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

- "The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks." (J. Greenfield & K. Short (2003). "*Software Factories"*)
- 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 Sytems - Generate executable code from specifications using a series of transformations on the spec while maintaining semantics.
  - Smith, D.R. (1990). “KIDS: A Semiautomatic Program Development Sytstem”.
- Program Synthesis – Generate code from formal specifications using mathematical theorem proving. Code is extracted automatically from a constructive proof of the specification
  - Mana, Z. and Waldinger, R (1992). “Fundamentals of Deductive Program Synthesis
  - Kreitz, C (2003). “Building Reliable, High-Performance Networks with the Nuprl Proof Development System”.
- Design Patterns – present successful solutions to common software problems
  - Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). <u>Design Patterns, Elements of Reusable Object-Oriented Software</u>.

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 be a specification of a program $S = \langle I, Ops_S, Ord_S \rangle$
- I be the set of inputs in the specification $(i_1, i_2, \ldots)$
- Ops_S be the set of operators in the specification $(op_1, op_2, \ldots)$
- Ord_S be the set of orderings on Ops_S $(ord_1, ord_2, \ldots)$ where
  - $ord_i = \langle op_j \prec op_k \rangle$
- $Real_{op\_i}$ be the realization of op_i, $Real_{op\_i} = \langle Ord_{R\_i}, C_{R\_i} \rangle$
- C_{R_i} be a set of components
- 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 Ord_{R_S} such that

- Each realization $Real_{op\_i}$ in R_S is semantically equivalent to the corresponding operator in S, op_i:
- $Real_{op\_i} \equiv_S op_i$
- $Ord_{R\_S} \equiv_S Ord_S$

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.

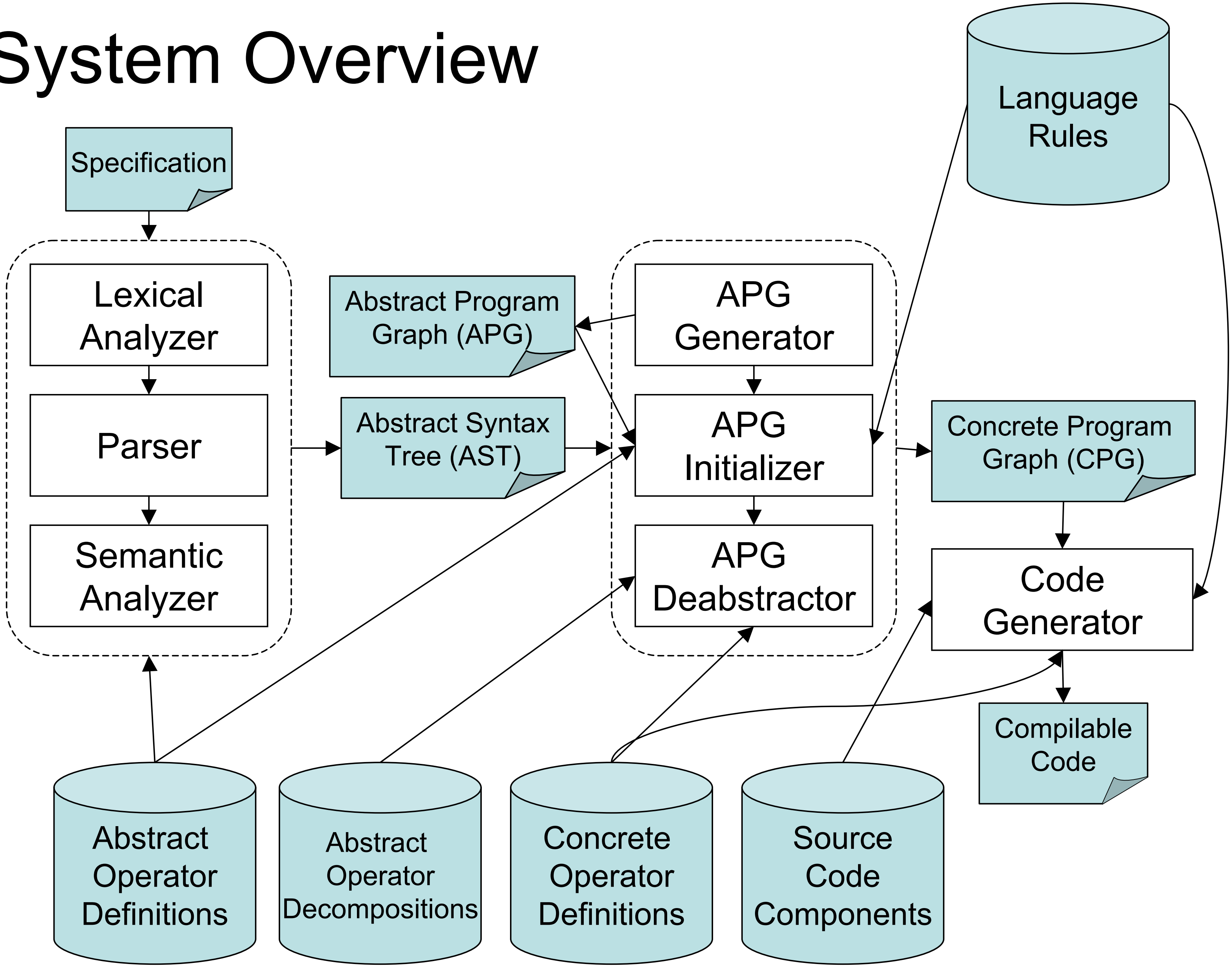
Outline

- 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

System Overview



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

    store
          data: a
          destination: "console"
          format:"\\v " forall i in a

end integer_sort
```

Variable declaration

Abstract Operators

Abstract Operator Definitions

- Schema based
- Basic syntax:

OperatorName

  Field1: field1Value

  …

  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

```
#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 algorithm allowed

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

Field of a non-basic data type

Basic types

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

Default value

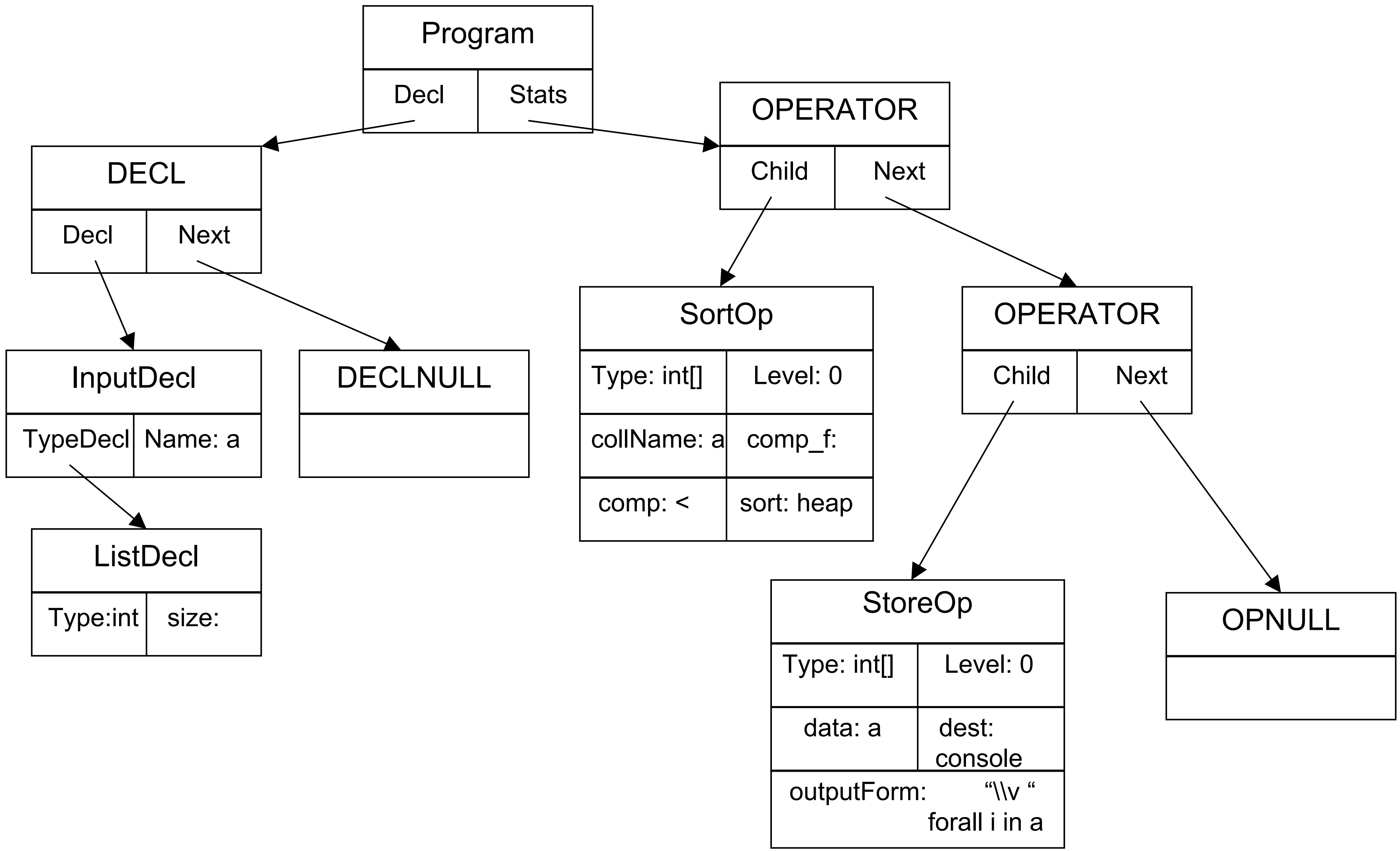
Methodological Motivation

- 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
Stats
DECL
Decl
Next
OPERATOR
Child
Next
InputDecl
TypeDecl
Name: a
DECLNULL
ListDecl
Type:int
size:
SortOp
Type: int[]
Level: 0
collName: a
comp_f:
comp: <
sort: heap
OPERATOR
Child
Next
StoreOp
Type: int[]
Level: 0
data: a
dest: console
outputForm: “\\v “ forall i in a
OPNULL

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

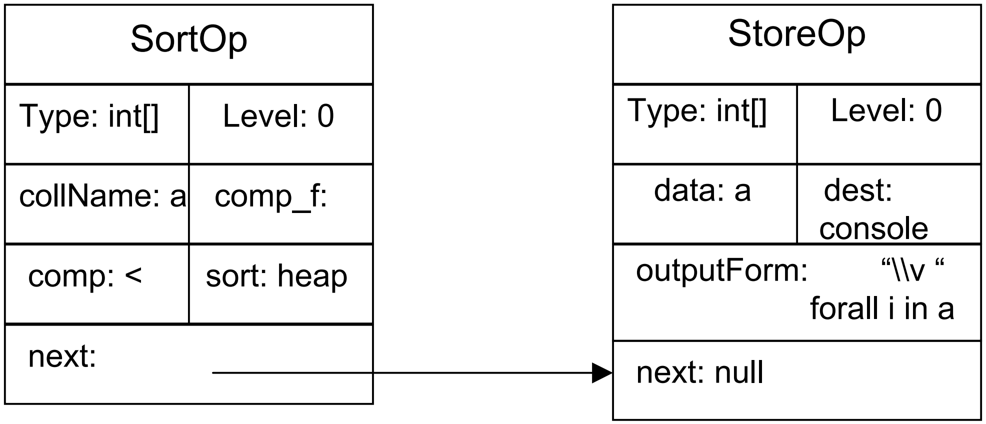
1. Generate the APG from the AST
2. Find implicit and necessary abstractions in the initial APG to create the working APG
3. Use the working APG to decompose abstractions until only concrete operators remain

1. Generating the APG

```
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


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

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
```

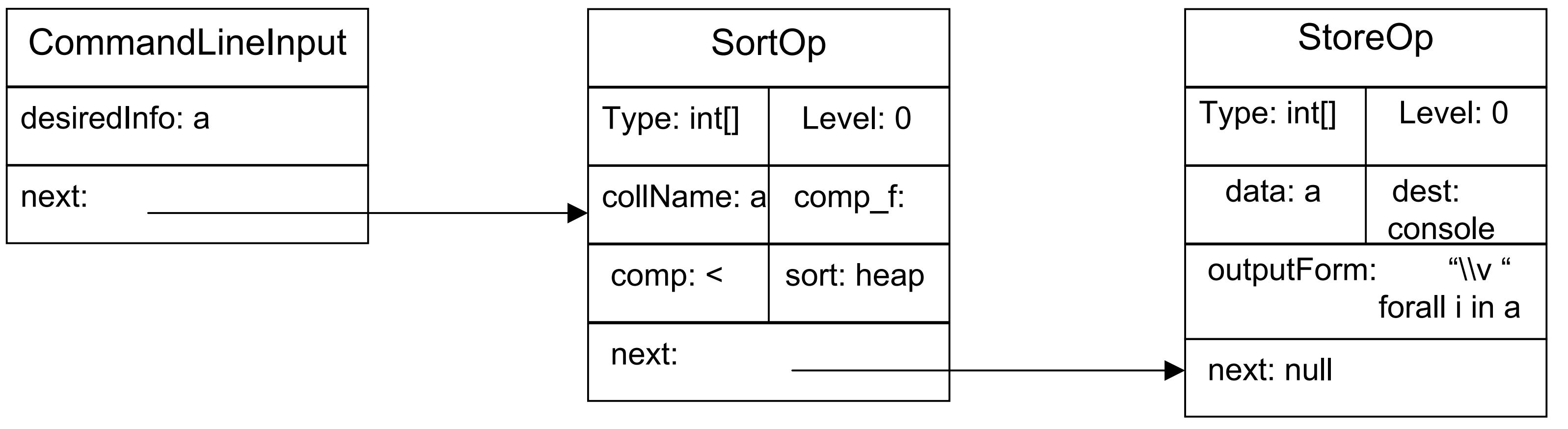
Disconnects and Error Handlers

- 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

```
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 decompostions
  - 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

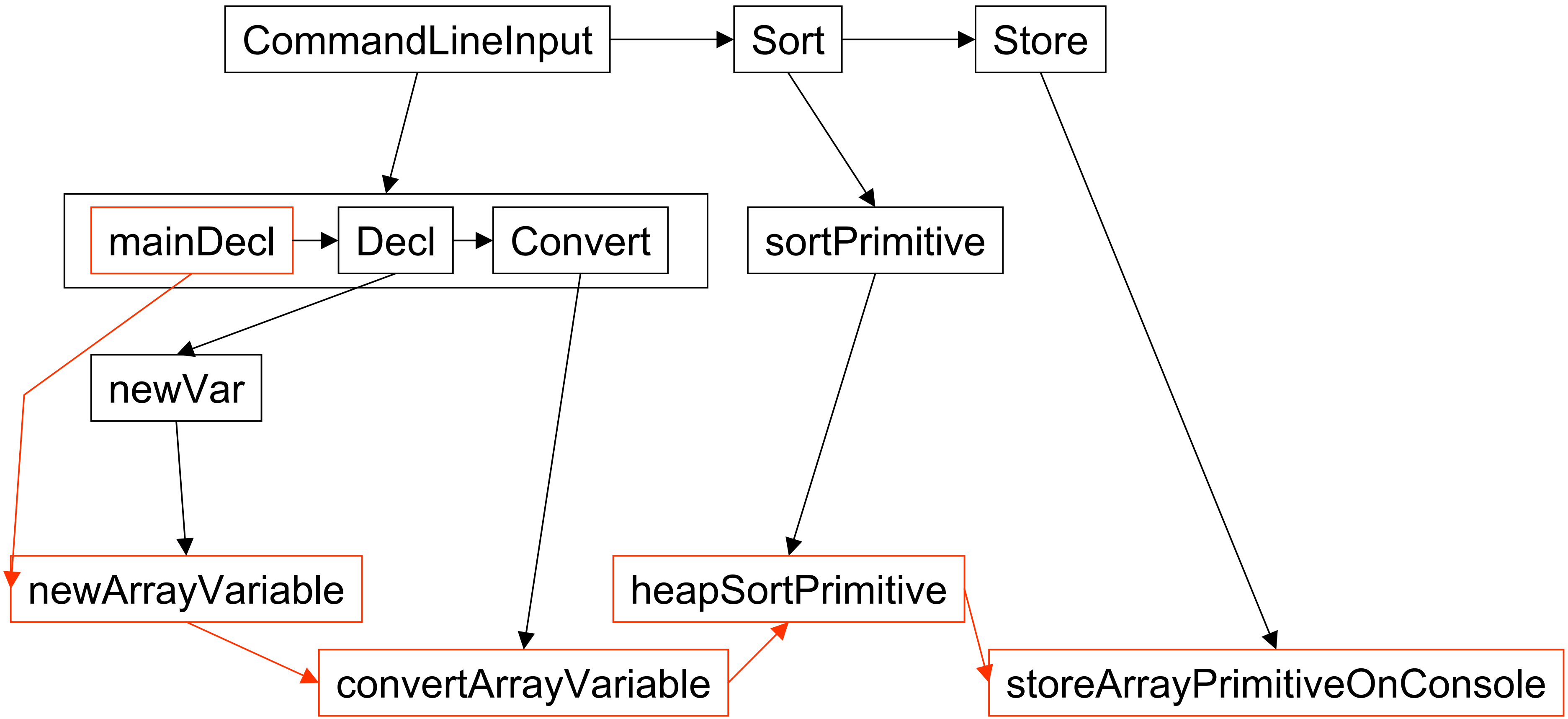
ordering mainDecl -> Decl

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



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

Concrete Operator Example

```
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 %>);\n"
        wrapper = "private static void <%= localFunctionName %>(<%= sortArgType %> <%= sortArg %>)\n{\n\t
                       <%= functionCall %>\n}\n"
        functionCall = "Sort.<%= libraryFunctionName %>(<%= sortArg %>);\n"
        availableOperators = {"heapSort"}
        void decideOp()
        {
                   int flagIndex = flags.indexOf("<%= libraryFunctionName %>");
                   flagValues[flagIndex] = availableOperators[0];
        }
}
```

Concrete Declaration Example

```
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 = ""
}
```

Example Code Skeleton

```
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 %>
      }
}
```

Example Source Code Component

```
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);
    }
}
```

```
private static void buildHeap(<%= listType %> list[])
{
    for (int i = list.length / 2; i >=0; i--)
        heapify(list, i, list.length);
}

public static void heapSort(<%= listType %> list[])
{
    buildHeap(list);
    for (int i = list.length - 1; i >= 1; i--)
    {
        swap(list, 0, i);
        heapify(list, 0, i);
    }
}
```

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

Finished Code

```
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?