Interconnection Styles

Software Design Following the Export (Server) Style

Software Design Following the Export (Client) Style
Software Design Following the Export Client / Export Style

Software Design Following the Import Style

Software Design Following the Import / Export Style
Interconnection Style Formalism

- An ISF specification consists of a finite set of logic rules. There are two kinds of ISF rules:
  - **Permission**: Determine the syntactically legal relations between the components of a software design.
  - **Definition**: Define new relations based on patterns of entities and relations.

Interconnection Style Formalism (Cont’d)

- Each rule consists of a finite set of entities and relations.
- Entities represent design components and relations represent interconnections between these components.
Reverse Engineering (Interconnection Styles)

**Design Example**

- M2
- M3
- M4
- M5
- M6
- SS1
- SS2
- SS2.1

**Design Example in Datalog**

- % Entity Types
  - subsystem("SS1").
  - subsystem("SS2").
  - subsystem("SS3").
  - subsystem("SS4").
  - subsystem("SS5").
  - subsystem("SS6").

- % Constraint Relations
  - contain("SS1", "M1").
  - contain("SS2", "M2").
  - contain("SS3", "M3").
  - contain("SS4", "M4").
  - contain("SS5", "M5").
  - contain("SS6", "M6").

- % Module Dependencies
  - use("M2", "M1").
  - use("M3", "M2").
  - use("M4", "M3").
  - use("M5", "M4").

© SERG
Formal Definition of ISF

- The syntax of ISF is described by presenting the visual symbols of the notation.
- These symbols represent design:
  - entities
  - relations
  - rules
- The semantics of ISF is described using Datalog.

Formal Definition of ISF

- The semantics of ISF is presented bottom-up:
  - semantics of ISF entities and relations
  - semantics of ISF rules in terms of the semantics of entities and relations

ISF Entities

- ISF entities are depicted as labeled rectangles.
- Each ISF entity represents a set of typed design components such as:
  - modules, classes, subsystems, etc.
- There are three types of entities:
  - [Label]
Example of an ISF Entity

• An example of an ISF entity is:

![Subsystem]

• **Formally:** subsystem(SS)
  - where SS is a free variable.

• **Informally:** “The set of all subsystems in a software design.”

---

ISF Relations

• ISF relations are depicted as:
  - labeled edges
  - nested rectangles

• Each ISF relation represents a relation of a software design such as:
  - containment (depicted as nested rectangles)
  - import
  - export
  - inherit, etc.

---

ISF Containment Relations

![Containment Diagram]

1. Entity contains simple entity
2. Entity contains transitive entity
3. Entity contains permission entity
**ISF Edge Relations**

1. Simple Relation
2. Negated Simple Relation
3. Reflexive Transitive Relation
4. Negated Reflexive Transitive Relation
5. Permission or Definition Relation

**Examples of Edge Relations**

Example edge relations include:
- use(S1, S2), subsystem(S1), subsystem(S2)
- not(use(S1, S2)), subsystem(S1), subsystem(S2)
- rtc_use(S1, S2), subsystem(S1), subsystem(S2)
- not(rtc_use(S1, S2)), subsystem(S1), subsystem(S2)

**ISF Rules: The Export Style (Part 1)**

The Export Style rules are:
- wf_export(PS, SS) :- ss(PS), ss(SS), contain(PS, SS).
- ill_formed(X, Y) :- export(X, Y).
- not(wf_export(X, Y)).
- wf_contain(PS, SS) :- ss(PS), ss(SS).
- ill_formed(X, Y) :- contain(X, Y).
- not(wf_contain(X, Y)).
- well_formed_design() :- not(ill_formed()).
**ISF Rules:**

The Export Style (Part 2)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>com_use(SS1,SS2)</code> := no(SS1), no(SS2), <code>com_use(SS1,SS2)</code>.</td>
<td>No communication between SS1 and SS2.</td>
</tr>
<tr>
<td><code>com_use(SS1,SS2)</code> := <code>com_use(SS1,SS2)</code>, no(SS1), no(SS2).</td>
<td>Communication exists between SS1 and SS2 if already exists.</td>
</tr>
<tr>
<td><code>com_use(SS1,SS2)</code> := no(SS1), <code>com_use(SS1,SS2)</code>, no(SS2).</td>
<td>Communication exists between SS1 and SS2 if already exists.</td>
</tr>
</tbody>
</table>

Reverse Engineering (Interconnection Styles) © SERG

---

**Export Client / Export Style**

Reverse Engineering (Interconnection Styles) © SERG

---

**Import / Export Style (Part 1)**

Reverse Engineering (Interconnection Styles) © SERG

---
Import / Export Style (Part 2)

Tube Style (Part 1)

Tube Style (Part 2)
On the Automatic Recovery of Style-Specific Structural Dependencies

Introduction

• Motivation
• Definitions
  – MDG
  – Clustering
  – Interconnection Styles
  – ISF
• Examples
• Tool

Motivation

• Documentation problems:
  – Missing or inaccurate design documentation
  – Poorly documented code
  – Original programmers not available for consultation
• Program understanding. Need for high-level information
  – Existing clustering techniques do not produce high-level dependencies.
• Design validation
**Module Dependency Graph**

- Nodes represent modules.
- Edges represent dependencies between modules (i.e., method invocation, variable reference, etc).
- Obtained automatically with Cia, Acacia, Chava.

**Example: Mini Tunis MDG**

- Clustering
  - Aggregate representation of MDG, that groups together modules that are tightly coupled.
  - Clusters represent subsystems.
  - Obtained automatically with Bunch.
Example: Mini Tunis after clustering

Structure Graph

- A clustered system may be visualized as a tree
  - Edges represent containment, use and style dependencies
  - Nodes represent modules and subsystems
- Easier to visualize large systems

Interconnection Styles

- Regulate the interactions between modules and subsystems
- High-level relations not present in source code or clustered system.
- Can be described formally with the Interconnection Style Formalism (ISF)
Example: Export Style

• Export style facilitates the specification of subsystem interfaces.
• If a subsystem S exports a module M, M belongs to S’s interface.
• Modules outside of S can use modules belonging to S’s interface.

Example: Export Style

• M3 can use M4 because it is exported by SS21 and SS21 is exported by SS2.
• M6 cannot use M5 because it is not exported by SS21. If M6 did use M5 there would be a stylistic violation.

ISF

• Visual notation that allows specification of constraints on configurations of components and dependencies
• Rules are represented as directed labeled graphs
  – Nodes: modules and subsystems
  – Arrows: dependencies and containment relations
**ISF (cont’d)**

- Types of rules:
  - PERMISSION: define the set of well-formed configurations
  - DEFINITION: define new relations based on patterns of software components and relations.

**ISF Specification of Export Style**

Example: Mini Tunis after Edge Repair
**Tool**

- **Style Editor**
  - Enables the user to define custom interconnection styles visually
- **Edge Repair Utility**
  - Takes structure graph and style and finds the missing structural dependencies automatically
  - Checks if a design satisfies stylistic constraints

**Integration of Reverse Engineering Tools**

- **Source Code Analysis Tool**
  - Module Dependency Graph
- **Clustering Tool (Bunch)**
- **Well-formed Design**
- **Style Definition**
- **Edge Repair Utility**
- **Clustered MDG**

**Techniques**

- Edge Repair
- Rule Checking
- Code Generation
**Edge Repair**

- Given
  - Structure Graph
- Style definition

---

**Goal:** find the missing style dependencies such that:
- Visibility between modules is minimized
- No dependency (i.e., use or style) violates any stylistic constraint.

---

**Algorithms**

- Huge search space (i.e. cannot try all possible combinations)
- Optimization methods:
  - **Hill climbing:** start with a random configuration and find better ones by making small changes
  - **Edge removal:** start with “full” configuration and remove edges until no better solution can be found
  - **Genetic algorithm**
**Fitness Function**

- Measures the quality of a configuration
  - Good:
    - Configurations with a large number of well-formed *use* relations
  - Bad:
    - Configurations with a large number of ill-formed *style* dependencies
    - Configurations with high visibility (many *see* relations)

**Simplistic visibility**

- Computing visibility is an $O(N^2)$ operation
- Frequent operation
- Approach: simplistic visibility
  - Approximation to standard visibility
  - Given by the number of *style* dependencies
  - $O(1)$ operation
- Drawback: does not work for some styles
  - Non-recursive (permission rules are not defined in terms of other permission rules)
  - Modules are exposed by direct (i.e., non-transitive) relations with their ancestors.

**Fitness Function**

$$
\text{quality}(C) = \begin{cases} 
\frac{wfs}{ifs +ifu} & \text{ifs } \neq 0 \text{ or } ifu \neq 0 \\
\frac{1}{wfs} & \text{ifs } = 0, ifu = 0, wfs \neq 0 \\
\text{MaxS} + 2 & \text{ifs } = 0, ifu = 0, wfs = 0 
\end{cases}
$$

- *ifu*: ill-formed *use* dependencies
- *ifs*: ill-formed *style* dependencies
- *wfs*: well-formed *use* dependencies
- *wfs*: well-formed *style* dependencies
- MaxS: maximum number of *style* dependencies in C
**Hill Climbing**

- Start from random configuration of dependencies
- Select an edge, to add or remove, such that the new configuration is better than the current one
- Repeat until no better solution can be found

**Edge Removal**

- Start from full configuration
- Remove edges that makes the current solution better until no better solution can be found

**Hill Climbing vs. Edge Removal**

- **Hill Climbing**
  - Adds relations and then removes them (might add unnecessary relations)

- **Edge Removal**
  - Always starts with a full configuration (even if only a few relations are required)
  - Fewer neighbors to consider as the algorithm advances
### Performance:

**Hill Climbing vs. Edge Removal**

<table>
<thead>
<tr>
<th></th>
<th>Hill Climbing</th>
<th>Edge Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler (19 nodes)</td>
<td>0.19 s</td>
<td>0.14 s</td>
</tr>
<tr>
<td>Mimi Tunis (32 nodes)</td>
<td>0.8 s</td>
<td>0.46 s</td>
</tr>
<tr>
<td>Grappa (117 nodes)</td>
<td>8.5 s</td>
<td>3.67 s</td>
</tr>
<tr>
<td>Incl (212 nodes)</td>
<td>25.72 s</td>
<td>9.2 s</td>
</tr>
</tbody>
</table>

### Genetic Algorithm

- **Encoding**
- **Crossover**
- **Fitness function**
- **Mutation**

### Optimizations

- **Pessimistic edge reparability:** avoid processing relations that can *never* exist (e.g., export relations between siblings in the Export Style)
- **Simplistic visibility measurement:** based on the number of style dependencies in a configuration (reduces $O(N^2)$ operation to $O(1)$).
**Rule Checking**

- Goal: check well-formedness of a dependency with respect to a permission rule.
- Can be reduced to a pattern matching problem $\Rightarrow$ the rule is matched to the structure graph.

![Rule Checking Diagram]

**Pattern Matching**

- Invalid pattern matches:

![Pattern Matching Diagram]

**Pattern Matching Algorithm**

- Based on Depth-First Search
  - State:
    - Set of fixed nodes
    - Set of free nodes
  - Goal: find a state with no free nodes
  - Initial state: the set of fixed nodes contains the endpoints of the relation being checked
Pattern Matching Example

Choosing the Node to Fix

• If the node is not connected to a fixed node, it can assume any value (i.e., N possibilities)
• Choose a node connected to a fixed node to constrain the possible values for the node

Pessimistic Edge Reparability Algorithm

• Algorithm: based on the pattern matching
  – Full configuration with all dependencies as unknown
  – Repeat until no more unknown dependencies exist
    • Pick a dependency tagged as unknown
    • Check if it is reparable (i.e., perform a pattern match of every related permission rule). If an unknown dependency is found during the match, check its reparability (i.e., recursive call of this algorithm)
    • If a match is found and all involved dependencies are reparable, tag the relation as reparable. If not, tag it as irreparable.
**Code Generation**

- Java code is generated to check each rule
  - Take advantage of Java’s compile-time and run-time optimizations
  - Some decision making is performed when the rule is translated instead of when it is checked

---

**Code Optimization**

- Take advantage of the fact that
  - Rules are static
  - Search tree depth is fixed and known for each rule (depends on the number of nodes in the rule).
- … to determine the order in which rule’s nodes are fixed at translation time.

---

**Code Optimization (cont’d)**

- When rules *look* like trees (most times), topological sorting can give a good ordering of nodes.
Conclusion

• Tool
  – find missing high-level dependencies in a system
  – validate a design against a style specification

• Further research:
  – Optimization of edge repair algorithm (e.g., using incremental algorithms)
    • Replace the “simplistic visibility” measurement
  – Repairing designs by reclustering.