

Static Checking of Interoperating Components

Mark Grechanik, William R. Cook, Don Batory

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{gmark, cook, batory}@cs.utexas.edu

Karl J. Lieberherr

College of Computer and Information Science
Northeastern University
Boston, MA 02115-5000
lieberherr@ccs.neu.edu

ABSTRACT

Components (objects, modules, or programs) access and manipulate each others data. When components are written in the same language, the static checking of component references is well-understood. In contrast, when a component references another component, called a foreign component, that is written in a different language or that is hosted on a different platform, the static checking of references may not be performed, delaying discovery of type errors to runtime. *Foreign Object REification Language (FOREL)* is an extension of object-oriented languages that provides a common abstraction for foreign component access and manipulation. Using type checking and static analysis, illegal references to foreign components in FOREL can be discovered at compile-time. We describe our implementation, prove the soundness of the FOREL type system, and give our type checking algorithm. We used FOREL on a commercial system and discovered errors that were not detected during its initial design and testing.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features; D.2.12 [Interoperability]; D.2.4 [Software / Program Verification]: Validation

General Terms

Experimentation, Languages, Theory, Verification

Keywords

Interoperability, adaptive programming, foreign objects, static analysis, type inference

1. INTRODUCTION

Components are modular units (e.g., objects, modules, or programs) that interact by reading and writing each others data. Components are hosted on a platform, which is a collection of software packages. Platforms export *Application Programming Interfaces (APIs)* through which users can invoke platform services to access and manipulate hosted components. The *Java Virtual Machine (JVM)*, for

example, is a platform for running Java bytecodes; it exports APIs that can be used by non-Java programs to access and manipulate Java objects. An *eXtensible Markup Language (XML)* parser is a platform for reading XML files; it exports APIs that can be used to access and manipulate XML documents.

An abstract view of a component is a graph of interconnected objects. An XML document, when hosted by an XML parser, is a tree of *Document Object Model (DOM)* objects. A Java program, when hosted by a JVM, becomes a graph of Java objects. Path expressions can be used to navigate from one object to another in such graphs. The expression "a.b" means start at object named a and traverse to b, where b is the name of the member field of a that references some object. More generally, methods can replace field names in a path expression enabling computations to be performed enroute.

When components are written in the same language, conventional type checking algorithms can be used to verify the correctness of path expressions statically. Contemporary programming languages handle these common situations with great success. However, there are many other situations where the static type checking of path expressions is not attempted, resulting in the run-time discovery of type errors. This arises when a (*native*) component references a *foreign* component, which is written in a different language or that is hosted on a different platform. Consider the following examples.

A C program can manipulate Java objects by invoking JVM APIs. (For example, a static method can be located in a Java object by calling the JVM API `GetStaticMethodID` with the name of the method passed as a parameter). Broadly, JVM APIs take a string of a path expression as input, and the JVM interprets that string. If a member name is incorrect, or a method call has the wrong number of parameters, then these errors are discovered only at run-time. None of these errors will be detected by the type system of a C compiler.

Another example is accessing databases via *Java DataBase Connectivity (JDBC)*. SQL statements are passed as strings to JDBC APIs. At run-time, these strings are parsed and matched to a given database schema. If the target relation does not exist or attribute names are misspelled, the error is reported at run-time. A third example is a Java program accessing data from an XML document. The program uses DOM APIs to navigate a parsed XML document, where navigation methods take the string name of a desired XML node type as a parameter. If a name is incorrect, the error is discovered at run-time.

In each example, there is enough information to detect these errors statically at compile-time, as the Java/C executables or database/XML schemas are readily available. The problem is that the type systems for different components (e.g., a Java program, a C

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Submission to ICSE-2006, May 20-28, 2006, Shanghai, China
Copyright 2006 ACM 0-89791-88-6/97/05 ...\$5.00.

program, a database, an XML document) are not the same. While there is prior work to perform such static checks on the interaction of specific pairs of components (e.g., Java and databases) [15], there is no general algorithm to accomplish the static type checking of references to arbitrary foreign components.

In this paper, we describe and evaluate *Foreign Object Reification Language (FOREL)*, an extension for object-oriented languages that provides a general abstraction for foreign component access and manipulation. Components are abstracted as graphs of objects and a set of basic operations is provided for traversing these graphs to access and manipulate their objects. Traversal strategies, borrowed from adaptive programming [25], are used in FOREL to simplify foreign component navigation and manipulation.

FOREL type checking coupled with a conservative static analysis mechanism reports potential errors or ensures their absence when referencing foreign components. We formalize the type checking rules of FOREL, present its operational semantics, and prove the soundness of its type system. We define a static analysis of FOREL expressions that is based on the combination of algorithms defined in adaptive programming, the formal type system of FOREL, and our type inference algorithm. We tested FOREL on *Archer Analyzer*, a real-time semiconductor overlay analysis and control system geared for the optical overlay metrology tool *Archer 10* manufactured by California-based KLA-Tencor Corporation, and a number of smaller systems with interoperating components. We detected a number of known and unknown errors in these applications with high precision and a low false-positive rate thus proving the effectiveness of our approach.

2. PERSPECTIVE

Various technologies have been developed over the last twenty years to provide access to foreign components. All have problems and none are particularly general. The most common way to access foreign components is using an *Interface Description Language (IDL)*. IDL-based approaches (RPC, DCOM, CORBA) require programmers to define interfaces in implementation-language neutral language that can be translated into language-dependent client and server code using an IDL compiler. This approach suffers from multiple drawbacks; notably the necessity to deal with an additional foreign type system (the IDL), and to maintain client and server sets of code. IDLs do not simplify the accessing of databases or XML documents; other technologies are used.

JAXB and Apigen are examples of the generator-based approaches that provide APIs and tools to automate the mapping between XML documents and program objects. Various object-relational mappers automate the mapping between databases and programs. These tools are limited to specific environments, and are not substitutes for IDLs or JDBC. And they too have problems. If an XML or a database schema contains thousands of types, then thousands of corresponding classes are generated, one for each XML type. In addition to increased compile-times, obvious problems arise when an XML or a database schema evolves or when an XML type has the name of a reserved word in the target programming language. Work on type-checking schema references is similarly limited. While progress has been made to type-check database references [14], it is not at all evident how these approaches can generalize to accessing XML documents or objects in other programming languages.

Despite the dissimilarity of these technologies and the problems that they were designed to address, databases, XML documents and Java programs all share a common abstraction, namely, they are graphs of objects. How one accesses these objects varies among platforms, i.e., navigating and accessing Java objects via the JVM API is quite different than navigating and accessing XML nodes

via DOM parsers. Large systems frequently use many of these platforms. Consequently, programmers must learn many different ways to express exactly the same ideas. This contributes substantially to program complexity. This situation is common in software design: different packages implement the same functionality via different APIs. And the solution is also well-known: frameworks standardized a set of APIs and hide the details of their various implementations. (This is, in fact, the basis on which CORBA rests). In prior work, ROOF was our attempt to use a framework to reduce program complexity by providing an implementation-independent way to write path expressions to foreign objects objects that could be XML nodes, Java objects, or database tuples [Roof reference]. In this paper, we go a significant step further by showing that we can type-check such path expressions, unifying the analysis for a significant class of disparate problems.

3. FOREL

In this section, we present FOREL as a type system for object-oriented programs that provides a common abstraction for interoperating components. We describe the FOREL language through a series of examples. A more precise description of the type system is provided in Section 5.

3.1 Background

FOREL is based on *Reification Object-Oriented Framework (ROOF)* which abstracts the common functionality of different platforms by presenting a small set of operations for navigating and manipulating foreign components [17]. These operations are navigating to foreign components, setting and getting their values, modifying their structures, and invoking their methods. The ROOF abstraction makes the task of static checking of interoperating components tractable by reducing the multiplicity of platform APIs to a small set of operations on foreign objects that are common to all platforms. By introducing a simple extension to grammars of object-oriented languages we enable the collection of information about foreign components at compile time. This information is used to perform static type checking in order to determine possible errors that could otherwise be detected only at the runtime.

Reification OPERator (ROPE) is a basic type in FOREL that denotes a communication channel between components through which they interact when running on different platforms. Its purpose is to make foreign components first-class entities in native programs (i.e., to reify them), and to enable programmers to perform operations on these components the same way they work with native components (i.e., by addressing them directly without the use of complicated platform APIs). ROPEs store the structures of foreign components in an intermediate format, and their abstraction enables programmers to concentrate on high-level operations on foreign components without getting bogged down in the low-level platform APIs.

3.2 A Concrete Example

ROPE objects are declared using the `ro` keyword followed by the location of the schema describing foreign components. In Figure 1, at line 1 ROPE `x` denotes a foreign object whose schema is given in the file `orgstr.xsd`. Line 2 navigates to the field `stock` of the foreign component `CEO`, obtains its value and stores it in the local variable `stockOptions` of type `int`. Line 3 sets the value of the field `salary` of the foreign object `CTO` that is contained in the foreign object `CEO` to the result of an expression that multiplies the value of the local variable `stockOptions` by real number `15.2` that is assumed to be the price of a single share. At line 7 ROPE `y` is declared pointing to the Java class `orgstr.class`.

Lines 4, 5, and 6 depict ROPEs inserting and deleting foreign objects. The VPEngineering component is created and aggregated in the CEO object at line 4 by invoking method InsertPart on the ROPE x. In line 5 field salary of the type float is created and inserted in the object VPEng, and in line 6 the object CTO with all of its fields is deleted.

3.3 Operations

FOREL operations include navigating to foreign components, reading their values into local variables, writing values from local variables into foreign components, and changing them by inserting, deleting, and copying fields. Invoking foreign methods using ROPEs and returning their results and storing them in local variables are also discussed.

3.3.1 Navigations

In FOREL, names of foreign components are specified as strings. Brackets [] are used to guard these names to prevent possible naming conflicts between names of foreign objects and names of native object and language keywords.

3.3.2 Invoking Foreign Methods

In FOREL navigating to a function in a foreign component results in its invocation and the retrieval of the resulting return value, if any. At lines 8 and 9 foreign method IncreaseStock of the component CEO is invoked to increase the number of shares owned by the CEO, and its return value specifying the resulting number of shares is put in the local variable newShares. The first parameter of this foreign method is named annual for an annual increase in stock options, and the second parameter is percent specifying the percentage of the increase.

4. CHECKING FOREL EXPRESSIONS

In this section we use a toy example to illustrate the algorithm used to check the correctness of FOREL expressions. Recall that the input to this algorithm is a native object written using FOREL and a schema describing foreign objects with which this native object interoperates. The schema is shown in Figure 2 as a graph whose nodes are foreign objects and the edges are the references between them. Consider the FOREL expression $x["s"][e1][e2]["d"]$, where x is a ROPE object connecting these objects and navigating from the foreign object named "s" to the foreign object named "d". Expressions e1 and e2 compute values of the intermediate nodes in the traversal path at runtime. Expressions e1 and e2 are replaced with the object name variables α and β correspondingly, and the original expression is converted into traversal strategy $S = s \rightarrow \alpha \rightarrow \beta \rightarrow d$. Then an inference algorithm is invoked on this strategy to compute sets of values for the variables α and β .

The gist of the algorithm is in deciding which foreign objects should be visited from the node s in the graph in order to reach the

```

1: ro ["orgstr.xsd"] x = new ro;
2: int stockOptions=x["CEO"]["stock"];
3: x["CEO"]["CTO"]["salary"]=stockOptions*15.2;
4: x["CEO"].InsertPart("VPEngineering","VPEng");
5: x["CEO"]["CFO"].InsertField("Salary","float");
6: x["CEO"].DeletePart("CTO");
7: ro ["orgstr.class"] y = new ro;
8: int newShares = y["CEO"]["IncreaseStock"].
9:   Invoke(<"annual"=1><"percent"=10>);

```

Figure 1: A fragment of FOREL program.

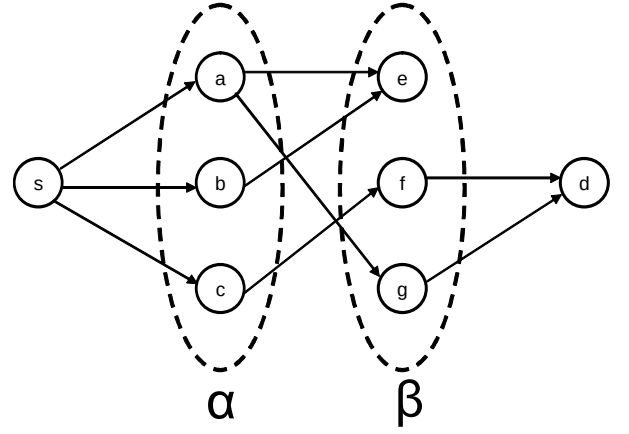


Figure 2: Navigation paths from the source object named s to the destination object named d.

destination node d given a strategy S. By finding all paths leading to the destination node d starting with the source node s we can conclude whether the navigation expression is correct. If no path exists, then this expression is incorrect. Alternatively, if there are paths leading from s to d by traversing two objects, then this expression is correct provided that expressions e1 and e2 evaluate to the names of the nodes in the discovered paths. We describe this algorithm informally in this section. Its formal description will be given shortly.

We introduce the function $first(s)$ computing a set of edges that can be traversed from node s. These edges lead to a set of objects designated by the variable α . Function $first(s)$ is computed using a graph reachability algorithm, and it returns edges that could lead to the target node. According to Figure 2, $\alpha = \{a, b, c\}$. Then for each element of α , function $first$ is computed. As a result we obtain $\beta = \{e, f, g\}$, where $first(a) = \{e, g\}$, $first(b) = \{e\}$, and $first(c) = \{f\}$. Proceeding to the next step we obtain $first(e) = \{d\}$, $first(f) = \{d\}$, and $first(g) = \{d\}$. From the computed node values a worklist W is formed that contains a set of all computed paths, $W = \{(s, a, e), (s, a, g, d), (s, b, e), (s, c, f, d)\}$. Each path is taken from W and checked to see whether it contains nodes s and d. If both nodes are present in the path as its source and target, then a valid path is obtained. If no paths exist, then an error is issued and this algorithm terminates.

An example of an incorrect FOREL expression is $x["s"][e1][e2][e3]["d"]$. All paths between nodes s and d have at most two objects. Therefore no matter what values are computed at runtime for expressions e1, e2, and e3 they cannot represent objects in a valid path between the source and the destination objects.

5. FORMALIZATION

A key property of FOREL is in ensuring the correctness of operations on foreign components. If a native component accesses nonexistent foreign components or attempts to set a value of a foreign component that is not compatible with its type, then the FOREL compiler should produce an error. A FOREL compiler type checks foreign components and guarantees that incorrect operations cannot be executed. Type checking rules should be based on a solid foundation and they should be guaranteed correct. We now precisely define type graphs, paths, and traversals and use these definitions as a foundation for the formalization of FOREL.

5.1 Type Graphs, Paths, and Traversals

Components are described by their types that are sets of fields and methods (i.e., method signatures including types of their parameters in the specified order and return types). A field of a component is a named object (component) of some type. Data component types have only sets of fields. Component types can be complex or simple. Complex component types can contain fields and simple types cannot. For example, XML tags containing other children tags are of complex types, and XML attributes are of simple types since they do not contain other tags and attributes. We use terms type and class interchangeably. An object is an instance of a class. We do not consider inheritance and subtyping relations between classes.

For simplicity we treat the types of all component uniformly as classes. We select class and object graphs used in adaptive programming [26] as common formalisms for FOREL to represent all components uniformly. Classes may contain fields that reference instances of some classes, or they may aggregate other classes by defining them within the scope of their parent classes. For example, an XML object may contain the definition of some other XML object thereby aggregating it.

Let T be finite sets of type names and F of field names or labels, and two distinct symbols $this \in F$ and $\diamond \in F$. Type graphs are directed graphs $G = (V, E, L)$ such that

- $V \subseteq T$, the nodes are type names;
- $L \subseteq F \cup \{\diamond\}$, edges are labeled by field names, or " $\diamond$ ". We call edges that are labeled by " $\diamond$ " *aggregation edges*, and edges that are labeled by field names *reference edges*;
- $E \subseteq L \times V \times V$, edges are cross-products of labels and nodes. If $\langle l, v_1, v_2 \rangle \in E$, we write $v_1 \xrightarrow{l} v_2$;
- for each $v \in V$, the labels of all outgoing edges with the exception of " $\diamond$ " are distinct;
- for each for each $v \in V$, where v represents a concrete type, $v \xrightarrow{this} v \in E$.

An object graph is a labeled directed graph $O = (V', E', L')$ that is an instance of a type graph $G = (V, E, L)$ under a given function $Class$ that maps objects to their classes, if the following conditions are satisfied:

- for all objects $o \in V'$, o is an instance of the concrete type given by function $Class(o)$;
- for each object $o \in V'$, the labels of its outgoing reference edges are exactly those of the set of labels of references of $Class(o)$ including edges and their labels inherited from parent classes;
- for each edge $o \xrightarrow{e} o' \in E'$, $Class(o)$ has a reference edge $v \xrightarrow{e} u$ such that v is a parent type of $Class(o)$ and u is a parent type of $Class(o')$.

An object graph is a model of the objects, represented in the heap or elsewhere, and their references to each other. A collection of fields in an object graph is a set of edges labeled by field names. A collection of aggregated objects in an object graph is a set of edges labeled by " $\diamond$ ". A path in a type graph $G = (V, E, L)$ is a sequence of nodes and labels $p_G = \langle v_0 e_1 v_1 e_2 \dots e_n v_n \rangle$, where $v_i \in V$ and $v_i \xrightarrow{e_{i+1}} v_{i+1}$ for $0 \leq i \leq n$. We define a concrete path to be an alternating sequence of object names and labels excluding

" $\diamond$ ". Since some classes are compiled away and are not present in the object graphs (e.g., namespaces in C++ are not a part of object representations), the paths in object graphs are subsets of the corresponding type graphs.

An object graph has the special object $o_r \in V'$, o_r represents a collection of root objects $o \subseteq V'$ in the object graph O given by function $root: O \rightarrow o_r$. This object has type $Class(o_r) = r_o$ and its relation with objects in its collection is expressed via $o_r \xrightarrow{\diamond} o' \in E'$.

5.2 Example of a Type Graph

An example of the type graph of the organizational structure of a company is shown in Figure 3. A FOREL program based on this graph is shown in Figure 1. CEO is a root component that has field *stock* of type *int* and aggregates type CTO. CTO is a type that has fields *salary* of type *Check* and *boss* of type CEO. Type *Check* has in turn fields *amount* of type *float* and *issuer* of type CEO. Instances of this type graph can be equally implemented in XML, Java, and other languages.

5.3 Formalization of a Traversal Problem

FOREL ROPE statements specify navigation paths through foreign components. Finding whether these paths exist is an instance of the reachability problem that is formulated as follows. Given an object o of some type in the object graph, find all reachable components that satisfy certain criteria. Reachable components are found by following edges starting from the object o . Every edge in the object graph is an image of a has-part edge in the type graph: there is an edge $e(o_1, o_2)$ in O only when there exist types v_1 and v_2 such that object o_1 is of type v_1 , v_1 has an e -part of type v_2 , and o_2 is of type v_2 .

The first node of a path p is called the *source* of p and the last node is called the *target* of p . The object graph O is traversed starting with an object v_1 and guided by paths from a set of paths p by performing depth-first search on O with p used to prune this search. The resulting traversal history is a depth-first traversal of the object graph along object paths agreeing with the given concrete path set.

We formalize the problem of finding all reachable objects from a given object o that satisfy certain criteria as follows. For each pair of classes c and c' , we need to find a set $FIRST(c, c')$ iff it is possible for an object of type c to reach an object of type c' by a path beginning with an edge e . More precisely, $FIRST(c, c') = \{e \in E, \text{ such that there exists an object graph } O \text{ of } C \text{ and objects } o \text{ and } o' \text{ such that:}$

1. $Class(o) = c$,
2. $Class(o') = c'$, and
3. $o \xrightarrow{e^*} o'$.

The last condition, $o \xrightarrow{e^*} o'$ says that there is a path from o to o' in the object graph, consisting of an edge labeled e , followed by any sequence of edges in the graph. Our lack of information about the actual graph is represented by the existential operator.

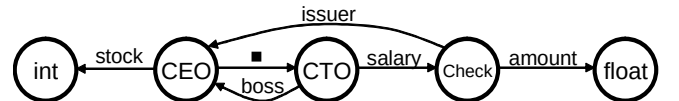


Figure 3: Type graph of the organizational structure of a company.

5.4 FOREL

We formalize the FOREL language based on the type graph model. We present the syntax of FOREL, give its operational semantics and type checking rules, and prove the soundness of its type system.

5.4.1 Syntax

Figure 4 presents a `ClassicJava`-based syntax of FOREL. The metavariable C ranges over class names; field and fd range over field declarations and field names respectively; τ ranges over types; meth ranges over methods; ρ ranges over reification operators; ROPE ranges over methods of reification operators; T ranges over variants that are used to specify values of parameters when invoking foreign methods, and defn ranges over class definitions. A program in FOREL, P , is a pair (defn, e) of class definitions and an expression.

FOREL extends `ClassicJava` in several ways. Reification operators designate communication channels between programs through which they interact, and they are represented by objects of type `ro`. Attribute L points to the locations of foreign components. T ranges over navigation and access expressions in ROPEs. An error expression is also included, representing failed casts and null dereferences. Operational semantics and reduction rules for FOREL are omitted from this paper for brevity and can be found in our technical report [18].

The rationale for having the `attribute` function is in disambiguating accesses to components and their fields. For example, a child component of XML may have the same name as an attribute of its parent component. Accessing them is ambiguous without specifying whether the destination component is an attribute or a tag.

5.4.2 Typing Rules

P	$::=$	$\overline{\text{defn}}\ e$
defn	$::=$	$\text{class } C \text{ extends } C \{ \overline{\text{field}} \overline{\text{meth}} \}$ $\mid \text{interface } i \text{ extends } i \{ \overline{\text{meth}} \}$
field	$::=$	$\tau \text{ fd}$
meth	$::=$	$\tau \text{ md}(\overline{\text{arg}}) \{ \text{body} \}$
arg	$::=$	$\tau \text{ var}$
body	$::=$	$e \mid \text{abstract}$
e	$::=$	$\text{new } C \mid \text{new } \rho \mid \text{new } \rho \text{ constraint } [L]$ $\mid \text{null} \mid \text{fd} \mid \text{fd} = e \mid e.\text{md}(\overline{e})$ $\mid \text{view } \tau \text{ e} \mid \text{let } \text{var} = e \text{ in } e \mid e.\overline{N}$ $\mid \text{valueOf}(e) \mid \text{ROPE} \mid n \mid s$
N	$::=$	$[e] \mid \text{attribute}(e) \mid (\overline{T})$
ROPE	$::=$	$e.\text{InsertPart}(e.e)$ $\mid e.\text{InsertField}(e.e)$ $\mid e.\text{DeletePart}(e.e)$ $\mid e.\text{DeleteField}(e.e) \mid e.\text{Store}(e)$
ρ	$::=$	$\text{ro} \mid [L] \text{ ro}$
τ	$::=$	$C \mid i \mid \rho$
T	$::=$	$\langle \text{var} = e \rangle$
var	$::=$	a variable name or this
C	$::=$	a class name or Object
i	$::=$	interface name or Empty
fd	$::=$	a field name
md	$::=$	a method name
n	$::=$	numerical value
s, t	$::=$	string value

Figure 4: `ClassicJava`-based grammar of FOREL

Typing and type graph judgments, shown in Figure 5, are of the forms $\Gamma \vdash e : T$ and $E \vdash \tau \xrightarrow{f} \delta$, read “In the type environment Γ , expression e has type T ” and “In the type graph E , type τ has field named f which has type δ ” respectively. The connection between the type graph E and the location of the foreign schema L , supplied as an attribute to the rule `T-ROLNEW`, is established through function `getGraph`: $L \rightarrow E$. Types defined in the type graph E are mapped to the types defined in the type environment Γ through the function `typeMap`: $\tau_E \rightarrow \tau_\Gamma$. Function `matchType`: $s \rightarrow \tau_E$ takes a string constant s defined in Γ and maps it to the set of types τ defined in the type graph E .

Rules `T-OLGET1` and `T-OLGET2` looks up the type of a foreign object described by string variable s , producing its equivalent type in the native program. There are also two typing rules for modifications of foreign components – the insertion rules `T-IORT` which insert fields and parts into foreign programs, and the deletion rule `T-DPRT` which remove fields from foreign components. FOREL follows `ClassicJava`’s lookup rules for method types and method bodies. Due to the lack of space we explain only one rule, `T-OLGET1`. Similar explanations for the other rules can be found in our technical report [18].

The type rule `T-OLGET1` assigns a type to the ROPE expression $e[s]$ where s is the name of a foreign component and e is the grammar nonterminal designating an expression. In the type environment Γ , expression e has type τ . In the type graph E , type τ is mapped to the type ϑ which aggregates an object of type ξ . Type lookup rule `matchType` performs a lookup of type of the foreign object named s and finds the matching type ξ that maps to the type δ in the native component. The type of the expression $e[s]$ is δ which is the consequent of this type rule.

5.4.3 Type Soundness

We can show the type soundness of FOREL through two standard theorems, preservation and progress. Type soundness implies that the language’s type system is well behaved. In a type-safe language like `ClassicJava`, well-typed programs do not get stuck, that is they pass the type checking algorithm successfully or halt with errors. We state the theorems here, leaving the proofs to a companion technical report [18].

THEOREM 1 (PRESERVATION). *If $\varnothing \vdash e : T$, and $e \hookrightarrow e'$, then $\varnothing \vdash e' : T$.*

PROOF. Preservation is proved by induction on the derivation rules. $\square$

THEOREM 2 (PROGRESS). *If $\Gamma \vdash e : T$, then either e is an irreducible value, contains an error subexpression, or else $\exists e'$ such that $e \hookrightarrow e'$.*

PROOF. The proof is by induction on the rules of the type checking. $\square$

6. MAIN STEPS OF STATIC CHECKING

The task of static checking of FOREL programs is greatly simplified when the names of foreign components names are defined as string constants. In this case type checking rules shown in Figure 5 are applied directly to ROPEs to validate their correctness. However, if the names of some foreign components are specified using string expressions, then the values of these expressions may or may not be determined at compile time. The Soot framework is used to recover the names of foreign components using an algorithm for

$$\begin{array}{c}
\text{T-OLGET1} \\
\frac{\Gamma \vdash e : \tau \quad \text{matchType}(s) = \xi \quad \text{typeMap}(\tau) = \vartheta \quad E \vdash \vartheta \xrightarrow{\circ} \xi \quad \text{typeMap}^{-1}(\xi) = \delta}{\Gamma \vdash e[s] : \delta} \\
\\
\text{T-OLGET2} \\
\frac{\Gamma \vdash e : \tau \quad \text{matchType}(s) = f \quad \text{typeMap}(\tau) = \vartheta \quad E \vdash \vartheta \xrightarrow{f} \xi \quad \text{typeMap}^{-1}(\xi) = \delta}{\Gamma \vdash e[s] : \delta} \\
\\
\text{T-ONGET} \\
\frac{\Gamma \vdash e : \tau^* \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash e[n] : \tau} \\
\\
\text{T-INVK} \\
\frac{\Gamma \vdash e : \sigma_1, \dots, \sigma_n \rightarrow \tau \quad \text{for each } i \in \mathbb{N}, \Gamma \vdash p_i : \sigma_i \wedge v_i : \sigma_i}{\Gamma \vdash e(\langle p_1 = v_1 \rangle \dots \langle p_n = v_n \rangle) : \tau} \\
\\
\text{T-IORT} \\
\frac{\Gamma \vdash e : \tau \quad \text{typeMap}(\tau) = \vartheta \quad \text{matchType}(s) = \xi \quad \text{matchType}(t) = \chi \quad \text{typeMap}^{-1}(\xi) = \delta \quad E \not\vdash \vartheta \xrightarrow{\chi} \xi}{\Gamma \vdash e.\text{InsertPart}(s, t) : \delta \quad E \vdash \vartheta \xrightarrow{\chi} \xi} \\
\\
\text{T-DPRT} \\
\frac{\Gamma \vdash e : \tau \quad \text{typeMap}(\tau) = \vartheta \quad \text{matchType}(s) = \xi \quad \text{matchType}(t) = \chi \quad E \vdash \vartheta \xrightarrow{\chi} \xi}{\Gamma \vdash e.\text{DeletePart}(s, t) : \tau \quad E \not\vdash \vartheta \xrightarrow{\chi} \xi} \\
\\
\text{T-ROLNEW} \\
\Gamma \vdash \text{new } [L] \text{ ro} : \text{ro}
\end{array}$$

Figure 5: FOREL typechecking.

string expression analysis, and if this attempt is successful, then the type system is used to perform type checking on the recovered names. However, if the names of foreign components cannot be recovered at compile time, then ROPEs with unknown names are abstracted as adaptive strategies, and type graphs are used to perform the last step of the analysis that is based on the *Traversal Graph Analysis (TGA)* defined in adaptive programming [26].

6.1 Strategies

A strategy $S = (R, \pi, \delta)$, where $R = \{s, d\}$, where s and d are the source and target components of a path in an object graph, and $R \subseteq O$, where O is the set of objects in a type graph, $\pi = \{e, \alpha\}$, where e is a set of fields and α is a set of variables that designate a set of some edges $\alpha \subseteq e$, and $\delta = \{\rightarrow, \dashrightarrow\}$ is a set of transition edges representing complex and simple types respectively. Each node in a strategy S is either the name of some foreign component or a variable designating some foreign components. Using the notation from Section 5.1 we write $\pi(o, o')$ to designate a set of objects $\{o'\}$, such that each object o' of this set is a part of the object o expressed by some edge $e \in \pi$ such that $e(o, o')$. The basic idea of transforming reification statements into strategies is in defining strategy graph edges $a \rightarrow b$ and $a \dashrightarrow b$ for reification statements $x["a"]["b"], x("a")["b"]$

and $x["a"]("b"), x("a")("b")$ respectively. Thus, a strategy is an abstraction of reification statements, and it is also an abstraction of a set of paths in the type graphs.

For example, strategy $\text{CEO} \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \text{amount}$ for reification expression $x["CEO"][\text{strexpl}][\text{strexpl2}]("amount")$ for the type graph shown in Figure 3 designates strategy S , where $s = \text{CEO}$, $d = \text{amount}$, α_1 is a variable designating components of a complex type computed via string expression strexpl , and α_2 is a variable designating a component of a simple type computed via string expression strexpl2 . Computing $\pi_1(\text{CEO}, o')$ we obtain $\{\text{CTO}\}$, and computing $\pi_2(\text{CTO}, o')$ we obtain $\{\text{CEO}, \text{check}\}$.

Each node in a strategy is assigned a distinct sequence number, and nodes are expressed as pairs (i, π) . We introduce functions $\Delta_i : \mathbb{N} \times \mathbb{N} \rightarrow \delta$ and $\Delta_\pi : \pi \times \pi \rightarrow \delta$. Given two sequential natural numbers k and $k+1$, the function Δ_i computes the transition edge between nodes that are assigned these numbers in S , or $\varnothing$ if there is no transition edge. Correspondingly, given two nodes π_q and π_r in some type graph, function Δ_π computes the transition edge between nodes, or $\varnothing$ if there is no transition edge.

6.2 Algorithm

When the values of string expressions in reification statements cannot be computed at compile time, they can be inferred using a TGA-based algorithm *BuildPathTree*. It takes a set of reification statements and type graphs as its inputs and transforms each reification statement into an adaptive strategy with variables replacing string expressions. *BuildPathTree* computes possible values for each variable and generates traversal paths for each strategy. If the algorithm generates no paths, then a type error is produced. If at least one path is generated, then the FOREL compiler report possible errors since values of expressions that compute names of foreign objects may not be in the computed paths.

The TGA-based algorithm *BuildPathTree* for computing valid paths for reification expressions and statements is shown in Algorithm 1. The basic idea of this algorithm is to compute the set $\text{FIRST}(c, c')$ that is defined in Section 5.3. FIRST set is computed using two procedures: *BuildPathTree* and *ComputePath*.

Procedure *BuildPathTree* takes the set R of source and target components in S and set π as input parameters. The output of this procedure is a tree of valid paths in a type graph that satisfy a given strategy. Some of the input components may not make it into the path tree because they do not start any valid paths.

Procedure *BuildPathTree* calls procedure *ComputePath* shown in Algorithm 2, which in turn recursively calls itself. *ComputePath* takes three parameters: a component o that is a potential current node in the path, sequence number i of the node in the strategy S , and the transition edge δ between nodes in S that are assigned two sequential natural numbers i and $i+1$. The goal of this procedure is to color the potential current node o in the path as either *red* or *blue*. When colored *red* object o is considered a dead end on the path in the type graph that does not lead to the designated target nodes. Otherwise, it is colored *blue* and this color is propagated up to the source nodes which are subsequently included in the path tree.

The termination condition for procedure *ComputePath* is defined as the sequence number i being equal to or greater of the number of nodes in the strategy, $|\pi|$, or if there is no transition edge from the current node. When reaching the termination condition we color the current node *blue* and return from the procedure. In the calling procedure we check the color of the node, and if it is *blue*, then we attach this node to its parent node in the path tree.

6.3 Pruning and Generating Paths

```

BuildPathTree(  $R \in S, \pi \in S$  )
for all  $s \in R$  do
    ComputePath( $s, 0, \Delta_i(0,1)$ )
    if color( $s$ ) = red then
        remove  $s$  from  $R$ 
    end if
end for

```

Suppose that program A is connected to program B via ROPE object x . If the program B is connected to program C via its ROPE object y , we can interoperate programs A and C by composing ROPE objects x and y . Reification operators are nonsymmetrical, i.e. reifying types from program I to program J is not the same as the converse. ROPE has the transitive property, i.e. by applying ROPE R_{IJ} to an instance of program I we reify it to J. Then by applying ROPE R_{JK} to program J we reify it to program K. The same result is achieved by applying ROPE R_{IK} directly to program I. This property is useful since it enables the composition of ROPE instances to obtain a new ROPE. Finally, an identity ROPE reifies objects to the same type system, and interestingly, this is useful. Consider a Java program that needs to analyze its own structure. The identity ROPE enables reflective capabilities.

```

ComputePath( $o \in O, i \in \mathbb{N}, \partial \in \delta$ )
if  $i \geq |\pi|$  or  $\partial = \emptyset$  then
     $\text{color}(o) \mapsto \text{blue}$ 
else
    for all  $o' \in \pi_i(o, o')$  do
        if  $\Delta_\pi(o, o') = \partial$  then
             $\text{ComputePath}(o', i + 1, \Delta_i(i, i + 1))$ 
            if  $\text{color}(o') = \text{blue}$  then
                 $\text{AddChildToTree}(o, o')$ 
            end if
        end if
    end for
    if  $\text{children}(o) = \emptyset$  then
         $\text{color}(o) \mapsto \text{red}$ 
    else
         $\text{color}(o) \mapsto \text{blue}$ 
    end if
end if

```

Communication integrity is an important criterion for architectural conformance [27]. In the context of interoperating components it specifies that each component in the implementation may only communicate directly with the programs to which it is connected in the architecture of a polylingual system. Composition of ROPES should not violate communication integrity.

We encode architectural constraints when defining instances of ROPEs in FOREL. These constraints define applications with which a given program can interoperate. An example is a statement that specifies a constraint is `ro ["P"] x = new ro constraints ["Q"]`. This constraint effectively prohibits the program `P` to communicate with other programs but `Q` in a polylingual system, explicitly or implicitly. Our static analysis algorithm ensures that such constraints hold.

The time complexity of `BuildPathTree` algorithm is exponential to the size of the type graph for each reification statement in a FOREL program. Since the algorithm involves the search of all nodes and edges in the type graph that contains cycles for each node in the strategy, its complexity is $O((V + E)^{\max(|\pi|)})$ where V is the number of nodes, E is the number of edges in the type graph, and $\max(|\pi|)$ is the maximum number of nodes in strategies. The operations of putting successors in the table of variables take $O(1)$.

In general, the number of nodes $\max(|\pi|)$ in strategies is much smaller than the number of nodes in type graphs. It is also rare that all graph nodes have to be explored for each node in a strategy. The theoretical limit on computational complexity of `BuildPathTree` algorithm is exponential. However, our experimental evaluation showed that in practice the running time of the algorithm is small and does not exceed one minute for large schemas.

THEOREM 3 (SOUNDNESS). *If the analysis performed by FOREL-based compiler does not report any type errors, then the FOREL program is type safe.*

PROOF. We outline the proof for the soundness theorem by specifying the major steps. As we proved in Section 4.6 the FOREL type system is sound if used on reification statements where names of foreign objects are defined explicitly. The TGA-based analysis whose correctness is proven to determine all possible values that string expressions can take [26]. Finally, we use our type checking algorithm to determine the correctness of the FOREL program using all determined combinations of the values of the name variables and expressions of foreign objects. This concludes the sketch of the proof of this soundness theorem. $\square$

Our prototype implementation included the FOREL compiler which is based on standard Java, and static checking algorithms. We

wrote the FOREL compiler in C++, and we based it on the Pro-Grammar visual environment for building parsers that are platform-independent, programming language-independent and reusable [1]. Our implementation contains less than 3,000 lines of code. Its analysis routines detect errors in foreign components using type checking rules and static analysis algorithms as specified in Sections 5.4.2 and 6.

Since it puts additional burden on programmers to create formal descriptions of foreign components, we automated this process by extracting type graphs and XML schemas from programs automatically using different tools [30, 6]. These tools accept instances of foreign components and output formalisms that can be used by the FOREL compiler. The latter interfaced with these tools and used the extracted type graphs and schemas to performed its functions.

8. EXPERIMENTAL EVALUATION

Our goal in evaluating FOREL is to determine how effective FOREL type checking is when compared with other approaches. Since we know of no other approach that statically checks interoperating components, we compare our approach with manual testing effort. We applied FOREL to a real-world commercial project, to a program written by a student, and to two commercial programs that used large-scale schemas in two different domains. We report the results of these evaluations in this section.

8.1 Archer Analyzer

We applied our approach to the *Archer Analyzer (AA)*, a software package geared for the *Archer 10* optical overlay metrology systems manufactured by California-based KLA-Tencor Corporation [3, 2]. The purpose of optical overlay measurements is to detect and fix misalignments between layers of semiconductor chips that were put on a silicon wafer using microlithography processes.

AA is created as an open system and its interoperating components are hosted by such platforms as *Enterprise JavaBeans (EJB)*, CORBA, and .Net assemblies. The components for AA are created using C++ and different low-level APIs were used for parsing XML and HTML data, invoking Java methods using *Java Native Interface (JNI)*, and interoperating with CORBA and .Net components.

The first release of AA occurred in June, 2001, and its testing continued through the September of 2001. FOREL compiler was not created at that time, and ROPEs were implemented as a library using C++ templates. Bugs were detected during the testing phase manually by a group of test engineers who executed test cases. Found bugs for AA representative programs are shown in Table 1. For example, *DbDataAdapter.cpp* is a program that contains 2,151 lines of code. It interoperates via three ROPEs with other programs and data whose schemas collectively contain 826 types. *DbDataAdapter.cpp* contains 295 ROPE statements that reference 147 foreign components.

Manual testing of the first release uncovered fourteen bugs in this program that can be grouped in seven categories:

- Wrong names of referenced foreign component;
- Wrong operations on foreign components;
- Wrong assumptions about types of foreign components;
- References to wrong parts of schemas;
- Violations of communication integrity;
- Violations of schema constraints.

Last violation requires an additional explanation. Suppose that a schema specifies that some XML object may not contain more than certain number of children of some type. We found that this constraint was frequently violated, and these violations led to dangerous consequences. Instead of programs failing right away, they continued to run and produce incorrect data leading to failures in different components that used this data. This separation of cause and effect both temporally and spatially made it very difficult to localize these bugs and fix it.

Two and half years later the FOREL compiler was completed and applied to the first release of AA. Since we already knew what bugs were discovered during the testing, we were interested to see how our FOREL compiler performs with respect to human effort. While approximately three months of manual regression testing of *DbDataAdapter.cpp* revealed fourteen bugs, FOREL compiler in less than seven seconds discovered 53 bugs, 26 of which were confirmed including those found through testing. Similar results were obtained for other programs thus confirming the viability of our approach.

8.2 papiNet and Metalex

While the results of the evaluation of our approach proved successful for AA, we wanted to evaluate FOREL on systems from different domains, in order to answer the following experimental questions:

- Is the FOREL typechecking practical on industrial large schemas that contain over 1,000 different types?
- Does the FOREL abstraction make it easier to reengineer existing applications?

Methodology. We performed a case study during which we reengineered existing applications for legal and paper supply chain domains. The former application was written for a legal office, and it used *Metalex* schema [20]. *MetaLex* is an open XML standard for the markup of legal sources. The latter application was written by a now defunct startup company for *papiNet*, a global transaction standard for the paper and forest supply chain [21]. The combined source code of both applications was about 30,000 lines of C++ code.

The intention of this study is to manually reengineer legacy systems with interoperating components to evaluate the effort and to see whether our FOREL compiler can find bugs in the reengineered code that were not found in the original legacy code.

Results. The study took about fifty hours for one of the authors of this paper to reengineer the source code to use FOREL. The process involved locating fragments of code that used MSXML parser and replace it with FOREL ROPE statements. The size of the code was reduced by 30% simply by replacing repetitive use of MSXML API with concise ROPE expressions and statements. More complex systems would probably require more time for reengineering with unknown reduction of source code in size, if any.

Type graph generation took a little over eleven seconds for a schema that contains 1,653 types and elements. FOREL type checking algorithm took 7.8 and 2.2 seconds for *papiNet* and *Metalex* applications respectively. For the *papiNet* application 49 bugs were detected, 31 of which were confirmed through manual code inspection, and for the *Metalex* application five bugs were detected two of which were confirmed later.

9. RELATED WORK

Our work builds on a number of existing solutions for polylingual systems and adaptive programming. The most closely related work

C++ Program (A) Archer Analyzer (S) Student Work (R) Reengineered	Size					Analysis Time, sec		Bugs Detected		
	Program, LOC	Schema, No. of Types	No. of ROPE Stmts	No. of Refd Objs	Foreign Progs	Generating, Type Gr.	Semant. Analysis	Testing Phase	With FOREL	Confirmed Errors
DbDataAdapter (A)	2151	826	295	147	3	3.6	2.8	14	53	26
FindRecipeView (A)	754	49	32	50	5	0.2	2.3	5	12	11
RecipeManager (A)	913	826	114	68	17	3.6	5.2	1	9	6
CDBQueueSettings (A)	224	16	35	24	2	0.03	0.8	12	12	12
CXMLEOverlayLotData (A)	453	669	72	181	8	3.1	3.7	3	18	9
DataTestGenerator (S)	338	1653	31	115	12	11.2	1.2	8	13	8
papiNet (R)	7938	1653	313	853	1	11.2	7.8	-	49	31
MetaLex (R)	13128	66	25	19	1	0.3	2.2	-	5	2

Table 1: Experimental results

falls into two categories: interoperable systems and languages using traversal techniques and algorithms.

Component interoperability is a functional aspect that programmers should be able to add to or remove easily from existing software. If such changes are complex, then interoperable software is hard to maintain and evolve. When a programmer creates an interface using IDL s/he can select certain types to declare interface members because they may closely map to desired types in the selected implementation language. For example, if an IDL interface is used to generate C++ wrapper code then IDL types that define this interface are likely to be C++-friendly. When the same IDL interface is used to generate Java wrapper code, programmers replace some IDL types with Java-friendly types. IDL-based approaches have numerous problems described in [23] leading to software that is difficult to maintain and evolve.

An alternative approach to IDL is an automatic mapping between components written in different languages. PolySPIN [9] is a representative of this approach, and it enables programmers to map types directly between different programs. A tool called PolySPINNER analyzes class definitions written in different languages, matches their structure, and generates code that enables objects of matched classes to interoperate seamlessly, i.e. if objects of types *c1* and *c2* exist in different programs, for example, in Java and C++ correspondingly. Both types have to exist in Java and C++ to begin with. After applying PolySPIN approach a call to method *f1* of an object of type *c1* is translated by the generated code into the call to the matched method *f2* of some object of type *c2*. The problem with this approach is that it requires complex matching mechanism to determine isomorphisms between foreign types.

Exu [22] is also an alternative approach to IDL. It enables C++ classes to be accessed from Java classes using JNI. For any C++ class Exu generates a corresponding Java proxy classes and JNI-based interoperability code. This approach is limited as it works only for Java classes that interoperate with C++ classes. In addition, it is difficult to maintain and evolve Exu-based systems because generated isomorphic classes may be changed by programmers.

Navigation traversal paths are integral part of FOREL. Until recently, the automation of traversal of object structures using succinct representations has been unique to Demeter [12]. The connection between reification expressions that navigate to foreign objects and traversal specifications in Demeter is following. The latter is used to generate code that performs the required traversal to the destination object while the former is the code that performs the traversal. In adaptive programming a change of requirements for a

foreign program induces changes of the traversal specifications and the subsequent regeneration of the code that manipulates foreign objects.

XML is a new standard for defining and processing markup languages for the web that uses grammars (also called document type definitions or schemas) to define a markup language for a class of documents. These grammars are akin to class graphs in Demeter. Because of significance of XML as a data interchange standard, an effort is made to integrate XML in the type system of various languages [29, 19, 28].

FOREL language is similar in its functionality to XPath, a language introduced by W3 Consortium to select subsets of XML document elements [11]. XPath expressions are used to describe sets of objects, in the sense that the value of an expression is an unordered collection of objects without duplicates. The way elements are selected in XPath is by navigation, somewhat resembling the way one selects files from an interactive shell, but with a much richer language. XPath was proposed as input to a universal object model walker for arbitrary Java objects [5].

In the context of programming languages, traversals are frequently used as a part of attribute grammars, for traversing abstract syntax trees [13]. Using conventional programming techniques, the details of traversals must be hard-coded in the attribute grammar; this fact makes attribute grammars hard to maintain, say in the case of some modifications in the grammar [24]. In the Eli system [16], this problem is addressed by separating the details of the grammar from the underlying algorithm, using traversal specifications which basically correspond to single edge strategy graphs.

Meta-programming techniques have also been developed for traversals. In [10], a simple kind of traversal (corresponding to a one layer tree graph) is used in a metaprogram; this traversal scans all objects and executes the specified code at the desired targets.

ArchJava is an example of the most recent sophisticated language support for user-defined architectural connectors [8, 7]. ArchJava enables a wide range of connector abstractions, including caches, events, streams, and remote method calls. Developers can describe both the run-time semantics of connectors and the typechecking semantics.

10. CONCLUSION

The contributions of this paper are the following:

- a type system that enables the verification of foreign objects via encapsulated strategies in FOREL;
- an implementation in C++ that uses static analysis and algo-

rithms from adaptive programming;

- a formalization of our FOREL type system using a formal model of ClassicJava and a proof outline of its soundness;
- a novel algorithm for inferring types of foreign objects for FOREL encapsulation strategies; and
- as a result of an empirical evaluation of FOREL we find bugs in a real-world commercial program.

Our experience suggests that FOREL is practical, and its type checking algorithm is efficient. Our implementation of FOREL can be downloaded from a website [4].

11. REFERENCES

- [1] Programmer web site. <http://www.programmer.com/>.
- [2] Archer 10: Archer analyzer automated, real-time overlay metrology analysis.
http://www.klatencor.com/products/archer10/archer_analyzer_tech_factsheet.html, 2002.
- [3] Kla-tencor's software boosts overlay metrology.
<http://www.siliconstrategies.com/story/OEG20020708S0052>, 2002.
- [4] Forel web site. 2005.
- [5] Jaxen. 2005.
- [6] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, October 1999.
- [7] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning with archjava. In *Proc. European Conference on Object-Oriented Programming*, 2002.
- [8] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *International Conference on Software Engineering (ICSE)*, Orlando, Florida, pages 187–196, 2002.
- [9] D. J. Barret. *Polylingual Systems: An Approach To Seamless Interoperability*. PhD thesis, University of Massachusetts at Amherst, 1998.
- [10] R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):20–54, 1984.
- [11] J. Clark and S. DeRose. Xml path language (xpath). 1999.
- [12] Demeter research group.
<http://www.ccs.neu.edu/research/demeter/>, 2005.
- [13] G. Goos and W. Waite. *Compiler Construction*. Springer Verlag, 1984.
- [14] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *International Conference on Software Engineering*, Edinburgh, UK, pages 1–18, 2004.
- [15] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *26th International Conference on Software Engineering (ICSE)*, pages 645–654, 2004.
- [16] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.
- [17] M. Grechanik, D. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *International Conference on Software Engineering*, Edinburgh, Scotland, UK, May 2004.
- [18] M. Grechanik, W. R. Cook, and K. J. Lieberherr. Static checking of object-oriented polylingual systems. Technical Report TR-05-09, Department of Computer Sciences, The University of Texas at Austin, May 2005.
- [19] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. Xj: Integration of xml processing into java. In *Poster at WWW*, 2004.
- [20] <http://www.metalex.nl/pages/welcome.html>. *Metalex*, 2002.
- [21] <http://www.papinet.org.papiNet>, 2002.
- [22] A. Kaplan, J. F. Bubba, and J. C. Wileden. The exu approach to safe, transparent and lightweight interoperability. In *25th International Computer Software and Applications Conference (COMPSAC)*, pages 393–400, 2001.
- [23] A. Kaplan, J. V. E. Ridgway, and J. C. Wileden. Why idls are not ideal. In *9th IEEE International Workshop on Software Specification and Design*, 1998.
- [24] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
- [25] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
- [26] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.
- [27] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
- [28] E. Meijer and W. Schulte. Xml types for c#. In *BillG ThinkWeek Submission*, 2001.
- [29] E. Meijer and W. Schulte. Unifying tables, objects, and documents. In *Proceedings Declarative Programming in the Context of Object-Oriented Languages (DP-COOL)*, 2003.
- [30] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.