RC 20718 (02/05/97)
Computer Science

IBM Research Report

The Mockingbird System: A Compiler-based Approach to Maximally Interoperable Distributed Programming

Joshua Auerbach, Mark C. Chu-Carroll

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

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

1 Introduction

Programmers wishing to write distributed applications face a painful choice. If they use the best available distributed programming languages, programming will be relatively easy, but they will not be able to publish interfaces for use from other languages. If they use a technology based on a separate interface definition language (IDL), they get coverage of multiple languages, but, for reasons detailed below, the programming task becomes much harder.

The Mockingbird system is our solution to this problem. It is a set of experimental tools that enables interoperation across languages while avoiding many of the complexities of IDL-based programming. Programmers prepare for multilanguage interoperation by enhancing existing programming language declarations with annotations rather than by using IDL, yielding advantages that we will demonstrate by example. This improvement is enabled by our use of a more appropriate criterion of correctness for the translation of values between different type systems, one that is less restrictive than the criterion used by previous systems. Empowered by this change, our tools use compiler technology to generate automated communication support at a higher level of abstraction than previous tools. For reasons that are intrinsic to our approach, Mockingbird programs are able to interoperate with preexisting programs and do not require everyone to adopt Mockingbird techniques.

In section 2 we show, through an example, why IDL-based tools are harder to use than single-language solutions. In section 3, we use the same example to show how a Mockingbird programmer is able to get nearly the best of both worlds. In section 4, we introduce *interconvertibility*, which is Mockingbird's criterion of correctness for interlanguage value transmission, and contrast it with the *type preservation* criterion used by other systems. In section 5 we introduce Mockingbird Signature Language (MockSL), the internal notation that (1) captures the interconvertibility property and (2) retains other information needed by the Mockingbird emitter to generate communication stubs, and we show how the interfaces in our example are translated into MockSL. In section 6, we briefly detail how communication stubs are generated from MockSL. In section 7 we describe the present status of the Mockingbird prototype, suggest our future directions, summarize, and conclude.

2 Distributed Programming Languages versus IDL-based Programming

All recent tools to support distributed programming have a *marshaling subsystem*, consisting of (1) compile-time facilities that analyze interface declarations to generate *marshaling stubs*, and (2) runtime facilities to support those stubs. Marshaling stubs gather values in program variables to form messages, and parse incoming messages to update program variables.

Tools that employ marshaling stubs fall into two categories. In distributed programming languages (PLs), the programmer gets automated marshaling "for free" when a compiler for the language analyzes the type and interface definitions of the language. There are many research distributed languages, such as Emerald [7], Argus [10], Hermes [14], and SR [2]; see [6]. There are also many proposed distributed extensions to pre-existing languages, such as Java RMI [1], Compositional C++ [8], and Concert/C [3]. In all these solutions, a single set of declaration terms is used for all data and interfaces, both those used in local computation and those used for remote interoperation. It is this feature that makes these languages easy to use. However, protocol messages exchanged between processes encode the language's type system, which makes it hard to publish interfaces for use in different languages.

With IDL-based tools, the programmer declares interfaces and data to be used for remote interoperation in a specialized language with its own type system. An IDL compiler translates these declarations into equivalent declarations in each PL and also generates the marshaling stubs. IDL-based tools include research results like Courier [17] and Matchmaker [9], as well as the competing proposals of various companies and consortia, such as SUN ONC [15], OSF DCE [13], Microsoft's Distributed COM [11], and OMG's CORBA [12]. Because the IDL's type system has some translation into each language, multi-language interoperation is achieved. The declarations imposed by IDL, however, don't use the full expressiveness of the PL and may not be convenient for local computation. If the programmer adopts the IDL-derived types throughout the program, the program's

```
public class Point {
   private float x;
   private float y;
// methods
}

public class PointVector extends Vector { }

public class Line {
   private Point start;
   private Point end;
// methods
}

interface LineFitter extends Remote.proxy {
   public Line fitline(PointVector points);
} // An implementation of LineFitter will run in the server // to be remotely invoked by the client
```

Figure 1: Line Fitting in Distributed Java

overall style is compromised. If he isolates the use of these types to the communicating parts of the program, he ends up moving values between a set of computation types and a set of IDL-derived communication types.

We illustrate this contrast with an example. The problem involves a remote computation in which a set of points is transmitted and a line, fitted to those points, is returned. We show this problem first in figure 1, which depicts how it might be solved using a distributed version of Java similar to Sun's Java RMI proposal [1].

Notice that the data types `Point` and `Line` have public methods (not shown) that are important for local computation but not relevant when points and lines are transmitted. Notice also that the `PointVector`, used to collect multiple points, was implemented trivially and conveniently in terms of Java's standard `Vector` class. Implicitly, a `PointVector` contains only `Point` objects, but nothing in this declaration states this constraint explicitly. Finally, notice the clause `extends Remote.proxy` on the interface declaration. This is an indication (some form of which will be necessary in any practical distributed Java) that method invocations made via this interface may be remote.

We assert without further argument that writing this was easy. Writing the same thing in a distributed form of C++, as shown in figure 2, would also be easy. Our C++ version differs from the Java version more than it has to in order to illustrate some key points later on.

We've assumed in figure 2 that the computational parts of this application prefer a representation of collections of points as paired arrays of x and y coordinates rather than as collections of distinct `Point` objects. This preference is exhibited in the concrete form taken by the `PointVector` class. A natural formulation in C++ for an interface that has only one operation is as a function rather than a class, as we have shown. In addition, C++ often uses pointers to uninitialized variables as a way to obtain outputs, which we have also done in this example. The `[out]` annotation, shown in Concert/C [3] syntax, is an indication, (some form of which will be needed in any practical distributed C++) that these parameters are transmitted from callee to caller if the function is invoked remotely.

Of course, the Java and C++ implementations, although individually easy to write, do not interoperate, and programmers working with these two distributed languages have no way to create interoperable declarations. So, now we will look at how this interface appears in the OMG CORBA IDL. We show the IDL declaration in figure 3.

The declaration is just as easy to write in CORBA IDL as in C++ or Java. However, the types that are declared here are not the same as the ones that were judged by the programmer to be the most appropriate for local computation. CORBA's single heterogeneous aggregate constructor, `struct`,

```
class Point {
   float x;
   float y;
public:  // methods
};

class PointVector {
   int len;
   float * xs;
   float * ys;
public:  // methods
};

class Line {
   Point start;
   Point end;
public:  // methods
};

typedef void fitline_func(
      PointVector * points,
      [out] Line ** fit);
```

Figure 2: Line Fitting in Distributed C++

```
struct CommPoint {
   float x;
   float y;
};

struct CommLine {
   CommPoint x;
   CommPoint y;
};

typedef sequence<CommPoint> CommPointVector;
interface LineFitter {
   CommLine fitline(in CommPointVector pts);
};
```

Figure 3: Line Fitting in CORBA IDL

```
class CommPointVector {
public:
   CommPointVector(); // default constructor
   CommPointVector(Ulong max); // maximum constructor
   CommPointVector(Ulong max, Ulong length, CommPoint * value,
         Boolean release = FALSE);
   CommPointVector(const CommPointVector&);
   ~CommPointVector();
   CommPointVector& operator=(const CommPointVector&);
   Ulong maximum() const;
   void length(Ulong);
   Ulong length() const;
   CommPoint& operator[](Ulong index);
   const CommPoint& operator[](Ulong index) const;
};
```

Figure 4: The C++ Realization of CORBA's sequence constructor

defines only data, not methods. So, `CommPoint` and `CommLine` are not the same `Point` and `Line` that programmers wanted to use for local computation. CORBA's only dynamic homogeneous aggregate constructor, `sequence`, induces a specific, single representation in each PL. Figure 4 shows how the IDL compiler is supposed to translate `CommPointVector` into C++ according to the CORBA [12] standard. Translations of some IDL types into Java are even less natural (in proposals such as [16]) since Java lacks an obvious mapping for IDL constructs such as `union`s and `out` parameters.

We show all the detail not only because it is complex, but primarily because it is so semantically specific and unrelated to the needs of local computation. A `CommPointVector` has exactly the behavior that CORBA dictates, not the application-specific behavior that the C++ programmer wanted (such as the ability to extract the x and y values as separate arrays).

3 The Mockingbird Approach

We aim to allow programming as in the C++ and Java examples, not imposing arbitrary abstractions from a separate IDL, but permitting the C++ and Java interfaces to interoperate with each other. For the line-fitting example and many others, this goal is achievable using Mockingbird techniques. The cost is that each programmer must add slightly more information to the *existing* declarations. This can be done without perturbing the semantics of these declarations as they affect local computation.

The Mockingbird Java programmer must provide the information that `PointVector` will contain `Point` objects only instead of arbitrary objects. The altered declaration is as follows.

```
public class PointVector extends Vector [| elementtype(Point) |] {};
```

Note that this assertion about `PointVector` is simply an accurate refinement of its specification, not a change to its actual semantics.

The Mockingbird C++ programmer must provide some more information about the `PointVector` class.

```
class PointVector {
  int len;
  float * [| required, length(len) |] xs;
  float * [| required, length(len) |] ys;
public: // methods
};
```

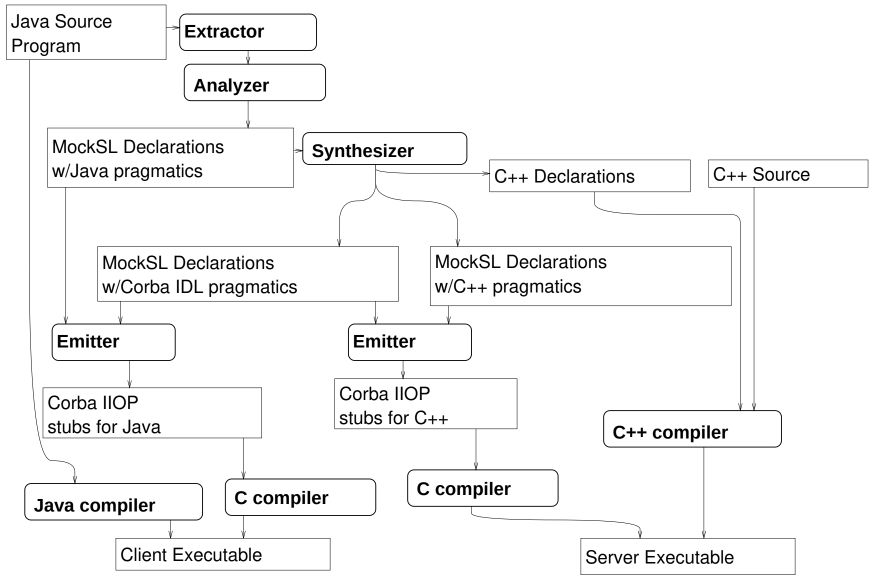


Figure 5: An example of the Mockingbird compilation process

The `required` annotation says that the pointer it follows can never be NULL. The `length` annotation says that the pointer addresses an array whose length is to be found in an integer value.

Note that Java and C++ had already been extended in order to support distribution (the `extends Remote.proxy` convention for Java and the `[out]` annotation for C++). These additional extensions are modest in that context. In Mockingbird, we use lexically unambiguous delimiters `[| |]` to mark annotations. This allows them to be filtered out prior to compilation by a simple lexical process. Stylized comments might have been used as well, although we find the present syntax to be more readable.

The stubs necessary to produce interoperation between the Java and C++ programs shown are otherwise generated automatically by Mockingbird. In addition, Mockingbird implementations of those two programs will also interoperate with a standard OMG CORBA program using the IDL in the illustration and communicating according to CORBA standards.

The actual Mockingbird toolset is organized into the elements that appear in figure 5. As shown, the Mockingbird system is built from the composition of four types of tools. *Extractors* extract the relevant datatype definitions from a source language (a programming language or an IDL). *Analyzers* generate our intermediate notation, MockSL, from the output of extractors. *Synthesizers* generate PL or IDL declarations from MockSL. The *emitter* generates marshaling stubs from MockSL.

Figure 5 illustrates a possible real-world development scenario based on our earlier examples. The Java programmer first wrote the interface of figure 1. He added the additional Mockingbird annotation we discussed, and fed the result through the Mockingbird Java extractor and the Mockingbird analyzer. The result was an internal form called MockSL (which we discuss below). The C++ programmer then interacted with a C++ synthesizer to help her finalize her preferred interface, that of figure 2. The synthesizer determined that this interface would be interoperable with the Java version. The result of this interaction was another MockSL specification.

The programmers could have stopped there, and simply asked the emitter to build the stubs so that their programs could interoperate. However, recognizing that not all programmers are Mockingbird programmers, they decided it would be prudent to have a CORBA IDL version of the same interface so that CORBA clients and servers could interoperate with theirs. They thus interacted with a CORBA IDL synthesizer to determine that the IDL specification of figure 3 would interoperate with the other two. The MockSL from this step captures not only the IDL but also the detailed formats of the messages (in the CORBA-mandated IIOP protocol) that must be transmitted. The emitter took the appropriate *pairs* of MockSL specifications and generated stubs for Java to/from

IIOP and for C++ to/from IIOP. The two programs not only communicate, but they do so in a way that allows CORBA programs to communicate with them as well.

In a slightly different version of the scenario, the CORBA IDL may have pre-existed and so the Mockingbird programmers simply fed it through the appropriate extractor and analyzer to produce MockSL. They then used Mockingbird synthesizers to find their preferred mappings into C++ and Java.

In the remainder of the paper we will show how Mockingbird is able to achieve this ambitious goal of flexible interoperation. In the process we will also suggest the limitations of the approach.

4 Interconvertibility: A New Criterion of Correctness

Previous distributed languages and IDL-based tools used *type and value preservation* as their criterion of correctness for their marshaling subsystems. If a value of type T leaves process A and arrives in process B, it should still be of type T and should have the same value as well. This criterion of correctness is both easy to understand and easy for marshaling subsystems to achieve but it is the main obstacle to providing tools that support multiple languages while remaining easy to use. If two processes employ different type systems, then it is clear that marshaling of some types will be impossible, because type T might not exist in the language of process B. This is why the IDL-based systems end up defining yet another type system, and a fixed and narrow set of translations between it and each PL. These fixed and narrow translations guarantee that type preservation will always be achievable.

The Mockingbird approach to this problem begins with the observation that type preservation is not a necessary condition of interoperation. In the end, no matter what notation is used to declare an interface, all that the interface can ultimately dictate is what information will flow across it. Neither side of the interface should be capable of observing what happens on the other side. Therefore, neither partner can know whether or not the exact types of values have been preserved. There are, of course, numerous semantic constraints on the interface that must be met if the overall system is to do anything useful, but even type preservation doesn't guarantee that all such constraints are met. In a multilanguage environment, the additional strictness of type preservation may be simply irrelevant to meeting the real application-level constraints.

Instead of the usual definition of correctness, Mockingbird substitutes a more liberal definition in terms of *interconvertibility.* Two types are interconvertible if there is an invertible mapping between values of one type and values of another. The Java and C++ definitions of `PointVector` are interconvertible in this sense. Many other possible representations of "a homogeneous collection of pairs of floating point numbers" will also be interconvertible with these; the collection could be formed as a linked list, etc. The system, however, may need to perform fairly extensive conversions as the values cross the interface.

In addition to interconvertibility for isolated data values, this concept extends readily to actions like remote method call or remote procedure call. The act of invoking the `fitline` method of the Java interface `LineFitter` produces a pair of messages (request and response) that are interconvertible with the messages produced by invoking the `fitline_func` interface in C++, which are in turn interconvertible with a pair of request and response messages that were sent and received explicitly. So, for example, a Java variable of type `LineFitter` is interconvertible with a C++ variable of type `fitline_func *`.

Mockingbird's marshaling system guarantees only that stubs perform invertible value mappings, not that they preserve the exact type. It guarantees that actions like method invocations communicate interconvertible messages, not that there is necessarily a similar method being invoked in the remote process. By using these more liberal rules, we are able to provide a better tradeoff between the needs of interoperability and the needs of individual languages.

In a distributed system, there are actually *two* opportunities to convert most values, because two processes that communicate are typically separated by a protocol system defined in terms of messages. The stub at one end of the wire converts from the type system of process A to the message's type system, and the stub at the other end converts from the message's type system to a type in process B. Since interconvertibility is transitive, the types used inside A and B are interconvertible if each is interconvertible with the message's type. However, the separation provides expanded opportunities for exploiting a relaxed notion of marshaling correctness. Even if process A employs an older marshaling technology that believes it is preserving the exact type for reconstruction in process B, process B may use a newer, Mockingbird marshaler that constructs some other, interconvertible type. Process A need not be changed, meaning that a Mockingbird-assisted programmer can communicate with whatever is out there.

Because two conversions typically intervene between processes, we are also able to make an important guarantee. If A and B use type systems that both define a type T, then type T will actually be preserved as long as A and B use the same mapping function between T and the interconvertible message type ("interconvertible" implies "invertible"). Consequently, Mockingbird loses nothing by using the weaker criterion in the homogeneous, single-language case, even though it gains a lot when attempting multi-language interoperation.

5 Mockingbird Signature Language (MockSL)

The Mockingbird analyzer translates programming language declarations of types and signatures into an intermediate representation called MockSL. MockSL was designed to do two things. (1) It describes the set of types with which a given type is interconvertible. (2) It preserves all of the concrete detail necessary to later generate conversion stubs between interconvertible types. The first of these goals must be approximated conservatively, or else both goals will be unachievable. If "converting" were defined in the most general way possible, conversion would be a Turing-complete problem, and automated stub generation would be impossible.

We've defined MockSL to capture a computable and conservative subset of interconvertibility by applying the following principles.

1. We define MockSL as a type system in its own right. We call its types *interconvertibility types*.

2. We define the interconvertiblity types of MockSL so they are differentiated by their *value sets* only, ignoring any characterizing operations not strictly necessary to define those sets. Mapping PL or IDL types into interconvertibility types is thus, typically, a *widening*: there are many different PL or IDL types corresponding to each interconvertibility type.

3. The translation of PL and IDL types into MockSL attempts to put type definitions into a canonical form, so that as much as possible, interconvertible PL and IDL types yield the same interconvertibility types in MockSL.

With these properties satisfied, the interconvertibility property over MockSL types reduces to structural equivalence in the MockSL type system. More intuitively, two types are interconvertible according to MockSL if, when reduced to sets of primitive values, there is a one to one correspondance between the elements of the two sets.

Interconvertibility types, as defined, contain *less* information than the original PL and IDL types from which they were generated. Some of this information, such as the characterizing operations of an object-oriented class, can be discarded, since it will not be used in stub generation. However, other kinds of information, particularly information describing the layout of structures in memory, must be preserved in order to generate stubs. In MockSL, this information is preserved in the form of *pragmatic annotations*, which are applied to each interconvertibility type to describe its representation in a particular programming language or communication protocol.

The structure of MockSL representations, and the particular type constructors that are used to generate representations, are presented in figure 6. The interconvertibility types of `MockSL` reflect the semantics of the PLs and IDLs that they are designed to support. There are `Integer`, `Real`, and `Character` primitive types, and an `Any` type. An `Abstract` type encodes programmer-defined types that Mockingbird is expected to leave alone. A `Capability` type is used to model

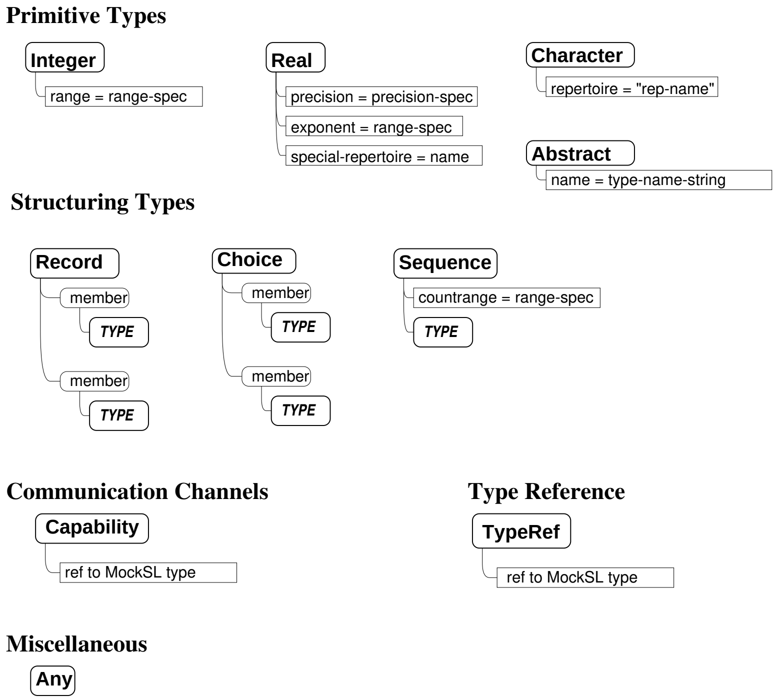


Figure 6: MockSL type constructors

remote references. A `TypeRef` type is used to express recursion or type inclusion when one type is completely included in the definition of another. A `Record` type constructs heterogeneous aggregates and a `Sequence` type constructs homogeneous aggregates. Finally, a `Choice` type constructs alternative types (as in the `variant` or `union` types of Pascal and C).

Because the MockSL type system is composed of familiar concepts, it might seem that our conservative definition of interconvertibility is just type preservation by a new name. However, our primitive types are defined so as to ignore any detail not essential to defining their value set. Similarly, our constructors capture only the issues of quantity, choice, and ordering that are strictly necessary to put constituents in one-to-one correpondence. There is no pointer or local reference type in MockSL, and so our analysis of pointers always reduces them to some other type. Sometimes this requires annotation (like the `required` annotation in our example).

In analyzing compound applications of type constructors Mockingbird analyzers also apply various canonicalizing rewritings to reduce the incidence of obviously interconvertible cases that are not structurally equivalent. For example, if a structure is nested in another structure, the analysis would absorb members of the inner structure into the outer one. If one declaration calls for there to be N values of type S and N values of type T in a communication, and another calls for there to be N tuples where each tuple has an S member and a T member, Mockingbird analyzes both into the *same* internal form (resembling the second declaration rather than the first). An example of this transformation is illustrated in figure 7 and is also implicit in the fact that Mockingbird finds the C++ `PointVector` and the Java `PointVector` of our earlier examples to be interconvertible. If a declaration employs recursion in such a fashion that a homogeneous aggregate would be equivalent for our purposes, then it would be rewritten as a homogeneous aggregate. Thus, a simple linked list will become a `Sequence` when analyzed by Mockingbird. Finally, by representing types as value sets, we ignore many details of declaration order and nonessential names in order to maximize interconvertibility.

Since all of the canonicalizing transformations that we employ are invertible, they are manifestly correct implementations of the interconvertibility criterion. However, we have not attempted to prove any properties of the *set* of transformations taken as a whole. Rather, the set has been chosen

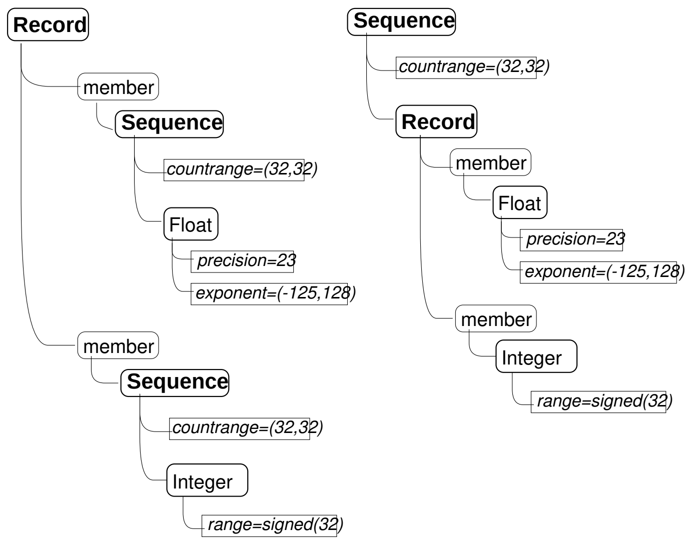


Figure 7: An example of a canonicalizing rewriting during Mockingbird analysis

heuristically to maximize interconvertibility over data structures that we've observed to be typical or important. As long as the interconvertibility property used in the implementation of Mockingbird remains *conservative* and *decidable*, we can improve coverage gradually by adding more and more such transformations.

5.1 Representing Types in MockSL

We will now describe how MockSL represents types and signatures, by illustrating its use on the examples presented earlier. The MockSL representation of the Mockingbird C++ linefitter example, without pragmatic annotations, is shown figure 8. This example illustrates many of the features of MockSLs representations of types. To begin, each type is described with a MockSL tree. These trees are rooted with *type constructors*, such as `Real`, `Record`, etc; and have children specifying the details of the type.

These type constructors can be divided into a number of families, as shown in figure 6 and discussed in the previous section. In this example, we can see examples of primitive types, structuring types, type references, and the `Capability` type. The only primitive type illustrated is a `Real`, but this can be taken as an example of how primitive types are represented in MockSL. The `Real` type is detailed with information specifying its specific properties for a particular target compiler. These details completely qualify the exact range of values which can be represented by an instance of the type. Other primitive types are dealt with similarly.

The example also shows two structuring types, `Record`, and `Sequence`. `Record`s are used to build each of the types presented in the original example. Each record consists of an unordered collection of `member` trees, each of which contains the MockSL type of the particular member of the structure. For instance, in the representation of the `Point` type, the record has two members, for each of the two fields of the original `Point` type. Each of the members contains a `Real` type, describing the allowable range of values of a `float`.

`Sequence` is used to represent the `PointVector` type from the original program. The `Sequence` type has two components: a specification of its length, and a specification of the element type of the sequence. In this case, the element type is a pair of `Real` values. This demonstrates one of the canonicalizing transformations used by MockSL. In the original example, there are two arrays of floating point values, which (by virtue of length annotations in the original declarations) we know must be of the same length. The direct representation of that structure in MockSL would be as a `Record` containing two `Sequence`s. However, since they share an identical length constraint,

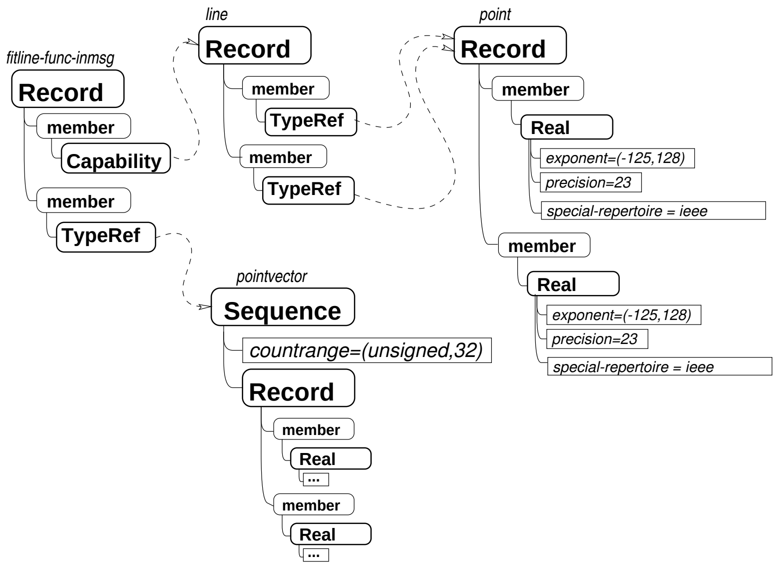


Figure 8: The MockSL representation of the C++ example in figure 2

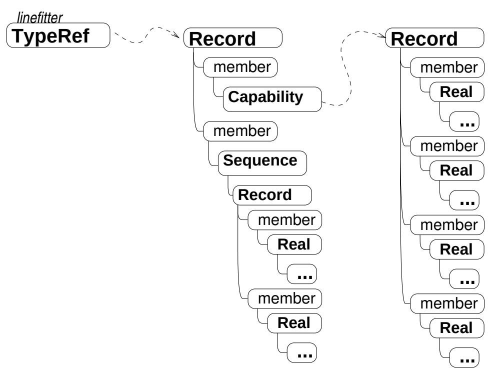


Figure 9: MockSL form of the CORBA fitline example

MockSL merges the two `Sequence`s into a single `Sequence` of pairs.

The `fitline_func_inmsg` type arises as follows. In MockSL, we represent remote procedure calls as a pair of messages. The *in-message* contains the input parameters to the call and a `Capability`, which is used to represent the caller's expectation of receiving a particular type of reply. The reply itself is called the *out-message* (in this example, its type is `line`). Because the *in-message* for `fitline_func` included a reply `Capability` in addition to the input parameters, no type in the C++ declaration exactly corresponded to it, so Mockingbird generated a `Record` type for it.

5.2 Interconvertibility

In figure 9, we illustrate the meaningful sections of the MockSL that represents the CORBA IDL version of our example. Once all of the `TypeRef` nodes are replaced with the structures that they reference, these two trees are *structurally equivalent*, and therefore, are interconvertible under Mockingbird. The MockSL from the Java version is not explicitly illustrated in this paper, because it would look the same as the C++ version at this level of detail (the pragmatic annotations are different). It is also structurally equivalent, and therefore interconvertible.

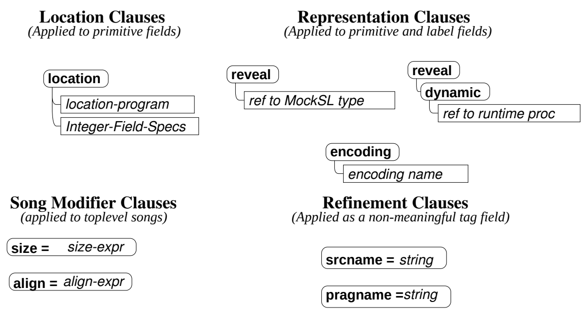


Figure 10: The pragmatic annotations of MockSL

5.3 Pragmatic Annotations

The meaningful parts of MockSL, as presented and illustrated above, allow Mockingbird to determine when two signatures are interconvertible. However, they do not contain enough information to allow the emitter to generate conversion stubs, since they lack all details of how the particular value-set of a MockSL type is represented as a PL or IDL type. This information is provided in the form of *pragmatic annotations* attached to type constructor nodes. The basic set of pragmatic annotations is illustrated in figure 10. We will now briefly illustrate how pragmatic annotations are used to preserve representation details.

We again return to our earlier C++ example to illustrate the pragmatic annotations generated for some of the MockSL types. Mockingbird always analyzes for a particular target machine and compiler. For this illustration we chose the IBM RS/6000 workstation running AIX, and the xlC compiler. Selected portions of the example are shown in figure 11.

Pragmatic annotations can be divided into two classes: regular annotations, which can be attached to any type constructor, and *leaf* annotations, which are disallowed for the structuring types `Record`, `Choice`, and `Sequence`. Regular annotations contain information about the alignment constraints of the entire type on a particular architecture, the overall size of a type, and information to relate the type back to the original type definition from which it was generated.

The leaf annotations include location clauses and representation clauses. Location clauses specify a procedure that can be used to locate each part of a structuring type. For demarshaling, location clauses also contain information needed to allocate storage and construct objects. The syntax of the location mini-language used in the body of a location clause is specified in figure 12. Representation clauses contain data that is used to emit the details of marshaling code, as discussed in the next section.

As can be seen from figure 11, location clauses on leaf nodes reflect the semantics of enclosing structuring nodes. For example, a location clause for the elements of a sequence uses a `next` operator to form a loop visiting each element in turn. Location clauses are placed on leaf nodes rather than on the structuring nodes so that the canonicalizing transformations (which rewrite and reorder the structuring nodes while preserving the leaf nodes) will leave them intact. The `offset`, `indirect`, and `next` operators correctly describe the concrete representation as being a pair of arrays even though the interconvertibility type is a sequence of records.

6 Generating Code from MockSL

Mockingbird code generation works from two MockSL specifications. The two specifications must be for the same interconvertibility type, but may differ in their pragmatic annotations. For example, one MockSL specification may describe an object as it appears in memory in a C++ process. Another may describe a structure as it appears in a protocol message. However, both MockSL specifications in this example would be `Record` types with a certain set of `member` nodes.

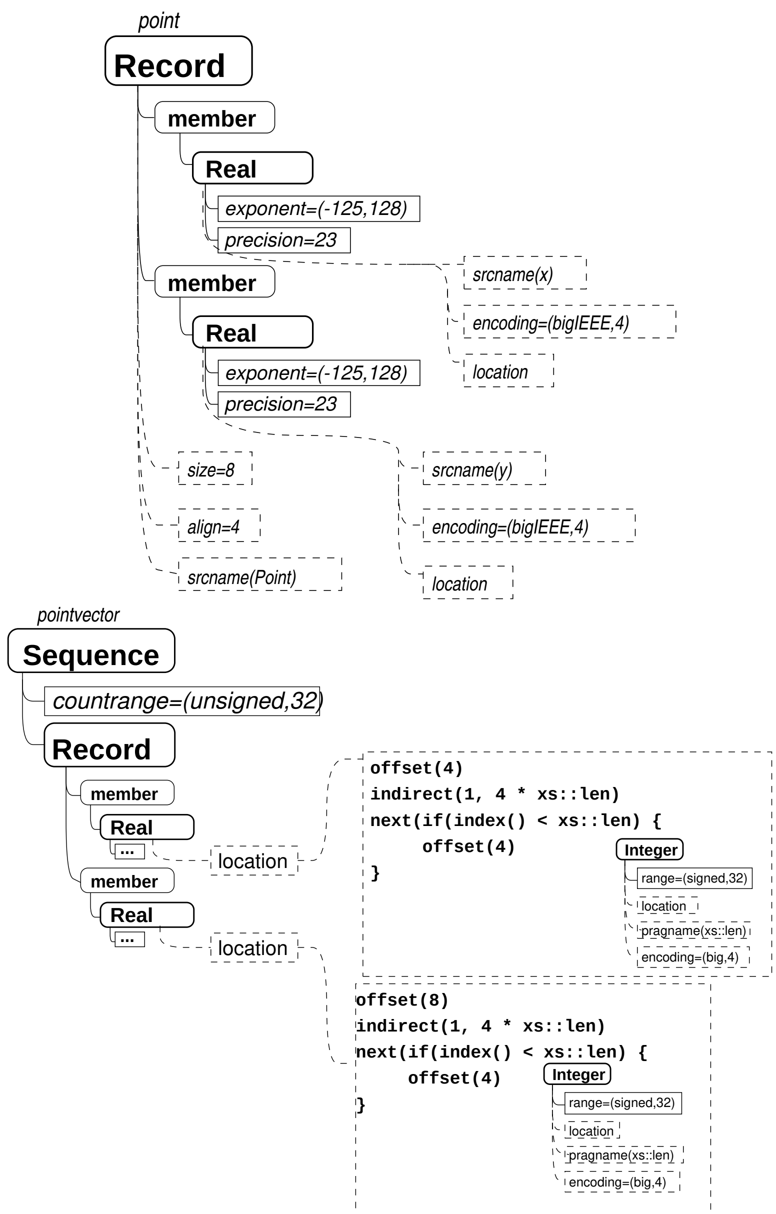
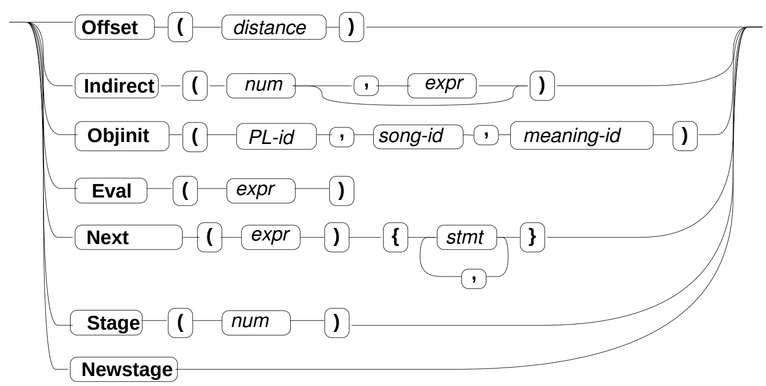


Figure 11: MockSL Types with Pragmatic Annotations

.

Location Statements



Location Expressions

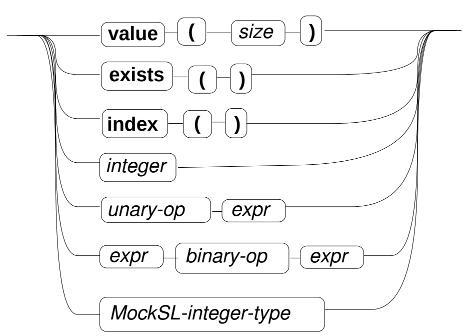


Figure 12: The syntax of location clauses

The stub generator walks the two MockSL trees in parallel and finds corresponding type nodes. This algorithm succeeds by definition, since interconvertible MockSL types are structurally equivalent. The location node attached to each leaf node is transformed into machine instructions that compute the address of the value described by the type node. The address may be in program memory or in a communication buffer. For values that are contiguous or related by simple pointer relationships, the code to locate them will have many common subexpressions. These can be recognized and eliminated by standard techniques, leading to efficient stubs.

There are some additional complications introduced to this process by conditionals and iteration over sequence members, but these cases can be handled simply using standard code generation techniques.

7 Status and Conclusions

We have constructed a prototype version of Mockingbird, with extractor and analyzer support for C++, Java, and CORBA IDL. Our emitter and runtime support Mockingbird programming in both C++ and Java. Interoperability between those two languages is being iteratively improved as we work on details of the system. Full interoperability with standard CORBA components is our next planned milestone. The prototype runs on PCs under Windows NT, Windows 95, and OS/2, and on IBM RS/6000 workstations under AIX. It is readily portable to other platforms.

Both Mockingbird languages have been used to develop applications. Mockingbird Java is being used by a fellow research group to develop a Java framework for collaborative objects [16]. Mockingbird C++ was used to prototype a semi-automated authenticated system for distributing binary code via the World Wide Web. We are evaluating other candidate applications, with emphasis on ones that significantly exploit multilanguage interoperability.

In this paper, we've accomplished the following.

1. We defined *interconvertibility*, a new criterion for stub correctness in distributed programming. We showed that interconvertibility is a better criterion for multi-language interoperation than the more restrictive type preservation criterion.

2. We showed that a system based on interconvertibility can interoperate with older systems using more restrictive criteria, so that Mockingbird does not have to be adopted by everyone in order to be useful.

3. We described MockSL, an intermediate representation which describes type signatures in terms of interconvertible value-sets. We described how type signatures in C++, Java and CORBA IDL can be analyzed into MockSL.

4. Although finding all possible interconvertible types is not achievable, we showed how this goal can be approximated conservatively by using an interconvertibility type system and an extensible set of canonicalizing transformations.

5. We showed how code is automatically generated from MockSL.

The Mockingbird project builds on the insights and experiences of the Concert project [18, 3, 4, 5], particularly the idea of using a specialized internal representation for interfaces [5]. However, in Mockingbird we have organized the compilation system more explicitly around the problem of interoperation. In addition, the Mockingbird tools are implemented so as to avoid either nonstandard compilers or preprocessing, making them much cleaner to use, and faster to adapt to new languages.

We expect the Mockingbird project to continue refining these ideas, and measuring the results. Future work includes (1) formalizing the analysis process, (2) completing the initial interoperation demonstration between C++ and Java, (3) completing CORBA support, (4) optimizing the stubs, (5) expanding and regularizing the annotations applied to C++ and Java types, (6) considering other canonicalizing transformations, (7) designing synthesizers (components to help a programmer find a "good" declaration in one language to match one used in another), (8) extending Mockingbird to other languages (like SmallTalk and Cobol) and (9) extending Mockingbird to other IDL-based technologies (like Distributed COM).

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1.2 edition, 1993.

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

[14] 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.

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

[16] Sun Microsystems, Inc. *Mapping IDL to Java*, alpha2.1 edition, may 1996.

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

[18] S. A. Yemini, G. Goldszmidt, A. Stoyenko, Y. Wei, and L. Beeck. Concert: A high-level-language approach to heterogeneous distributed systems. In *The Ninth International Conference on Distributed Computing Systems*, pages 162–171. IEEE Computer Society, June 1989.