JTS: Tools for Implementing Domain-Specific Languages

Don Batory, Bernie Lofaso, and Yannis Smaragdakis
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{batory, bernie, smaragd}@cs.utexas.edu

Abstract[1]

The Jakarta Tool Suite (JTS) aims to reduce substantially the cost of generator development by providing domain-independent tools for creating domain-specific languages and component-based generators called GenVoca generators. JTS is a set of precompiler-compiler tools for extending industrial programming languages (e.g., Java) with domain-specific constructs. JTS is itself a GenVoca generator, where precompilers for JTS-extended languages are constructed from components.

1 Introduction

Software generators are among the most effective methods of achieving software reuse. Generators reduce maintenance costs, produce more evolvable software, and provide significant increases in software productivity [Deu97, Die97, Kie96]. From a technical standpoint, generators are compilers for *domain-specific languages* (*DSL*s) or general-purpose programming languages with domain-specific extensions [Cor90, Sma97]. Such languages express fundamental abstractions of a domain using high-level programming constructs. The P2 data structure generator is an example: P2 extended the C language with cursor and container data types [Bat93-94]. This allowed P2 users to program in data-structure-specific abstractions, which resulted in substantial improvements in productivity, program clarity, and performance.

Implementing a domain-specific language as an extension of an existing programming language (called a *host language*) has several advantages. First, we can leverage off existing functionality and not have to re-implement common language constructs. Second, the extensions themselves only need to be transformed to the point where they are expressible in the host language. Third, existing infrastructure (e.g., development and debugging environments) can be reused. All these factors result into lower implementation costs for language developers and decreased transition and education costs for users.

Nevertheless, adding domain-specific constructs to a general programming language presents severe technical difficulties. Programming languages are generally not designed to be extensible, and the ones that are (e.g., Lisp and a variety of other functional languages) have not gained wide acceptance. Addressing the needs of the industry (where C, C++, and Java prevail) is paramount for promulgating generator technology. Our interest in DSLs comes from our work in the design and construction of component-based generators, called *GenVoca* generators [Bat92-97]. Target applications are specified as compositions of reusable components; GenVoca generators convert these compositions into source code. From our experience, there is a serious lack of tools to simplify the construction of these generators. We estimate that over 60% of the effort in building a GenVoca generator involves the creation of a largely domain-independent programming infrastructure (e.g., component specification languages, component composition languages, etc.).

The *Jakarta Tool Suite (JTS)* is aimed at providing this common infrastructure: it is a set of domain-independent tools for extending industrial programming languages with domain-specific constructs. JTS is designed specifically for creating DSLs and GenVoca generators. JTS consists of two tools: Jak and Bali. The *Jak* language is an extensible superset of Java that supports meta-programming (i.e., features that allow Java programs to write other Java programs). *Bali* is a tool for composing grammars. JTS is itself a GenVoca generator. Languages and language extensions are encapsulated as reusable components. A *JTS component* consists of a Bali grammar file

1. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

(which defines the syntax of a language or extension) and a set of Jak files (that define the semantics of the extension as syntactic transformations). Different combinations of these components yield different language variants. Bali and Jak work cooperatively to automatically convert a composition of components that defines a language variant into a preprocessor for that variant.

The implementation of JTS is bootstrapped: JTS is written in Jak and Jak is also the first customized language that has been produced by JTS. (That is, new extensions are written in Jak (Java); the Jak preprocessor is then extended by this new component so that it can be used to write other extensions, and so on). In the following sections, we review the current features of Jak and Bali and explain the strategy behind their implementation. Afterwards, we explain the novelty of JTS and differentiate JTS from other language specification and construction tools.

2 The Jak Language

Jak is an open, extensible superset of Java. It extends Java with support for meta-programming (i.e., features that enable Java programs to write other Java programs). In the following sections, we explain two key features of Jak —namely, AST constructors and Generation Scoping — that distinguish it from Java. Both have been implemented as JTS components and are examples of the kinds of language extensions that JTS is capable of expressing.

2.1 AST Constructors

JTS internally represents programs and code fragments as two kinds of trees. A *surface syntax tree (SST)* is a parse tree of a code fragment (as defined by some grammar). An *abstract syntax tree (AST)* is a semantically-checked SST that has been annotated with type declarations and references to the symbol table. An SST is converted into an AST by invoking the **`typecheck()`** method on the root of the tree. In this section, we present the (surface syntax) tree constructors and composition methods in Jak.

A *tree constructor* is a code-template operator, analogous to the Lisp **`quote`** construct. It converts a code fragment into an SST; the value of a constructor is a pointer to the root. The expression constructor **`exp{`** … **`}exp`**, for example, encloses a syntactically correct Jak expression. When the constructor is evaluated, an SST for that expression is created, and the root of that tree is the result. Similarly, **`stm{`** … **`}stm`** is the corresponding constructor for Jak statements. SSTs can be unparsed (into text) using the **`print()`** method:

```
AST_Exp x = exp{ 7 + z*8 }exp;
AST_Stm s = stm{ foo(3);
                 if (y<4) return r; }stm;

x.print( );    // outputs “7 + z*8”
s.print( );    // outputs “foo(3);
               //    if (y<4) return r;”
```

There are presently 17 different tree constructors in Jak, the most commonly used are listed in Table 1.

Code fragments are composed using *escapes*, the counterpart to the Lisp **`comma`** (unquote) construct. The example below shows a statement constructor with an escape **`$stm(body)`**. When the constructor is evaluated, the SST of **`body`** is substituted in the position at which its escape clause appears.

```
AST_Stm body = stm{ if(i>40) foo(i); }stm;
AST_Stm loop = stm{ for(i=1; i<10; i++) {
                       $stm(body); } }stm;

loop.print();
       // outputs “for (i=1; i<10;i++)
       // { if (i > 40) foo(i); }”
```

Unlike Lisp and Scheme which have only a single constructor operator (e.g., backquote/comma), multiple constructors in syntactically rich languages are common (e.g., [Wei93], [Chi96]). The main reason has to do with the ease of parsing code fragments. We avoided the complications described in [Wei93] by making explicit the type of SST that is returned by a tree constructor. The result is a slightly more complicated but robust system.

Constructor	Escape	Class	AST Representation Of
`exp{` … `}exp`	`$exp(` … `)`	`AST_Exp`	expression
`stm{` … `}stm`	`$stm(` … `)`	`AST_Stmt`	list of statements
`mth{` … `}mth`	`$mth(` … `)`	`AST_FieldDecl`	list of data member and method declarations
`cls{` … `}cls`	`$cls(` … `)`	`AST_Class`	list of class and interface declarations
`id{` … `}id`	`$id(` … `)`	`AST_QualifiedName`	qualified name

Table 1: AST Constructors and Escapes

Although the tree constructors of Table 1 are presently specific to Jak, this will not always be the case. Tree constructors can be added for other languages, such as CORBA IDL, embedded SQL, (subsets of) C and C++, so that IDL code, embedded SQL code, etc. can be generated.

2.2 Generation Scoping

Tree constructors and escapes are not sufficient for code generators (meta-programs); there must also be a mechanism to solve the *variable binding* or *inadvertent capture* problem [Koh86], which arises when independently-written code fragments are composed. Consider the following parameterized macro that defines a variable **temp** and initializes it to be twice the value of parameter **x**:

```
macro(x) { int temp = 2*x; ... }
```

Now consider application code that defines a variable, also called **temp**, and that invokes **macro(temp)**:

```
int temp = 5;
macro(temp);
```

The code that is produced on expansion is incorrect:

```
int temp = 5;
{ int temp = 2*temp;  ... }    // wrong
```

The inner **temp** variable was to be initialized using the outer **temp** variable; instead the uninitialized inner **temp** variable is used to initialize itself! The problem is that the **temp** identifiers are not sufficient to disambiguate the variables that they reference.

Hygienic, lexically-scoped macros (HLSM) were designed to solve this problem. HLSM relies on a "painting" algorithm that ensures identifiers are bound to the correct variables [Ree91]. Often, HLSM is implemented as a preprocessing step that mangles variable names to ensure their uniqueness:

```
int temp_0 = 5;
{ int temp_1 = 2*temp_0; ... }  // right
```

HLSM's applicability is limited to *macros* (pattern-based source code transformations). Since JTS supports programmatic (as opposed to macro or pattern-based) tree construction, we devised *Generation Scoping (GS)*, an adaptation and generalization of HLSM that is suited for JTS. We originally developed GS for Microsoft's Intentional Programming (IP) system [Sim95], and used it to develop the DiSTiL generator, an IP-version of the P2 generator [Sma96-97]. The IP implementation of GS used handles to symbol table entries to represent variable references (see also [Tah97]). Since JTS produces domain-specific preprocessors, we chose an alternative implementation that mangles identifiers. In the following sections, we review its features.

2.2.1 GS Environments

A GS *environment* is a list of identifiers (i.e., class or interface names, data member or method names, etc.) that are local to a set of related code fragments. To ensure there is no inadvertent capture, local identifiers are mangled. Associated with each environment instance is a unique *mangle number*, an integer that is attached to an identifier to make it unique. For example, if an environment's mangle number is **005** and identifier **i** is to be mangled, identifier **i_005** is produced.

Environments are associated with classes; environment instances are associated with objects. Class **foo** below defines an **environment** with identifiers **i** and **j**. Each **foo** instance creates an environment containing identifiers **i** and **j**. Different **foo** instances represent distinct environment instances. Whenever a tree constructor is evaluated by a **foo** object, it does so in the context of that object's environment. Thus, if **x** and **y** are distinct **foo** instances, and **x.bar()** and **y.bar()** return code fragments, the returned fragments will be isomorphic in structure, but will have different names for **i** and **j**.

```
class foo {
   environment i, j;   // ids to mangle
   AST_Exp bar() { return exp{ i+j }exp; }
}

foo x = new foo();// assume mangle# is 000
foo y = new foo();// assume mangle# is 001

x.bar().print(); // yields "i_000+j_000"
y.bar().print(); // yields "i_001+j_001"
```

With the above capabilities, the variable binding problem presented earlier is easily avoided. One defines a class (**macroExample**) with an **environment** that contains the **temp** identifier. A method of this class (**macroCode**) uses a tree constructor to manufacture the body of the "macro". The **temp** variable that is defined internally to that tree is given a unique name via mangling, so inadvertent capture can not arise.

```
class macroExample {
   environment temp;

   AST_Stmt macroCode(AST_QualifiedName n)
   {   return stm{ int temp = 2*$id(n);
                   ... }stm;
   }
}
```

Since identifiers in an environment need to be explicitly designated, the JTS version of generation scoping is not fully automatic.[2] Associating environments with objects does, however, represent an improvement compared to the explicit creation of unique identifiers (as with Lisp's **gensym** [Gra96]) and the manual substitution of mangled names (via explicit escapes) into generated code fragments. Identifiers are now encapsulated and can be treated as a group. Additionally, these groups can be arranged in complex configurations, as we will see next.

2.2.2 GS Environment Hierarchies

Environment instances can be organized hierarchically to emulate scopes in the name space of generated programs. As expected, identifiers of parent environments are visible in child environments and identifiers that are declared in a child environment hide identifiers in parent environments with the same name. Parent linkages among environments are made at meta-program run-time using the **environment parent** declaration. The example below shows that instances of class **baz** make their environments children of environments of **foo** objects. Note that a tree constructor for the expression **"i + k"** produces **"i_000 + k_002"** because identifier **i** is mangled by the **foo** environment while **k** is mangled by the **baz** environment:

```
class baz {
   environment k;
   baz( foo z ) { environment parent z; }

   AST_Exp biff() {return exp{ i+k }exp; }
}
```

```
baz r = new baz(x);  // x has mangle # 000
                     // r has mangle # 002

r.biff().print();  // yields "i_000+k_002"
```

More generally, generation scoping allows environment instances to be arranged in directed acyclic graphs. This permits the visibility of identifiers from multiple parent environments, which is indispensable when building GenVoca generators. Detailed examples of generation scoping are presented in [Sma96].

2.3 Tree Traversals

Jak provides a Java package of classes for searching and editing trees using objects of type **Ast_Cursor**. Methods that can be performed on cursors are listed in Table 2.[3] In the code fragment below, a cursor **c** is used to examine every node of a tree and subtrees that define interface declarations are deleted.

```
Ast_Cursor c  = new Ast_Cursor();
Ast_Node Root = // root of AST to search

for(c.First(root); c.More(); c.PlusPlus())
   if (c.node instanceOf Ast_Interface)
       c.Delete();
```

2.4 Jak Extensibility

Representing programs internally as parse trees offers a powerful form of language extensibility. This principle has been widely explored in the Lisp community and various syntax tree formats are commonly used in transformations systems (e.g., Microsoft's IP [Sim95], Open C++ [Chi96]). New kinds of tree nodes can have domain-spe-

2. The IP version of GS automatically enters identifiers into environments as tree constructors are evaluated. The JTS version reflects a design that was used in the P2 generator, where manual declaration of identifiers was used.

3. Tree editing methods guarantee syntactic correctness; however, they cannot guarantee semantic correctness.

Cursor Operation	Meaning
First(r)	position cursor on root (**r**) of tree
More()	true if more nodes to examine in tree
PlusPlus()	advance cursor to next node of tree
Sibling()	skip the search of subtrees of current node
Parent()	reposition to parent of current subtree
Delete()	delete subtree rooted at current node
Replace(x)	replace the current node with **x**
AddAfter(x)	add tree **x** after the current node
AddBefore(x)	add tree **x** before the current node
print()	unparse the tree rooted at the current node

Table 2: Operations on Tree Cursors

cific semantics and transform, at reduction time, to a host language implementation (or, more accurately, a tree that defines this implementation). This approach is called *intention-based* programming[4] [Sim95]. For example, the P2 language extended the C language with cursor and container data types and operators. Tree nodes that represented these types and operations were intentions. At reduction time, a P2 program was converted into a pure C program by replacing cursor and container nodes with trees that defined their concrete C implementations.

JTS follows this approach (see Figure 1). A domain-specific program is converted into an AST by a lexical analyzer (lexer) and parser. The AST is then manipulated into another tree by a reduction program, and the resulting tree is unparsed into a pure host-language program (currently a Java program). Note that the reduction program itself is written in Jak, because Jak is specifically designed for tree creation and manipulation.

2.5 Perspective

Jak is an integration of a popular programming language (Java), with meta-programming concepts (tree constructors and generation scoping), and intention-based programming. The structure of Jak provides the basis for an inherently open precompiler. In the following sections, we answer the following questions:

- How are lexers and parsers produced by JTS?
- How is the reduction program produced by JTS?
- How is GenVoca related to JTS?

3 Bali: A GenVoca Generator of DSL Precompilers

Bali is the second tool of JTS. There are two aspects to Bali. First, it is a tool for writing precompilers for domain-specific languages. In this respect, Bali looks similar to other DSL-specification tools: the syntax of a DSL or language extension is specified using an annotated, extended BNF grammar. Second, Bali is a GenVoca generator. DSLs and their precompilers are specified as a composition of components; evolution of a DSL (e.g., adding and removing features) is accomplished by adding and removing components.

To show that Bali is a GenVoca generator, we will examine one of its most important applications: Jak itself. Jak is a preprocessor implemented as an extended version of Java using Bali. The reasoning behind this design decision is simple. Jak is really not a single language, but a family of related languages. There will be variants of Jak with/without generation scoping, variants with/without tree constructors, variants with/without CORBA IDL extensions, and so on. This a classical example of the *library scalability problem* [Bat93, Big94]: there are *n* features and often more than *n*! valid combinations (because composition order matters and feature replication is possible [Bat92]). It isn't possible or practical to build all combinations by hand. Instead, the specific instances that are needed can be generated. The JTS library presently includes components for the Java language, tree constructors, generation scoping, and domain-specific generators (e.g., P3 [Bat98]). Compositions of these components define a particular variant of Jak.

3.1 Bali as a DSL Compiler Tool

Bali transforms a Bali grammar into a preprocessor. A Bali grammar is BNF with regular-expression repetitions. For example, two Bali productions are shown below: one defines **`StatementList`** as a sequence of one or more **`Statements`**, while **`ArgumentList`** is defined as a sequence of one or more **`Arguments`** separated by commas. The use of repetitions simplifies grammar specifications [Wil93, Rea90a] and allows an efficient internal representation as a list of trees.

```
StatementList : ( Statement )+ ;
ArgumentList : Argument ( ',' Argument )*;
```

Bali productions are annotated by the class of objects that is to be instantiated when the production is recognized. For example, consider the Bali specification of the Jak **`SelectStmt`** rule:

4. Although the term is new, the idea is quite old. Lisp macros were powerful enough to express useful extensions to the language. They have been routinely used to encode new constructs in terms of core language constructs. We prefer, however, to use the term "macro" exclusively for pattern-based program transformations.

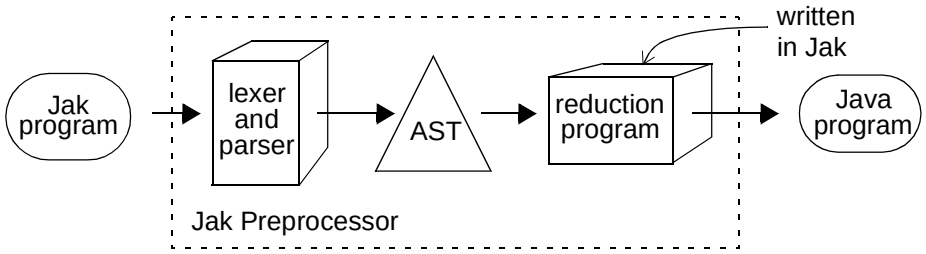

Figure 1: Data Flow in Jak

```
SelectStmt
 : IF '(' Expression ')' Statement ::IfStm
 | SWITCH '(' Expression ')' Block ::SwStm
 ;
```

When a parser recognizes an "if" statement (i.e., an **IF** token, followed by '**(**', **Expression**, '**)**', and **Statement**), an object of class **IfStm** is created. Similarly, when the pattern defining a "switch" statement (a **SWITCH** token followed by '**(**', **Expression**, '**)**', and **Block**) is recognized, an object of class **SwStm** is created. As a program is parsed, the parser instantiates the classes that annotate productions, and links these objects together to produce the SST of that program.

For each production, Bali infers (among other things) the constructors for tree nodes. Each parameter of a constructor corresponds to a token or nonterminal of a pattern.[5] For example, the constructor of the **IfStm** class has the following signature:

```
IfStm( Token iftok, Token lp,
 Expression exp, Token rp, Statement stm )
```

Methods for editing and unparsing nodes are additionally generated.

Bali also deduces an inheritance hierarchy of tree node classes. Consider Figure 2a which shows rules **Rule1** and **Rule2**. When an instance of **Rule1** is parsed, it may be an instance of **pattern1** (an object of class **C1**), or an instance of **Rule2** (an object of class **Rule2**). Similarly, an instance of **Rule2** is either an instance of **pattern2** (an object of **C2**) or an instance of **pattern3** (an object of **C3**). From this information, the inheritance hierarchy of Figure 2b is constructed: classes **C1** and **Rule2** are subclasses of **Rule1**, and **C2** and **C3** are subclasses of **Rule2**.

A Bali grammar specification is a streamlined document. It is a list of the lexical patterns that define the tokens of the grammar followed by a list of annotated productions that define the grammar itself. A Bali grammar for an elementary integer calculator is shown in Figure 3. To give readers an idea of the size of other grammars, the Jak grammar uses 160 tokens, 270 productions, defines 290 classes in 750 lines. The "meta" grammar for Bali grammars uses 20 tokens, 20 productions, defines 37 classes in 100 lines.

5. The tokens need not be saved. However, Bali-produced precompilers presently save all white space — including comments — with tokens. In this way, JTS-produced tools that transform domain-specific programs will retain embedded comments. This is useful when debugging programs that have a mixture of generated and hand-written code, and is a necessary feature if transformed programs will subsequently be maintained by hand [Tok95].

```
        // Lexeme definitions

"print"  PRINT
"+"      PLUS
"-"      MINUS
"("      LPAREN
")"      RPAREN
"[0-9]*" INTEGER

%%      // production definitions
        // start rule is Action

Action  : PRINT Expr        :: Print
        ;

Expr    : Expr PLUS Expr    :: Plus
        | Expr MINUS Expr   :: Minus
        | MINUS Expr        :: UnaryMinus
        | LPAREN Expr RPAREN :: Paren
        | INTEGER           :: Integer
        ;
```

Figure 3: A Bali Grammar for an Integer Calculator

Bali generates the following from a grammar specification:

- *A lexical analyzer.* We are using **Jlex**, a version of **lex** written in Java [Lof96].
- *A parser.* We are using **JavaCup**, a version of **yacc** written in Java [Hud96].
- *Inheritance hierarchies of tree node classes, with constructor, editing, and unparsing methods.*

There are obviously many methods that cannot be generated by Bali, including type checking, reduction, and optimization methods. Such methods are node-specific; we hand-code these methods and encapsulate them as subclasses of Bali-generated classes. (The reason for this will become clear in Section 3.2). Figure 4 shows the inheritance hierarchy of a Bali-generated precompiler. **AstNode** is the root of all node hierarchies; it is a hand-written class

(a)

```
Rule1 : pattern1   :: C1
      | Rule2
      ;

Rule2 : pattern2   :: C2
      | pattern3   :: C3
      ;
```

(b)

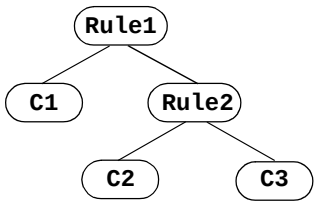

Figure 2: Grammar Inheritance Hierarchies

in the JTS **kernel** package. Its immediate subclasses are the hierarchy of subclasses generated from a Bali grammar. The terminal classes of this hierarchy are hand-coded and define the type checking, reduction, and optimization methods for individual nodes. It is these terminal classes that are instantiated during the compilation of a domain-specific program.

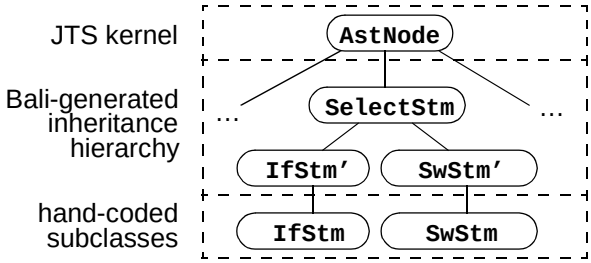

Figure 4: A Bali-Generated Class Hierarchy

3.2 Bali as a GenVoca Generator

GenVoca is a scalable model of component-based software construction [Bat92-97b]. The central idea is that software domains are characterized by a finite set of fundamental abstractions. By standardizing the programming interfaces to these abstractions, components can encapsulate reusable algorithms of a domain by exporting and importing standardized interfaces. A target system is specified by a composition of components called a *type equation*. Elementary compositions of components can be visualized as a stack of layers. Figure 5a depicts a system **S** where component **Z** sits atop **Y** which sits atop **X**. Its type equation is **S = Z[Y[X]]**.

GenVoca generators have been created for widely disparate domains. Interestingly, most have been written in the C language, and only two have been written in OO languages [Sin93-96, Van96]. A problem that we have faced for years but only very recently have been able to answer is: What is the relationship between GenVoca components and OO classes? The key lies in the relationship of layering and inheritance.

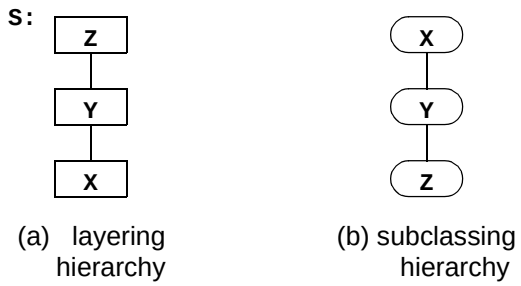

Figure 5: Layering and Subclassing Hierarchies

A common phenomenon in layered systems is *operation propagation* [Bat97b]: operations of lower layers are exported through the top of a system. In Figure 5a, suppose operation **g()** of layer **X** is to be exported by system **S**. This means that **g()** must be propagated through layers **Y** and **Z** (or in general, whatever layers are stacked above **X**). When an operation of **S** is called (such as **g()**), the corresponding operation of layer **Z** is invoked, which might call methods of layer **Y**, which further might call methods of **X**.

Now, suppose every component encapsulates a single class. To account for operation propagation *and* the processing of operations in layered systems, the subclassing hierarchy of Figure 6b comes to mind. Operation **g()** of class **X** is propagated to classes **Y** and **Z** by inheritance. Invoking an operation of **S** (such as **g()**) invokes the corresponding operation of class **Z**, which might call methods

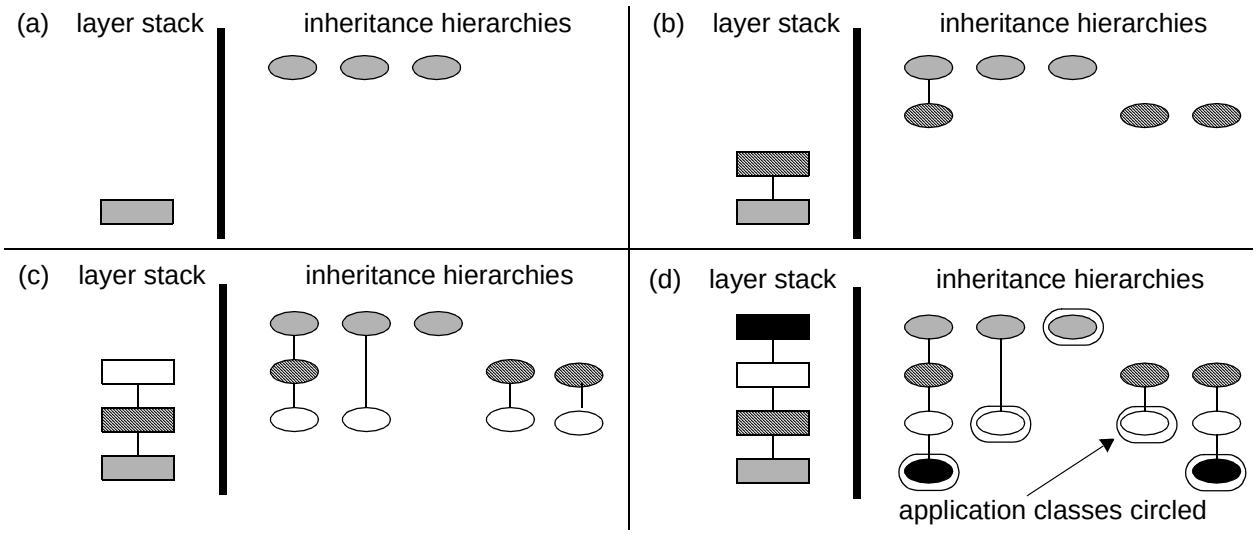

Figure 6: Component Composition and Inheritance Hierarchies

of class **Y**, which might further call methods of **X**. From this perspective, *inheritance hierarchies are inverted layer hierarchies*.[6]

The relationship of GenVoca components and OO classes can now be seen. *GenVoca components encapsulate suites of interrelated classes.* Figure 6a shows a (terminal) shaded layer that encapsulates three classes. Figure 6b shows a striped layer that also encapsulates three classes; when it is stacked on top of the shaded layer, one class becomes a subclass of the left-most shaded class, while the others begin new inheritance hierarchies. Figure 6c shows a white layer to encapsulate four classes. When stacked upon the striped layer, each of these classes becomes a subclass of an existing class. Lastly, Figure 6d shows the effect of adding a black layer, which encapsulates two classes. The application (which is defined by the resulting layer stack) instantiates the most refined classes (i.e., the terminal classes) of these hierarchies. These classes are circled in Figure 6d; the non-terminal classes represent intermediate derivations of the terminal classes. *Thus, when GenVoca components are composed, a forest of inheritance hierarchies is created. Adding a new component (stacking a new layer) causes the forest to get progressively broader and deeper* [Sma98].

The connection of these ideas to Bali and the Jak language is straightforward. A *JTS component* has two parts: The first is a Bali grammar file (which contains the lexical tokens and grammar rules that define the syntax of the host language or language extension). The second is a GenVoca component: a collection of multiple hand-coded classes that encapsulate the reduction, etc. methods for each tree node defined in that grammar file. These classes define the semantics of an extension. There are JTS components for Java (**Java**), SST constructors and explicit escapes (**SST**), generation scoping (**GScope**), and data structures (**P3** [Bat98]), among others.[7] The Jak language and precompiler is defined by a composition of these components, i.e., the type equation **Jak = P3[GScope[SST[Java]]]**.

The syntax of a composition is defined by taking the union of the sets of production rules in each JTS component grammar. The semantics of a composition is defined by composing the corresponding GenVoca components, as described previously. Figure 7 depicts the class hierarchy of the Jak precompiler. **AstNode** belongs to the JTS kernel, and is the root of all inheritance hierarchies that Bali generates. Using the composition grammar file (the union of the grammar files for the **Java**, **SST**, **GScope**, and **P3** components), Bali generates a hierarchy of classes that contain tree node constructors, unparsing, and editing methods. Each JTS component then grafts onto this hierarchy its hand-coded classes. These define the reduction, optimization, and type-checking methods of tree nodes by refining existing classes, just as in Figures 4 and 6. The terminal classes of this hierarchy are those that are instantiated by the generated preprocessor.

It is worth noting that Figure 7 is not drawn to scale. Jak consists of over 300 classes. The average number of classes that a language-extension component adds to an existing hierarchy ranges from 10 to 40. In terms of the number of classes a GenVoca component encapsulates, Bali components are clearly the largest we've ever encountered. However the simplicity and economy of specifying Jak using type equations is enormous: to build the Jak precompiler, all that users have to provide to Bali is the equation **Jak = P3[GScope[SST[Java]]]**, and Bali does the rest. To compose all these classes by hand (as would be required by Java) would be very slow, extremely tedious, and error prone. This is (another) good example why programming with reusable components (and hence at higher-levels of abstraction) offers big productivity gains. Additionally, the scalability advantages of GenVoca can easily be obtained: when new extension

6. Note that the converse is not true; there are layer hierarchies that are not inheritance hierarchies. Inheritance hierarchies arise whenever layer hierarchies refine a single abstraction (e.g., classes **X**, **Y**, and **Z** are different implementations of the same concept). When layers implement different abstractions, class composition relies solely on parameterization and does not involve inheritance.

7. Presently, Bali supports a single realm of components (**J**) that define and extend the Java language. Using the standard notation for realm definitions, **J = {Java, SST[J], GScope[J], P3[J],… }**.

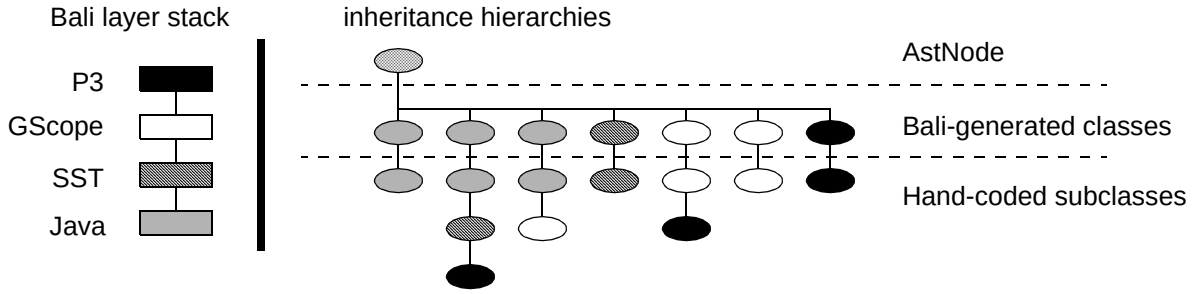

Figure 7: The Jak Inheritance Hierarchy

mechanisms or new base languages are specified as components, a subset of them can be selected and Bali will automatically compose a preprocessor for the desired language variant.

3.3 Perspective

The primary goal of JTS is to provide tool support for building GenVoca generators. Initially, it was unclear to us how language extensions could be encapsulated or composed; we feared that we would invent an ad hoc technology for defining and specifying components that was foreign to GenVoca. (This would then put a significant burden upon us to demonstrate the generality of this new model and its connection with other work, let alone how to address the odd situation of using a different component model to implement a more general model). Thus, realizing the connection between layering and inheritance hierarchies was a watershed event. It told us that JTS (or more specifically the central tool of JTS — Bali) was yet another example of GenVoca. Our focus immediately shifted away from an ad hoc implementation of Bali to one that would exhibit a principled and clean design.

4 Related Work

Meta-programming and syntactic transformations have been areas of active research for several decades. Since the volume of related work is enormous we will be selective in our presentation and only discuss approaches that are particularly close to JTS.

As should be evident from the previous sections, JTS is only concerned with the front-end of what is traditionally termed a *transformation system* (e.g., Draco [Nei89], Refine [Rea90b]). JTS mainly deals with parsing and the mechanics of syntactic transformation. Any sophistication of the transformation process (e.g., algebraic rewrites) will have to be provided by JTS client programs (e.g., the P3 generator [Bat98]).

Part of the novelty of JTS is that basic ideas of meta-programming and precompiler-compiler tools have transported to a "modern" and syntax-rich language platform (i.e., Java). The intricacies of our task are demonstrated, for instance, by the large variety of AST constructors discussed in Section 2.1 (compared to a single "quoting" operator for languages like Lisp).

Another contribution of JTS is in the way it achieves language extensibility: it does so through the prescripts of an architectural model (GenVoca): language extensions are encapsulated as components and languages and their preprocessors are assembled from these components.

It is instructive to compare this approach to that of Dialect [Rea90a]. Dialect is the front-end of the Refine transformation system and is in many ways analogous to the part of Bali described in Section 3.1. One of the biggest differences is in the way separate language extensions can be composed. By analogy to object-oriented programming, Dialect introduces the notion of *grammar inheritance*: a grammar (e.g., one defining a language extension) could "inherit" from another grammar (e.g., a base language). The resulting grammar is defined by taking the union of all productions contained in the two grammars — just like in JTS. An important difference, however, is that, unlike in Dialect, JTS grammars do not have to specify the grammar they are inheriting from. This is implicitly specified when grammar components are composed to form an entire language. The benefit is that a single JTS grammar component can be used to extend multiple base languages. Carrying the object-oriented programming analogy further, we could say that, instead of grammar inheritance, JTS allows *grammar mixins* (in the sense of OO *mixin classes* [Bra90]).

An interesting technical comment on comparing JTS with Dialect has to do with the way grammar rules are associated with inheritance between classes of AST nodes (see Section 3.1). Recall that JTS infers inheritance relationships from grammar rules. Conversely, Dialect requires that inheritance relationships be explicitly specified but infers grammar rules from them. The two approaches are semantically equivalent but we preferred having an explicit and compact representation of all grammar rules, as opposed to a mixed representation.

It is interesting to examine the relationship of JTS to *meta-object protocols (MOPs)*. The fundamental premise of a MOP is that class-specific extensions are themselves object-oriented in nature. Thus, they can be encapsulated in another class, called a *meta-class*. If a certain class *A* has meta-class *MA*, *A* is itself viewed as an object — an instance of *MA*. Methods of *MA* define extensions for all objects of *A*. For instance, methods of *MA* may define extension code for every construction of objects of class *A*, any assignment to such objects, or any method invocation on them.

Most MOPs are compositional: meta-classes contain code to be executed at a specified moment. There are, however MOPs where extensions are *transformational*: meta-classes contain code that transforms the source code of a class definition or use. The transformational MOP closest to JTS is Open C++ ([Chi95], [Chi96]). Open C++ encapsulates transformational extensions to C++ (i.e., syntax tree transforms, just like JTS) as meta-classes. Like JTS, Open C++ is implemented as a compiler that takes meta-class specifications as input and produces a preprocessor and compiler (packaged together) for extended C++ as output. Unlike JTS, however, no arbitrary syntactic extensions are allowed (the changes to the language syntax

are of one of a few pre-determined forms). The reason has to do with the complicated syntax of C++ and the difficulty of adding more syntax rules to it. The complexity of arbitrary syntactic extensions in JTS is what led us to represent them as GenVoca components. Compared with Open C++, the elements of JTS have direct counterparts: Jak corresponds to the meta-programming part (language for transformational extensions), while Bali is the counterpart of the meta-object protocol. Now we can see the role of JTS extensions as GenVoca components. Just like Open C++ (or any MOP) represents class-specific extensions as (meta-)classes, JTS represents arbitrary syntactic extensions as GenVoca components (encapsulated suites of classes). The main purpose of JTS has been to facilitate adding extensions for building GenVoca components in Java. By making the extension mechanism structure similar to that of the intended applications, the JTS design exhibits the same kind of simplicity and self-containment as meta-object protocols for object-oriented languages.

Microsoft's Intentional Programming (IP) system is a visionary project that has similar goals to JTS [Sim95]. IP inherently supports language extensibility through syntactic rewrites. It is not, however, concerned with surface language syntax but operates directly on an abstract syntax tree representation of a program. Additionally, IP transformations have no inherent knowledge of the semantics of any particular programming language. The purpose of IP is to become a powerful enough transformation system so that entire languages can be expressed as collections of cooperating transformations. We considered using IP but did not do so for reasons that had to do both with its current state (under development) and with our funding requirement for public availability of our code. Additionally, we were interested in experimenting with an extensible language system implemented around ideas from object-oriented and component-based programming.

5 Conclusions

Future software development tools will feature the generation and transformation of OO programs. Such tools will automate certain aspects of software design methodologies that aim at reuse, namely automating OO design patterns and generating domain-specific software from declarative specifications. The *Jakarta Tool Suite (JTS)* is designed with these applications in mind. JTS is a careful integration of three different technologies — meta-programming, precompiler-compiler tools, and component-based generators. JTS is also aimed at a growing community of software developers — those that use Java — who will benefit most from such tools.

The novelties of JTS are its integration of technologies and that JTS is an example of the very software design paradigm it was intended to support — GenVoca. With appropriate language support, it *is* substantially easier to write generators. And with clearly written and documented examples, it should be much easier to transition component-based generator technology from academic environments to industry.

In this paper, we have reviewed the two tools that comprise JTS: Jak and Bali. Jak is a JTS-produced language that extends Java with meta-programming features (e.g., tree constructors, generation scoping). Bali is the generator that produced Jak through component assemblies. The first GenVoca generator that we have built using JTS is P3 [Bat98], a Jak-based version of the P2 data structures generator. P3 was developed in a fraction of the time that was needed for P2. Moreover, the source code of P3 is *substantially* more elegant, readable, and maintainable because JTS provides the appropriate language constructs for building such generators. Further work on JTS will extend Jak to have language support for component definitions and compositions.

JTS runs on **Unix (Solaris)**, and **Windows (95** and **NT**) platforms. A beta-release became available in February 1998. For current information, release announcements, and the latest technical reports, please check our web page **`http://www.cs.utexas.edu/users/schwartz`**.

6 References

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat94] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT* 1994.

[Bat97a] G. Jimenez-Perez and D. Batory, "Memory Simulators and Software Generators", *1997 Symposium on Software Reuse.*

[Bat97b] D. Batory, "Component Validation and Subjectivity in GenVoca Generators", *IEEE Trans. Software Engineering, February 1997.*

[Bat98] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for Generators", *Int. Conference Software Reuse*, 1998.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, 1994.

[Bra90] G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

[Chi95] S. Chiba, "A Metaobject Protocol for C++", *OOPSLA 1995.*

[Chi96] S. Chiba, “Open C++ Programmer's Guide for Version 2”, SPL-96-024, Xerox PARC, 1996.

[Cor90] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow, “TXL: A Rapid Prototyping System for Programming Language Dialects”, *Computer Languages*, Vol. 16#1, 1991, 97-101.

[Deu97] A. van Deursen and P. Klint, “Little Languages: Little Maintenance?”, *Proc. SIGPLAN Workshop on Domain-Specific Languages*, 1997.

[Die97] P. Dietz, C. Jervis, M. Kogan, and T. Weigert, “Automated Generation of Marshalling Code from High-Level Specifications”, RNSG Research, Motorola, Schaumburg, Illinois, 1997.

[Gra96] P. Graham, *ANSI Common Lisp*, Prentice Hall, 1996.

[Hud96] S.E. Hudson, “Cup Users Manual”, Graphics Visualization and Usability Center, Georgia Institute of Technology, March 1996.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, “A Software Engineering Experiment in Software Component Generation”, *ICSE*1996.

[Koh86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, “Hygienic Macro Expansion”. In *SIGPLAN ‘86 ACM Conference on Lisp and Functional Programming*, 151-161.

[Lof96] B. Lofaso, “JLex Users Manual”, University of Texas Applied Research Laboratories, 1996.

[Nei89] J. Neighbors, “Draco: A Method for Engineering Reusable Software Components”. In *Software Reusability*, T.J. Biggerstaff and A. Perlis, eds, Addison-Wesley/ACM Press, 1989.

[Rea90a] Reasoning Systems, “Dialect User’s Guide”, Palo Alto, California, 1990.

[Rea90b] Reasoning Systems, “Refine 3.0 User’s Guide”, Palo Alto, California, 1990.

[Ree91] W. Clinger and J. Rees. “Macros that Work”. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991, 155-162.

[Sim95] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference, 1995.*

[Sin93] V. Singhal and D. Batory. “P++: A Language for Large-Scale Reusable Software Components”, *6th Annual Workshop on Software Reuse* (Owego, New York), Nov. 1993.

[Sin96] V.P. Singhal. “A Programming Language for Writing Domain-Specific Software System Generators”. Dept. of Computer Sciences, University of Texas at Austin, September 1996.

[Sma96] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. TR 96-37, Dept. of Computer Sciences, University of Texas at Austin, December 1996.

[Sma97] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conf. on Domain-Specific Languages*, 1997.

[Sma98] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components", *Int. Conference on Software Reuse* 1998.

[Tah97] W. Taha and T. Sheard, "Multi-Stage Programming with Explicit Annotations", *Partial Evaluation and Semantics Based Program Manipulation (PEPM'97)*, June, 1997, Amsterdam.

[Tok95] L. Tokuda and D. Batory, "Automated Software Evolution via Design Pattern Transformations", *Symp. on Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995.

[Wei93] D. Weise and R.Crew, “Programmable Syntax Macros”, *ACM SIGPLAN Notices* 28(6), 1993, 156-165.

[Wil93] D.S. Wile, “POPART: Producer of Parsers and Related Tools”, USC/ISI, November 1993.

[Van96] M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”, *OOPSLA 1996*, 359-369.