Dynamic Specification, Abstraction Hierarchies, and Code Generation in Automated Software Reuse

Jonathan Phillips
2/13/04
Outline

• **Software Reuse Overview**
  • Problem Statement
  • Solution Overview
    – Specification
    – Abstraction
    – Retrieval/Adaptation
• Contributions
• Future Work
Software Development


• Why is this bad?
  – Reinventing the wheel: programmers spend time in repetitive tasks
  – Lack of simple way of finding relevant components
  – Error prone, leads to reliability and safety concerns
  – Informal
Software Reuse

• Definition of software reuse: using existing software artifacts during the construction of a new software system. (Krueger, C.W.(1992). “Software Reuse”)

• Why is this good?
  – Cost effective, need not pay the price of redeveloping software
  – Potential for increased reliability
  – Can make programmers more efficient, shifts focus from mundane to meaningful
  – Encourages the industrialization of the software industry: common tools, components, processes
Examples of Reuse

- Component libraries
- Software schemas
- Design patterns
- Very high-level languages
- Graphical programming environments
  - Khoros (Cantata), Modelica
Automated Reuse

• Addresses some shortcomings in software reuse
  – Number of components can be prohibitive
  – Understanding correct component usage is difficult
  – Adaptation and integration

• How does automation help?
  – Raises the level of abstraction a programmer deals with.
Examples of Automated Reuse

• **Transformational Systems**
  – Reuse of design knowledge in specs and transformations (Divide-and-conquer, global search)

• **Application Generators**
  – Reuse domain specific abstractions and code

• **Code Generators**
  – Reuse schemas and templates
Difficulties

- Abstraction definition
- Domain limitations
- Uniformity in abstraction representation
- Applying abstractions
Existing Work

• Transformational Systems - Generate executable code from specifications using a series of transformations on the spec while maintaining semantics.

• Program Synthesis – Generate code from formal specifications using mathematical theorem proving. Code is extracted automatically from a constructive proof of the specification

• Design Patterns – present successful solutions to common software problems
  – Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns, Elements of Reusable Object-Oriented Software.
Outline

• Software Reuse Overview
• **Problem Statement**
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• Future Work
Problem Formulation

Given a user-defined specification and a library of source-code components, automatically retrieve, adapt, and integrate the components necessary to satisfy the specification.
Problem Formulation

Let

\[ S = \langle I, \text{Ops}_S, \text{Ord}_S \rangle \]

\( I \) be the set of inputs in the specification \((i_1, i_2, \ldots)\)

\( \text{Ops}_S \) be the set of operators in the specification \((\text{op}_1, \text{op}_2, \ldots)\)

\( \text{Ord}_S \) be the set of orderings on \( \text{Ops}_S \) \((\text{ord}_1, \text{ord}_2, \ldots)\) where

\[ \text{ord}_i = \langle \text{op}_j < \text{op}_k \rangle \]

\( \text{Real}_{\text{op}_i} \) be the realization of \( \text{op}_i \), \( \text{Real}_{\text{op}_i} = \langle \text{Ord}_{R_i}, C_{R_i} \rangle \)

\( C_{R_i} \) be a set of components

\( \text{Ord}_{R_i} \) be an ordering on \( C_{R_i} \)

Find a set \( R_S \) of realizations with respect to \( S \) and a set of orderings \( \text{Ord}_{R_S} \) such that

Each realization \( \text{Real}_{\text{op}_i} \) in \( R_S \) is semantically equivalent to the corresponding operator in \( S \), \( \text{op}_i \):

\[ \text{Real}_{\text{op}_i} \equiv_S \text{op}_i \]

\[ \text{Ord}_{R_S} \equiv_S \text{Ord}_S \]
Problem Formulation cont.

Component Adaptation

Let

$$T_{c_i}$$ be the set of tags in component $$c_i$$

Instantiate the tags for each component in the specification realization using I and information inherent in S.
Outline

• Software Reuse Overview
• Problem Statement
• **Solution Overview**
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• Future Work
Solution Overview

- Represent specification operators using abstract operators
- Abstract operators as schemas
- Formal language syntax and semantics for specifications
- Abstraction inference
- Language (Java, Python,...) specific rules for abstraction inference and code generation
- Abstraction hierarchy (includes components)
- Code generation technology in components
Outline

• Software Reuse Overview
• Problem Statement
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• Future Work
Specification Methodology

• Specifications consist of:
  – Variable declarations
  – Abstract operators

• Specifications used to generate abstract syntax trees
Example Specification: Sorting

```
program integer_sort
    define
        input list a of int
    end define

    sort
        collection_name: a
        comparison_field:
        comparator: '<'
        algorithm: heapsort
    end sort

    store
        data: a
        destination: "console"
        format:"
        forall i in a
    end store
end integer_sort
```
Abstract Operator Definitions

• Schema based
• Basic syntax:
  
  \[ \text{OperatorName} \]
  
  \[ Field1: field1Value \]
  
  \[ \ldots \]
  
  \[ Fieldn: fieldnValue \]

• Only certain types of field values are allowed:
  
  Basic types: int, float, double, String, char, etc.
  
  Var: holds a variable defined in the specification
  
  IOFormat: used in specifying the format of input/output data
  
  Typedef: defines a set of allowable string values for a field
  
  Type: the type required for an input or output to or from the operator
  
  Data type the fields or operator of a data type can be stored in these fields. Data type isn’t the actual field name rather, it would be Field or Operator (example on the next slide)

• Operator definitions are stored in separate files
Example Operator Definition

```plaintext
#SortOperator.def

absdef sort {
    Typedef sort_alg = {"insertionSort", "quickSort", "heapSort", "randomQuickSort"} 
    Var collection_name
    Field comparisonField
    char comparator
    String comparator
    sort_alg algorithm = "insertionSort"
}
```

Type definition to show the types of sorting algorithms allowed.

Var, means that any sort of variable can be placed in this slot.

Field for the type defined above, can only hold one of the given values.

Default value

Var: means that any sort of variable can be placed in this slot.

Basic types
Methodological Motivation

• Abstract operators define a specification language
• Language is dynamic according to user needs
• Operators can abstract large number of concrete operations
• Specification files make compilation of abstraction straightforward
Specification “Compilation”

- Lexical tokens are static
- Parsing changes based on abstract operator definitions
- Semantic analysis occurs as normal
- Compilation results in abstract syntax tree
- Errors can be generated after semantic analysis
Example AST

Program

DECL

DECLNULL

SortOp

Type: int[]    Level: 0

collName: a    comp_f:

comp: <        sort: heap

StoreOp

Type: int[]    Level: 0

data: a        dest: console

outputForm: "\v "

forall i in a
Ideas For Additional Language Features

• Allow for primitive code
  – Own type of abstract operator
  – Requires more complex compiler

• Inter-abstraction abstractions
  – More robust programs
  – Closer to real applications
AST to APG

• Compilation ends with an abstract syntax tree (AST)
• Strip off unnecessary nodes to create the abstract program graph (APG)
• APG is used during concretization
Outline

• Software Reuse Overview
• Problem Statement
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• Future Work
Abstraction System

• Goal of abstraction system:
  – Generate a concrete program graph using the abstract program graph

• Hierarchies of abstractions

• Logic resides in abstractions for choosing implementations (either abstract or concrete)

• Uses methodology similar to Hierarchical Task Networks (HTN)
Basic Abstraction System Algorithm

1. Generate the APG from the AST
2. Find implicit and necessary abstractions in the initial APG to create the working APG
3. Use the working APG to decompose abstractions until only concrete operators remain
1. Generating the APG

```python
GenerateAPG(AST)
currentNode = AST.root
while currentNode.leftChild != AbstractOp
    currentNode = currentNode.rightChild

APGroot = currentNode.leftChild
tempNode = currentNode
nextNode = APGroot
while tempNode != null
    if (tempNode.leftChild == AbstractOp)
        nextNode.addNode(tempNode.leftChild)
        nextNode = nextNode.child
    else
        nextNode.addGraph(GenerateAPG(tempNode.leftChild))
        nextNode = nextNode.child
    tempNode = tempNode.rightChild
```
**Initial APG**

<table>
<thead>
<tr>
<th>SortOp</th>
<th>StoreOp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong>: int[]</td>
<td><strong>Type</strong>: int[]</td>
</tr>
<tr>
<td><strong>Level</strong>: 0</td>
<td><strong>Level</strong>: 0</td>
</tr>
<tr>
<td><strong>collName</strong>: a</td>
<td><strong>data</strong>: a</td>
</tr>
<tr>
<td><strong>comp_f</strong>:</td>
<td><strong>dest</strong>: console</td>
</tr>
<tr>
<td><strong>comp</strong>: &lt;</td>
<td><strong>outputForm</strong>:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>next</strong>:</td>
<td><strong>next</strong>: null</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
forall i in a
```

```plaintext
\v
```

```plaintext
forall i in a
```
2. Finding Implicit Abstractions

APGinitialize(APG)
done <- false
disconnects <- true
handlers <- true

while !done

    variableDisconnects <- checkVariableFlow(APG.root)
    if variableDisconnects isempty
        disconnects <- true
    else
        disconnects <- false
        foreach v in variableDisconnects
            APG.resolveDisconnect(v)

    requiredErrorHandlers <- checkErrorHandling(APG)
    if requiredErrorHandlers isempty
        handlers <- true
    else
        handlers <- false
        foreach err in requiredErrorHandlers
            APG.addErrorHandler(err)

    done <- disconnects && handlers
Disconnects and Error Handlers

• A Disconnect is use of an uninitialized variable
  – Resolved by adding initializing abstractions
  – Symbols maintain an initialized flag
  – Abstractions keep track of variables used, initialized, and/or updated

• Error handlers
  – Resolved using IO “wrappers” to catch errors
  – Currently envisioned for IO operations
  – Eventually available for all operations
Language Rules

• First of concern in garnering information from the command line

• Database of rules necessary for implementing specific language constructs
  – Here, knowing the format of command line input inside a program
  – Data type conversions
  – Abstraction (and code retrieval) vs. simple commands
Updated APG

CommandLineInput
- desiredInfo: a
- next: 

SortOp
- Type: int[]
- Level: 0
- collName: a
- comp_f:
- comp: <
- sort: heap
- next: 

StoreOp
- Type: int[]
- Level: 0
- data: a
- dest: console
- outputForm: "\v "
- forall i in a
- next: null
3. Concretizing the APG

```plaintext
APGDeabstraction(APG)
concreteProgramGraph <- new CPG
nodeStack <- new Stack
concretizationStack <- new Stack
nodeStack.push(APG.root)
while not nodeStack.empty
    currentNode <- nodeStack.pop()
    concretizationStack.push(currentNode)
    while not concretizationStack.empty
        nextToDecompose <- concretizationStack.pop()
        newOperators <- decomposeNode(nextToDecompose)
        if newOperators == null
            concreteProgramGraph.add(nextDecomposition)
        else
            foreach op in newOperators
                concretizationStack.push(op)
        nodesToAdd <- getChildren(currentNode)
        foreach node in nodesToAdd
            nodeStack.push(node)
```
Abstract Operator
Decomposition

• Decomposition information stored in .decomp files. These files contain
  – Names of potential decompositions
  – Task network associated with the decomposition
  – Logic for deciding among potential decompositions
Example Decompositions

decomposition commandLineInput
  import commandLineInput.def
  decomp plainDecl
  decomp convertedDecl

define plainDecl
  concreteOp mainDecl
  abstractOp decl

ordering mainDecl -> Decl

define convertDecl
  concreteOp mainDecl
  abstractOp decl
  abstractOp convert

ordering mainDecl -> decl -> convert

decision_logic
  if desired_var.type == String || desired_var.type == String[]
    plainDecl
  else if desired_var.type != String && desiredVar.type != String[]
    convertDecl
  else
    ERROR

decomposition Decl
  import Decl.def
  decomp declNull
  decomp declInit
  decomp declNew

define declNull
  concreteOp simpleVar

define declInit
  abstractOp initializeVar

define declNew
  abstractOp newVar

decision_logic
  if parent != CommandLineInput
    simpleVar
  else if parent == CommandLineInput && (var.type == String || var.type == String[])
    initializeVar
  else if parent == CommandLineInput && var.type != String && var.type != String[]
    newVar
  else
    ERROR
Deabstraction Hierarchy for Integer Sort Example

- CommandLineInput
  - mainDecl
    - newVar
      - newArrayVariable
        - convertArrayVariable
  - Decl
    - Convert
  - Sort
    - sortPrimitive
      - heapSortPrimitive
        - storeArrayPrimitiveOnConsole
  - Store
Abstract vs. Concrete Operators

• Abstract operators can be decomposed
  – Into one or more operators
  – Multiple operators have explicit order

• Concrete operators reference actual code
  – Source code component
  – Source code instruction or fragment
    • Control flow
    • Declarations
Outline

• Software Reuse Overview
• Problem Statement
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• Future Work
Retrieval System

• Concrete Operators:
  – Contain simple line(s) of code with tags for adaptation
  – Contain the location of source code component for retrieval

• Simple due to the nature of the abstraction hierarchy
Concrete Operator Definitions

• Concrete Declaration
  – source_location, flags, defaults, flagValues, outputString, currentLocation

• Operations
  – Source_location, flags, defaults, flagValues, currentLocation, mainCall, wrapper, functionCall, availableOperators, decideOp
concretedef heapSortPrimitive extends sortPrimitive {
    source_location = ""
    flags = {"<%= localFunctionName %>",
            "<%= mainArg %>",
            "<%= sortArgType %>",
            "<%= sortArg %>",
            "<%= libraryFunctionName %>",
            "<%= leftOperand %>",
            "<%= rightOperand%>",
            "<%= comparator %>",
            "<%= booleanComparison %>"
    }
    defaults = {"", "", "", "x", ""}
    flagValues = new String[flags.length]
    currentLocation = ""
    mainCall = "<%= localFunctionName %>(<%= mainArg %>);
    wrapper = "private static void <%= localFunctionName %>(<%= sortArgType %> <%= sortArg %>)
               {
               <%= functionCall %>
    }
    functionCall = "Sort.<%= libraryFunctionName %>(<%= sortArg %>)
    availableOperators = {"heapSort"}
    void decideOp()
    {
        int flagIndex = flags.indexOf("<%= libraryFunctionName %>");
        flagValues[flagIndex] = availableOperators[0];
    }
}
Concrete Declaration Example

```java
concreteDecl mainDecl {
    source_location = ""
    flags = {"<%= className %>",
             "<%= classMethodsStart %>",
             "<%= nextMethod %>",
             "<%= classMethodsEnd %>",
             "<%= mainFunctionDeclaration %>",
             "<%= mainDefsStart %>",
             "<%= nextDefinition %>",
             "<%= mainDefsEnd %>",
             "<%= mainOpsStart %>",
             "<%= nextOperation %>",
             "<%= mainOpsEnd %>"
    }
    defaults = {"", ",", ",", ",", ",", ",", ",", "public static main(String args[])", ",", ",", ",", ",", ",", ","}
    flagValues = new String[flags.length]
    outputString = populateSkeleton()
    currentLocation = ""
}
```
public class <%= className %>
{
    <%= classMethodsStart %>
    <%= nextMethod %> // This is the flag used to add another operation
    <%= classMethodsEnd %>

    /*
     * The main function replaces the next flag "mainFunctionDeclaration".
     */
    <%= mainFunctionDeclaration %>
    {
        <%= mainDefsStart %>
        <%= nextDefinition %> // Where the next def is inserted
        <%= mainDefsStart %>
        <%= mainOpsStart %>
        <%= nextOperation %> // Where the next operation is inserted
        <%= mainOpsEnd %>
    }
}
private static void heapify(<%= listType %> list[],
    int index, int length)
{
    int left_i = index * 2 + 1, right_i = index * 2 + 2;
    int max;
    if (left_i < list.length && left_i < length &&
        <%= leftOperand %><%= comparator %><%= rightOperand %>
        <%= booleanComparison %>)
        max = left_i;
    else
        max = index;
    if (right_i < list.length && right_i < length &&
        <%= leftOperand %><%= comparator %><%= rightOperand %>
        <%= booleanComparison %>)
        max = right_i;
    if (max != index)
    {
        swap(list, max, index);
        heapify(list, max, length);
    }
}
Adaptation Methodology

- Use code generation techniques
  - Place tags in files
  - Use regular expressions to find tags
  - Replace tags with values found in concrete operator files
  - Flag values are determined from parent abstract operators
import java.io.*;
import java.util.*;

public class integerSort
{
    private static void convert1(String x[], int y[])
    {
        Convert.stringToInt(x, y);
    }

    private static void sort1(int x[])
    {
        Sort.heapSort(x);
    }

    private static void store1(int x[], String format)
    {
        Store.storeToConsole(x, format);
    }

    public static void main(String b[])
    {
        int a[] = new int[b.length];
        convert1(b, a);
        sort1(a);
        store1(a, "\v ");
    }
}
Outline

• Software Reuse Overview
• Problem Statement
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation

• Contributions

• Future Work
Contributions

• Dynamic specification methodology
• Automated application of abstraction and design knowledge
• Simplified Component Retrieval
Outline

• Software Reuse Overview
• Problem Statement
• Solution Overview
  – Specification
  – Abstraction
  – Retrieval/Adaptation
• Contributions
• **Future Work**
Future Work

• Implement current methodologies
• Develop further abstractions
  – Allow for different forms of representation
• Make specification system more powerful
  – Add primitive instructions
  – Add inter-abstraction abstractions
Questions?