
     - Yes
       - (7) Create partial semantic model
     - No
       - (3) Create language model based on minimal FCM

5. Add semantic model to FCM
   - [Semantic model of fragment language available?]
     - Yes
       - (6) Create complete semantic model
     - No
       - (7) Create partial semantic model

6. Create complete semantic model
   - [Support full semantics?]
     - Yes
       - (8) Refine component model
     - No
       - (7) Create partial semantic model

7. Create partial semantic model
   - [Partial semantic model available?]
     - Yes
       - (8) Refine component model
     - No
       - (4) Add constructs of interest (islands)

8. Refine component model
   - [Support semantics?]
     - Yes
       - (8) Refine component model
     - No
       - (7) Create partial semantic model

---

Language model
Grammar or metamodel of the fragment language
Yes

(1) Add language model to FCM
[Grammar or metamodel of the fragment language available?]
Yes

(2) Create complete language model
Language model
[Support full semantics?]
Yes

(5) Add semantic model to FCM
Semantic model
[Support well-formed ISC?]
Yes

(6) Create complete semantic model
Semantic model
[Support full semantics?]
Yes

(7) Create partial semantic model
Partial semantic model
A partial static semantics model of the fragment language
No

(8) Refine component model
Minimal language model
[Support well-formed ISC?]
No

---

Island FCM ➔ syntax-aware FCM
Token/String-based FCM ➔ island FCM
Island FCM ➔ island FCM

---

Syntax-aware FCM ➔ semantics-aware FCM
Semantics-aware FCM ➔ semantics-aware FCM

---

08.10.2015
Well-Formed and Scalable ISC
Slide 127 of 85
Scalable Invasive Software Composition

BPMN workflow of complex refinement activity

Refine component model

- (1) Specify extension grammar/metamodel
- (2a) Specify slot model
- (2b) Specify hook model
- (2c) Specify rudiment model
- (2d) Specify composer model
- Glue model: Spec gluing language model and composition model
- Slot model: Extension modeling new slots
- Hook model: Extension modeling new hooks
- Rudiment model: Extension modeling new rudiments
- Composer model: Extension modeling new (embedded) composers

Legend:
- XOR gateway/join
- OR gateway/join
- Sequence flow between tasks

[Language extensions required?]
[FCM extension model: Grammar/metamodel of compositional constructs and/or fragment boxes]

[Glue required?]
[Contradictions in FCM?]
[Fragment contracts required?]
[Reorganization desirable?]

(3) Specify glue model
(4) Adapt existing FCM specs
(5) Specify fragment contracts
(6) Conduct global refactoring

08.10.2015
Well-Formed and Scalable ISC
Slide 128 of 85
Scalable Invasive Software Composition

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 ::= 

fun Slot.isSlot = true
fun Slot.slotName = name
fun \{n \mid n \in N - S\}.isSlot = false
fun \{n \mid n \in N - S\}.slotName = ⊥

fun \{n \in N\}.child_{a11}.isHook = false
fun \{n \in N\}.child_{a11}.hookName = ⊥
Minimal ISC code generation

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

Composition Environment

BoxList

GenericFragment "Tpl1"

GenericFragment "Tpl2"

GenericFragment "Arg"

Composer List

Bind "TPL1" → "S3"

Bind "Arg" → "S1"

Bind "Arg" → "S2"

A [[S1]][[S2]]

B [[S3]] CD

X

BAXXCD

08.10.2015

Well-Formed and Scalable ISC
UPP: fragment component model + embedded composers

**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

```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
}
#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) {
        log("SettingX");
        this.x = x;
    }
    ...
    public float getZ() {
        return z;
    }
    public void setZ(float z) {
        log("SettingZ");
        this.z = z;
    }
}
UPP example: 2D and 3D vectors in Java

```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
}
#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.
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
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
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*