RC 21309 (09/30/98)
Computer Science

IBM Research Report

Mockingbird: Flexible Stub Compilation from Pairs of Declarations

Joshua Auerbach, Charles Barton, Mark Chu-Carroll, Mukund Raghavachari

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Abstract

Mockingbird is a prototype tool for developing interlanguage and distributed applications. It compiles stubs from *pairs* of interface declarations, allowing existing data types to be reused on both sides of every interface. Other multilanguage stub compilers *impose* data types on the application, complicating development. Mockingbird supports C/C++, Java, and CORBA IDL, and can be extended to other languages. Its stubs convert types whose structural equivalence would be missed by other tools, because it interacts with the programmer to refine the original declarations. We show that this kind of tool improves programming productivity, and describe, in detail, Mockingbird's design and implementation.

1 Introduction

It is often necessary to interconnect components residing on different machines or written in different languages. Crossing a machine boundary involves adjusting between the representation of information in program variables and its representation in network messages. Crossing a language boundary requires adjustment between different parameter passing conventions and conversion between different type systems. A common goal is to isolate such adjustments in *stub* code that is generated automatically from declarations.

Mockingbird is a prototype tool that compiles each stub from *two* declarations. Each pair of declarations is understood by the programmer to express two different forms of a single abstract interface. Each declaration may state the interface in a different programming language (C, C++, Java, etc.), allowing the full type system of both languages to be used effectively. This approach also permits existing application data types to be reused on both sides of the interface, so that the types used for computation and for communication are the same. Alternatively, one declaration may be in an Interface Definition Language (IDL), enabling interoperation with remote IDL-based non-Mockingbird components while still conferring the benefits of Mockingbird locally.

Other tools leave programmers with a painful choice between language coverage and ease of use. Various distributed programming languages and language extensions [2, 4, 7, 11, 9, 16, 21, 23] allow common data types to be used throughout a project, both for computation and for intercomponent communication, at the expense of limiting a project to a single language. This limitation can be unacceptable when integrating information systems that are already implemented in multiple languages. IDL compilers [13, 15, 17, 18, 19, 22, 24] support multiple languages by generating stubs from IDL. However, IDL compilers also generate programming language data declarations from the IDL that must be used for intercomponent communication. These imposed types are rarely what the programmer would have chosen for computational purposes. The programmer is faced with the error-prone chore of writing program logic to move information between an application's computational data types and the parallel set of imposed communication types. A final category of tools (here called X2Y tools) [14], translate directly from one programming language to another, generating both a stub and a new declaration. There is flexible use of the type system in the source language, but data types are once again imposed for the target language, with the attendant problems. In addition, X2Y tools provide only *ad hoc* pair-wise coverage of languages.

Mockingbird avoids these problems by providing a more intuitive framework to programmers for generating bridges between heterogeneous components while being equal to an IDL compiler in the set of languages it supports. Mockingbird's strength lies in its ability to generate stubs between programmer-supplied declarations with no declaration ever being imposed on the programmer by the tool. Mockingbird requires only that the two declarations be *isomorphic* in its internal type model (which gives a

```
public class Point {
   ... many public methods and constructors
   private float x;
   private float y;
}
public class Line {
   ... many public methods and constructors
   private Point start;
   private Point end;
}
public class PointVector extends java.util.Vector;
// A PointVector contains only Point objects
```

Figure 1: The types used in a sample Java graphical application.

many-to-many mapping, not one-for-one). Given two isomorphic declarations in programming languages, Mockingbird generates an efficient local stub that can be used when the components reside in the same process, and a network-enabled stub for the case where the components are in different processes. If one declaration is an IDL, Mockingbird generates a network-enabled stub obeying the network architecture implied by the IDL.

At present, Mockingbird supports Java, C, C++, and CORBA IDL. Mockingbird and its supported programs run on AIX and Windows 95/NT. The design of Mockingbird enables extensions to additional languages and platforms.

The balance of the paper is organized as follows. In Section 2, we show by example why generating stubs from two declarations is convenient for programmers. We describe the Mockingbird tool in Section 3. In Section 4, we discuss key algorithms and implementation decisions. We describe experiences with using Mockingbird to develop applications in Section 5. In Section 6, we outline possible avenues of future work. We discuss related work in Section 7, and conclude in Section 8.

2 An Example

We now contrast how an interlanguage interoperation problem is solved by an IDL compiler or a typical X2Y tool with the way in which the same problem could be solved in a tool that generates adapters from two declarations. Consider a graphical Java application, developed using the types in Figure 1, that wishes to use the C function whose declaration is shown in Figure 2. The `fitter` function fits a line through a set of points. An array of points is passed as the first parameter to the function, with the array's length as its second parameter. A single point is represented as an array containing the `x` and `y` coordinates. Output parameters for the endpoints of the fitted line are provided through a typical C convention: the caller passes pointers to `points` where the `fitter` will deposit the values. In this application, we wish to fill a (Java) `PointVector` with `Point` objects, pass it to the (C) `fitter` function, and get back a (Java) `Line` object.

Before an IDL compiler can help, we must describe the desired interface in IDL. We will use the OMG CORBA IDL in this example. In Figure 3, we show two possible ways of writing the interface. The interface `CFriendly` is a good match to the C function. The interface `JavaFriendly` is well suited to

```
typedef float point[2];

void fitter(point pts[], int count, point *start, point *end);
```

Figure 2: The `fitter` function in C.

```
interface JavaFriendly {
  struct Point {
    float x;
    float y;
  };
  struct Line {
    Point start;
    Point end;
  };
  typedef sequence<Point> PointVector;
  Line fitter(in PointVector pts);
};
```

(a)

```
interface CFriendly {
  typedef float Point[2];
  typedef sequence<Point> pointseq;
  void fitter(in pointseq pts,
      in long count,
              out Point start,
              out Point end);
};
```

(b)

Figure 3: (a) A Java-Friendly IDL interface (b) A C-Friendly IDL interface.

the existing Java types we want to use.

The translation of the two interfaces into Java is shown in Figure 4. Even `JavaFriendly`, which a Java programmer would find to be the more suitable of the two, has problems. Translation of the IDL gives us *new* `Point` and `Line` classes that do not have the methods needed for application purposes. The fixed translation also dictates that we use the array type `Point[]` rather than our preferred container `PointVector`. If the two IDLs were translated into C, the translation of `CFriendly` would be more appropriate, but even it would not match the original C exactly, requiring additional bridge code to be written.

A typical X2Y tool, such as J2c++ [14], would translate `fitter` into a Java interface similar to the Java translation of `CFriendly`. The difficulty in using the interfaces generated by these tools arises from the introduction of types that are not chosen by a programmer, but derived from types in a foreign language.

Now consider a *two declarations* tool like Mockingbird. In this case, a programmer would write the Java declaration in precisely the manner desired for application purposes (shown in Figure 5). Since the `JavaIdeal` type is chosen by the programmer and not imposed by a tool, it can reuse the application-specific types of Figure 1.

With some hints from the programmer, a *two declarations* tool can generate adapters between `JavaIdeal` and `fitter`, optimized either for local use or for use across a network. The same tool can adapt between these declarations and an appropriate CORBA IDL declaration such as `CFriendly` or `JavaFriendly`, permitting interoperation with CORBA implementations that use only the standard IDL compiler approach. From a single declaration like `JavaIdeal`, the tool may thus give us *several* adapters to other declarations. At runtime, we can use techniques previously studied in Concert [5] to choose an

```
public interface CFriendly
    extends org.omg.CORBA.Object {
    void fitter(float[][] pts,
                int count,
                CFriendlyPackage.PointHolder start,
                CFriendlyPackage.PointHolder end
         );
}

public final class Point {
    ... canned constructors and methods
    public float x;
    public float y;
}
public final class Line {
    ... canned constructors and methods
    public Point start;
    public Point end;
}
public interface JavaFriendly
    extends org.omg.CORBA.Object {
    Line fitter(Point[] pts)
}
```

Figure 4: Translation of IDL of Figure 3 into Java.

```
public interface JavaIdeal {
    Line fitter(PointVector pts)
}
```

Figure 5: Ideal translation of `fitter` into Java.

adapter based on the actual type and location of the object, transparently to the invoker.

3 The Mockingbird Tool

Mockingbird can automate stub compilation for the previous example because the correspondence between `JavaIdeal` and `fitter` is essentially structural. In both interfaces, points and lines are made up of floating point numbers, and both interfaces accept points and return lines. The differences have to do with details about how things are grouped. Mockingbird cannot always discover these details from declarations alone, and therefore, will not recognize the structural similarity without additional assistance from the programmer. In this section, we show how a programmer can provide the necessary information to Mockingbird to enable it to discover the structural correspondence that the programmer desires.[1]

The thesis underlying Mockingbird is that the type systems of modern languages, such as Java, C++, and CORBA IDL, are similar. Mockingbird uses an abstract model called the *Mtype* system to capture these similarities. To reconcile type declarations in different languages, the declarations are first translated into Mtypes. In areas where the mapping from language declarations to Mtypes may be ambiguous, a programmer may affect the translation by providing additional information. Algorithms based on a type-theoretic foundation are then used to determine when a conversion may be performed between types. Mockingbird can generate a two-way converter between types if the corresponding Mtypes are equivalent. If the Mtype of the first type is a subtype of the second, Mockingbird can generate a one-way converter from the first to the second.

The process used by a programmer to generate converters between types is shown in Figure 6. Mockingbird can parse C/C++ declarations, Java class files, CORBA IDL, or *project* files (representing a previously saved session with the tool). Once types have been read into the system, the programmer can affect their translation into Mtypes by attaching annotations to the declarations through an interactive process. For example, a Java `Line` with two fields of type `Point` will be interpreted by Mockingbird to refer to zero, one, or two `Point` objects, since either field may be `null`. Annotations may state that neither field is ever `null` and that neither may introduce an alias. Mockingbird will then conclude that every `Line` must contain two different `Points`. At any point, the programmer can save the current state of the parsed and annotated declarations in a project file for later use.

In order to generate converters, the programmer presents two types to the *Comparer*. This component converts these types into Mtypes, taking into account all annotations on the types, and compares the Mtypes to determine whether they are equivalent or in the subtype relation. The Comparer may use various rules of isomorphism, such as associativity, to find equivalence. For example, associativity implies that if a `Line` contains two `Point` objects, and a `Point` contains two `float` values, then a `Line` might match anything with four `float` values. If the Comparer determines that two types match, it saves information about structural correspondences between the Mtypes for use by the Stub Generator. If, however, a mismatch is found, the programmer can use the annotation-comparison process iteratively until a match is obtained. When the Comparer asserts that two types match, the Stub Generator produces code that may be compiled and linked with applications and a runtime system to provide a bridge between heterogeneous components.

Figure 7 shows a screen shot of a sample use of Mockingbird. The panel on the left lists the set of types currently loaded into the system; the type `fitter` of Figure 2 is currently selected. Its type declaration appears in the upper right, while a diagrammatic representation of the Mtype appears in the

[1] Of course, correspondence between interfaces may be semantic, not just structural. We discuss this issue in Section 6.

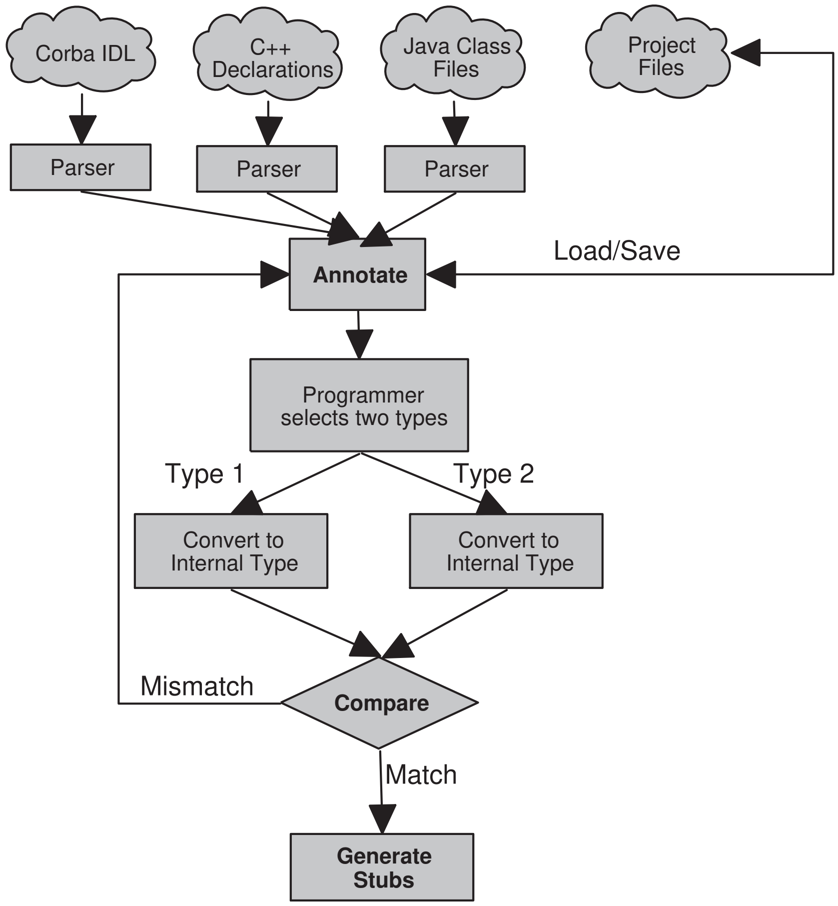


Figure 6: Anatomy of stub generation in Mockingbird.

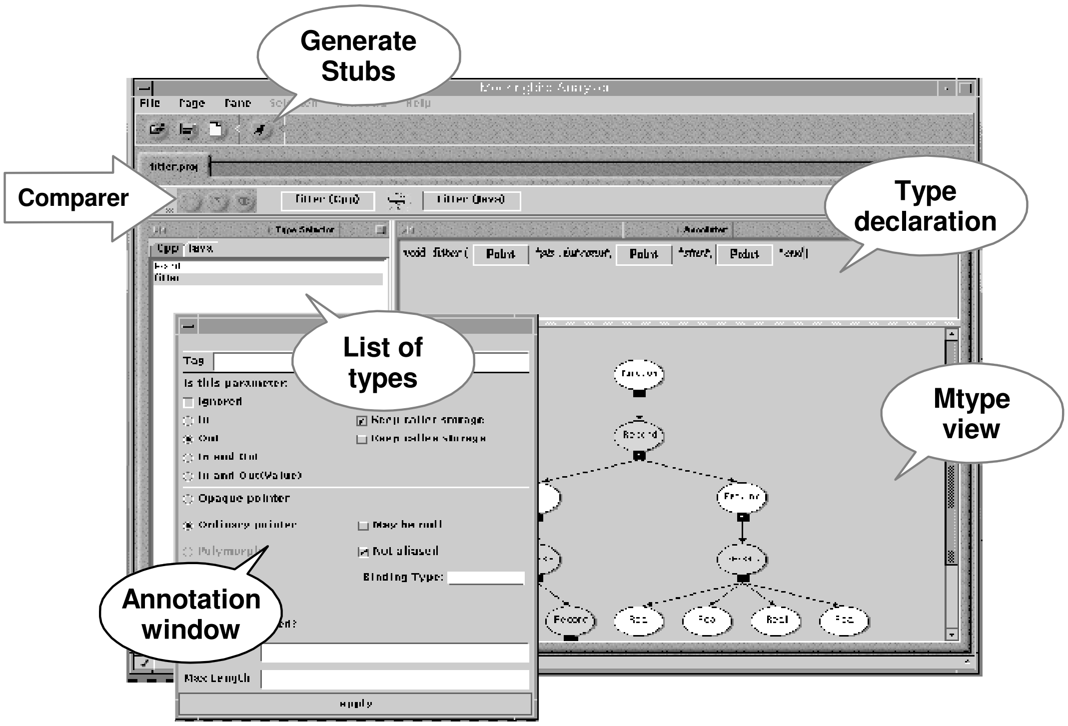


Figure 7: Screen shot of a sample Mockingbird session.

lower right. Selecting any part of the declaration results in the appearance of an annotation window for that type (we discuss the information programmers can express through these annotation windows in the subsequent sections). As annotations are applied by programmers, the Mtype display changes accordingly. The programmer can invoke the Comparer whenever he feels the Mtypes should match. Once the Comparer asserts that two types are interconvertible, the Stub Generator button can be used to emit stubs.

We now give an overview of the Mtypes (summarized in Table 1), focusing on how some representative annotations affect the translation of declarations to Mtypes. We then study how the process shown in Figure 6 would be used to generate stubs for the example of Section 2.

3.1 Primitive Mtypes

The *Primitive Mtypes* are the Integer, Character, Real, and Unit Mtypes. We now discuss how the properties of primitive types in our source languages are modelled by these Mtypes.

Integer types, e.g `short` and `int`, are distinguished by the range of values they support. Accordingly, there is a family of *Integer* Mtypes parameterized by range. For example, a Java `short` is translated into an Integer Mtype, with range $-2^{15} \ldots (2^{15}-1)$. By convention, booleans and enumerations with n elements are translated into Integer Mtypes with range $0 \ldots 1$, and $0 \ldots n-1$, respectively. Two integral types are equivalent if their ranges are equal. Two such types are in a subtype relation if one's range is

Table 1: The Mockingbird internal types (Mtypes).

Mtype	Description
Character	Corresponds to character types, e.g. `char`.
Integer	Corresponds to integral types, e.g. `int`.
Real	Corresponds to floating point types, e.g. `float`.
Unit	Corresponds to `void` or `null` types .
Record	Corresponds to aggregates, e.g `struct`.
Choice	Corresponds to disjoint unions (variants), e.g `union`, and other places where alternatives arise.
Recursive	Corresponds to types defined in terms of themselves.
Port	Used to implement functions, interfaces, etc.

a subset of the other's.

Programmers can annotate source language integer types to specify explicit ranges, overriding defaults based on the language (for Java or CORBA IDL) or the implementation (for C/C++). For example, a Java programmer could annotate a Java `int` to state that it will contain only unsigned values, and a corresponding C `unsigned int` to state that it will not exceed $2^{31} - 1$. The two will then be seen as equivalent.

Similarly, there is a family of *Character* Mtypes parameterized by their *glyph repertoires* (the set of glyphs that are considered representable in the type). C, C++, and Java `char` and `wchar_t` are translated into Character Mtypes by default. Two Character Mtypes are equivalent if their repertoires are the same, and one is a subtype of another if the latter's repertoire includes the former's. For example, a Character Mtype with repertoire `ISO-Latin-1` is a subtype of one with a repertoire `Unicode`. As with integer types, programmers can specify the repertoire by annotation.

C, C++, and Java permit most integral types to contain either integers or characters. Only by programming convention do `char` and `wchar_t` contain characters, while `int` and `short` contain integers. Through annotations, programmers can state which of the two Mtype families is intended.

For floating point types, there is a family of *Real* Mtypes distinguished by their precision and exponent. Again, translation into Real Mtypes assumes certain defaults for precision and exponent, which can be overridden by annotations.

Finally, the *Unit* Mtype is used to model the `void` and `null` types of Java, C/C++.

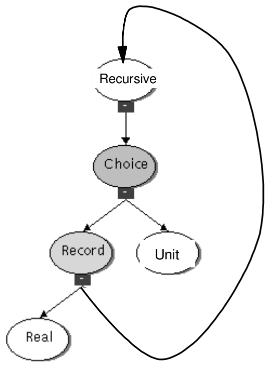
3.2 Compound Mtypes

Most languages have constructs for building ordered aggregates of heterogeneous types. These include the `struct` type in C and CORBA IDL, the `class` type in C++, and the `class` type in Java, if the class types are passed by value. These constructs are modelled by the *Record* Mtype and a list of *child* Mtypes. Thus, the `Point` type in Figure 1, which is intended to be passed by value, becomes a Record with two Real Mtypes as its children, and `Line` becomes a Record with its children being the Mtypes for its `Point` fields. In Section 3.3, we'll see that the input and output parameters of functions and methods are also Record Mtypes.

Arrays of a statically fixed size n, of type τ, are also converted into Record Mtypes. There are n

```
class List {
  float val;
  List next;
}
```

(a)



(b)

Figure 8: (a) A recursive Java list (b) Corresponding Mtype.

children, each identical to the Mtype of τ. Thus, the Java `class` type `Point` (with two `float` fields) has the same Mtype as the C type `point` (defined as `float [2]`).

Languages also have constructs, such as the `union` types in C/C++ and CORBA IDL, that represent choice between alternatives. Pointers or references that may be `null` are choices too, since they may or may not point to something. A reference to an object represents, in a sense, a choice of methods to invoke. A *Choice* Mtype captures this common property of choice as it appears in languages. The list of *child* Mtypes of a Choice Mtype represents the set of alternatives. For pointers and references, the child representing the `null` case is the Unit Mtype and the other is the referred-to type. A programmer may state through annotation that a pointer or reference may never be `null`; in this case the pointer/reference is translated to the Mtype of the referred-to type rather than to a Choice Mtype.

A recursive type declaration is translated into an Mtype graph containing a cycle. A special *Recursive* Mtype node is placed in the cycle, and back-pointers to this node represent self-references. For example, Figure 8a shows the definition of a Java linked list. Figure 8b shows the Mtype corresponding to the Java declaration.

An array type of indefinite size is not modelled by a Record Mtype, because it does not have a fixed number of elements, but an arbitrary number arranged in a total order. We translate all homogeneous and ordered collections of indefinite size into Recursive Mtypes. For example, the C array `float[]`, whose size is not known until runtime, would be represented by the Mtype of Figure 8b, which is how a Java linked list is represented as well. This implies that Mockingbird can generate adapters between these types. Arrays are sometimes implicit in C and C++ (`float[]` and `float *` can both be arrays). Regardless of whether an array is implicit or explicit, annotations may provide either a static length (resulting in a Record Mtype) or a runtime length (resulting in a Recursive Mtype).

3.3 Ports, Functions, and Methods

Mockingbird supports references to functions across language and process boundaries, as well as references to methods of objects and interfaces. We use the *Port* Mtype to model these constructs. A Port Mtype, `port`(τ), represents the addresses to which values of Mtype τ may be sent. The single child of each

Port Mtype node (i.e. τ) corresponds to the Mtype of the value that is to be sent. The Port Mtype can be used to model simple message passing. For example, `port(Integer)` may be the Mtype of queues (possibly remote) to which one can send integers.

Now consider a function, `F` that takes one `int` parameter and returns a `float`. A remote invocation of `F` would consist of an integer and an address to which the callee could send a floating point reply. This invocation would have Mtype `Record(Integer, port(Real))`. A reference to `F` represents the address to which invocations are sent and, therefore, has Mtype `port(Record(Integer, port(Real)))`.

In general, Mockingbird translates a function reference as $\texttt{port}(\tau)$, where $\tau = \texttt{Record}(\mathcal{I},\ \texttt{port}(\mathcal{O}))$, $\mathcal{I}$ is a Record Mtype formed from the Mtypes of the function's input parameters, and $\mathcal{O}$ is a Record Mtype formed from the Mtypes of the function's outputs. By default, all parameters are inputs and the return value is the single output, but any parameter may be annotated as in, out, or in-out.

Finally, Mockingbird views an object passed by reference as a port which accepts invocations of any of its methods. Its Mtype has the form $\texttt{port(Choice(}\tau_1, \ldots, \tau_n\texttt{))}$, where the τ_i are the Mtypes of the various method invocations. We noted earlier that objects can be passed either by reference, as here, or by value (in which case they become Record Mtypes).

3.4 Example

In our example, a programmer would load the C `fitter` (Figure 2) and `JavaIdeal` (Figure 5) declarations, along with supporting declarations (Figure 1). The programmer would then annotate the `start` and `end` parameters of `fitter` (which correspond to the `Line` return value of `JavaIdeal.fitter`) as `out` parameters, and `pts` as an array with length given by `count`.

As mentioned before, the programmer has to indicate that the two `Point` fields of every Java `Line` cannot be `null` or aliased. He must also indicate that `PointVector` can only contain non-`null` `Point` objects. Otherwise, since it inherits from the standard class `Vector`, it would seem that it could contain any object type including `null` references. Mockingbird predefines certain annotations on standard Java classes. For example, `Vector` is treated automatically as an ordered collection of indefinite size.

Eventually, this process terminates with both declarations translating to similar Mtypes. Ignoring some detail, both Mtypes are:

```
port(Record(L, port(Record(Record(Real,Real), Record(Real,Real)))))
```

where `L` is a recursive type representing a list of `Record(Real,Real)`. In general, the process need not terminate in *identical* Mtypes; one might be a subtype of the other, or they may differ by a rule of isomorphism such as the associative rule mentioned earlier. Once the Comparer confirms that these two types are interoperable, a stub can be generated.

4 Implementation

The current implementation of Mockingbird is written mainly in Java, except for portions of the supporting runtime, which are written in C. The C/C++ parser is a modified version of an IBM compiler, the CORBA IDL parser was built from an available grammar, and the Java parser is a simple extractor of type declarations from Java `.class` files. Type declarations are parsed into an internal data structure, called *Stype*, which is an abstract syntax tree representation of the original declaration. It also records all relevant annotations, both defaults and those explicitly applied by the programmer.

As mentioned before, in order to generate converters Mockingbird must determine whether the two selected types are equivalent, or whether one is a subtype of the other. This computation is made non-trivial by the possible presence of recursion in the definitions of these types. There have been various studies of the problem of deciding equivalence or subtyping between recursive types, in particular, the work of Amadio and Cardelli [1]. The algorithm used by the Mockingbird Comparer is based on the one given in [1].

We extend the Amadio-Cardelli algorithm with isomorphism rules to allow for more flexible matching of types. For example, we assume that the Record and Choice Mtypes are associative and commutative. By these rules, `Record(Integer,Record(Real,Character))` and `Record(Character,Real,Integer)` are equivalent. The set of isomorphism rules used by Mockingbird is influenced by the previous work of Bruce *et al.* [10], DiCosmo [12], and Rittri [20], though our rules necessarily differ due to differences in the type system. Future work on these issues is discussed in Section 6.

If the Comparer determines that two types are equivalent or one is a subtype of another, it generates a *coercion plan*. The coercion plan is an internal data structure that incorporates discovered structural correspondences between the two Mtypes, as well as information related to the concrete representation of their values in memory. This coercion plan is used by the stub generator to generate adapters between the two types. Local C to Java interactions are handled by generating JNI code, while distributed interactions may use IIOP or any other wire format. Currently, the stub compiler generates C code directly for the stubs. In the near future, we hope to design an intermediate representation that will allow for compiler optimizations to increase the efficiency of the code.

5 User Experiences

While the Mockingbird prototype is still in a state of evolution, it has already been used by colleagues working on other internal projects. In the near future, we plan to make it available to the research community, or as part of a product.

A substantial trial of Mockingbird involving a research prototype of a new version of the IBM Visual Age C++ Compiler is now underway. In this version, parts of the development environment will be reimplemented in Java, while the compilation engine would still be C++. The interface between the two parts consists of 500 highly inter-related classes with a total of several thousand methods. Mockingbird was first used to build a miniature version of the system with twelve carefully chosen classes representing most of the issues expected to arise with the full set. We have developed a scripting technique that allows annotations, worked out in detail with representative classes, to be applied in batch mode to a much larger set. Mockingbird deals efficiently with the miniature system and no semantic obstacles stand in the way of dealing with the full system. The scalability of Mockingbird's algorithms to the full system is an ongoing investigation.

Mockingbird has also been used in an experiment to develop a Java interface to part of the C++ programming API of Lotus Notes. The full Notes API consists of several thousand methods, of which this limited prototype covered a small, but representative, set of 30 classes. The feasibility of covering the complete API using Mockingbird was demonstrated.

Mockingbird was used in an experiment to support a Java programming framework for synchronous collaboration within electronic commerce applications. Our colleagues were developing *collaborative Java objects* — replicated Java objects capable of coordinated update at multiple sites. The algorithms needed to support these objects had been tuned for concurrency and latency avoidance, and required a message-passing rather than a remote invocation model. Our colleagues declared the 21 message types they needed as Java classes that indirectly incorporated 22 other application-specific Java classes. Mockingbird generated custom "send" and "receive" stubs for these messages, allowing our colleagues to implement their collaborative objects completely in Java and in the style their algorithms suggested. The collaborative Java objects were then used to develop an application in a field trial at a customer site. This project illustrates that Mockingbird is useful even for distributed programming within a single language, and that it supports messaging as well as remote invocation gracefully.

6 Future Work

We are engaged in making the present Mockingbird prototype more robust and complete, in finishing performance and scaling studies, and in pursuing challenging research questions that have been motivated by our experience on this project.

With respect to the prototype, we are building support for certain constructs, such as exceptions, unions, and the CORBA `Any` type (we support a dynamic type construct of our own which is similar to `Any`), whose implementation is currently incomplete. To exploit Mockingbird's subtyping support effectively in C++ and Java, the marshaling runtime should handle the case when a subclass is being substituted for the parent class. At present, it only detects this substitution when objects are passed by reference. Finally, we are working on developing a general framework for stub compilation that can handle the full complexity of the coercion plans that can be generated by the Comparer. At present, we use *ad hoc* techniques that handle most common situations, but which are not easily modified or extended. None of these deficiencies, however, has kept the prototype from being useful to our initial users.

The scaling of Mockingbird to handle interfaces with a large number of classes is still underway and its algorithms are being tuned accordingly. We are also engaged in establishing a realistic set of runtime performance benchmarks to determine whether our two-declarations approach adds any overhead compared to competing technologies (we do not anticipate that it will).

We recognize that Mockingbird handles only structural similarity. In certain cases, a programmer may also wish to provide semantic (non-structural) information. For example, perhaps one line is represented as a slope/intercept pair, and another line, as two points, and the programmer wishes to convert between the two representations. Dealing with such information requires the programmer to provide hand-written conversions which are then integrated with the automated structural ones. We are currently designing mechanisms for composing these programmer-supplied conversions with Mockingbird's structural ones.

The set of annotation panels is always being improved, but we are also considering fundamentally new ways of entering annotation-level information, for example, by drawing rubber bands between parts of types. We are also investigating how best to interact with the programmer concerning mismatches between signatures. Currently, we use a Mtype diagram to depict Mockingbird's view of a type with its current annotations. This approach works well with trained programmers, or when the Mtypes resulting from two different types can be normalized so as to look similar. In Mockingbird, however, types that that do not appear to be similar visually could be equivalent after application of the powerful Amadio-Cardelli and isomorphism rules. Mockingbird, therefore, needs more sophisticated diagnostics that will aid a programmer in isolating mismatches between types. In certain cases, it is desirable to avoid the Mtype display entirely in favor of showing the portions of the original declarations that may have to be annotated. We are exploring a variety of solutions to this problem.

A final area of future work concerns the theoretical underpinnings of Mockingbird. Both the algorithms by which we translate annotated declarations into Mtypes and the algorithms by which we compare

Mtypes raise questions of completeness, correctness, and decidability. Although we justify Mockingbird in this paper (and to our users) based on usefulness, we aim to characterize the supported set of transformations formally. We are developing a semantic model for the Mtype system for which the current set of transformations is sound. Completeness and decidability results exist in the literature for subsets of the rules used in the Comparer, but how to combine these results remains an open question. We have posed the decidability question for Mtype comparison under "intuitively reasonable" sets of isomorphisms slightly richer than the ones we presently employ. One formulation of the decidability problem (not solved) is available in [3].

7 Related Work

Our definition of isomorphism is a multilanguage generalization of *type isomorphisms* [20, 10, 12, 25]. The previous work used type signatures as search keys to find components in libraries. Type isomorphism was used to widen the search by ignoring inessential differences such as argument and tuple order, whether signatures were in curried or uncurried form, or whether a parameterized type was used in place of a specific one. Mockingbird extends these results to a type system including choices and recursive types. Mockingbird also turns the results to a different purpose by generating an adapter automatically. However, by enriching the type system with recursive types, we are unable to claim some of the deeper completeness results as in [10, 12].

The PolySpinner prototype [8] also exploits type isomorphism in a multilanguage context and generates adapters. It employs a generalization of the *relaxed matching* of Zaremski and Wing [25]. Mockingbird automates a wider class of isomorphisms than PolySpinner, particularly ones involving non-abstract constructed types such as unions, arrays, and objects with public instance variables. In addition, Mockingbird adapters are network-enabled and do not require recompilation of components.

Some IDL compilers, such as Flick [13], are more flexible than others. Flick has relaxed the strict one-for-one mapping that characterizes most such tools. The flexibility provided in Flick is, however, more limited. It accepts several IDLs as inputs, but an IDL declaration is still the only declaration used in stub generation. It uses *presentation styles*, mostly motivated by performance considerations, to select the form of the output. This is not as powerful as using two declarations for each stub.

The idea of generating stubs from programming language declarations rather than IDLs has been influenced by the Concert project [4, 5, 6]. We have reused some of the Concert runtime [5] for this project, and the Mtype system is loosely based on Concert Signature Language [6]. Unlike Concert, Mockingbird does not require any programming language extensions or modifications to compilers. Furthermore, the type isomorphisms explored in Mockingbird are much deeper.

8 Conclusion

The development of applications that use heterogeneous components or bridge existing software systems is a complex and error-prone task. We have developed a prototype tool, Mockingbird, that eases the burden on programmers by allowing them to use *existing* type declarations when generating adapters between interfaces. This is in contrast to single declaration tools, for example, most IDLs, that *impose* types on at least one side of the interface, forcing programmers to integrate these types into their existing application structure. Mockingbird maintains the same coverage of languages as IDL compilers, and uses powerful isomorphism rules and algorithms to allow for flexible matching of types. We have shown through a detailed description of the Mockingbird prototype, and through examples, that such a tool is feasible and useful. Initial experiences with using Mockingbird on other projects has confirmed our belief that compiling stubs from pairs of declarations is more natural to programmers. We are currently exploring several strategies for making Mockingbird more intuitive and efficient.

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[2] Gregory R. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[3] J. Auerbach, C. M. Barton, and M. Raghavachari. Type isomorphisms with recursive types. Technical Report RC21247, IBM T. J. Watson Research Center, 1998.

[4] Joshua S. Auerbach, Arthur P. Goldberg, Germán S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. Concert/C: A language for distributed programming. In *Winter 1994 USENIX Conference*, January 1994.

[5] Joshua S. Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *The 14th International Conference on Distributed Computing Systems*, June 1994.

[6] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Interface Definition Languages*, January 1994.

[7] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1991.

[8] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Fourth Symposium on the Foundations of Software Engineering (FOSE)*, October 1996.

[9] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[10] K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[11] K. M. Chandy and C. Kesselman. CC++: A declarative, concurrent object oriented programming language. Technical Report CS-TR-92-01, California Institute of Technology, 1992.

[12] Roberto Di Cosmo. *Isomorphisms of Types: From λ-calculus to Information Retrieval and Language Design*. Birkhauser, 1995.

[13] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM, June 1997.

[14] M. Hubbard and A. Schade. J2c++. *http://www.alphaworks.ibm.com*, 1997.

[15] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-87-150, CS Department, CMU, September 1986.

[16] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3), March 1988.

[17] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*. Microsoft Corporation, 1995.

[18] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1995.

[19] Open Software Foundation, Cambridge, Mass. *OSF DCE Release 1.0 Developer's Kit Documentation Set*, February 1991.

[20] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[21] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, January 1991.

[22] Sun Microsystems. *SUN Network Programming*, 1988.

[23] Inc. Sun Microsystems. Java remote method invocation specification. Technical report, Sun Microsystems, Inc., 1996.

[24] The Xerox Corporation. *Courier: The Remote Procedure Call Protocol*, December 1981. Technical Report XSIS 038112.

[25] A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering Methodology (TOSEM)*, April 1995.