Object Imaging

Satish R. Thatté*

Abstract

As heterogeneous and distributed object systems become more common, interoperability of system components is becoming a pressing issue. One important challenge in making such component-based systems work is support for object transport. Earlier data transport standards such as XDR focused primarily on marshalling RPC parameters and results for transport between different processor architectures. Effective use of the potential benefits of heterogeneous distributed object systems will require object transport at many levels of abstraction and granularity, ranging from small objects exchanged via RPC to large multimedia documents transferred across environments via the world-wide web. To emphasize the transformations involved, we refer to the problem of producing copies of objects across different environments as object imaging. In this paper we propose a very general notion of object imaging in which a flexible knowledge-base in the form of a dual logic of conversion and conformance is used to represent elements of imaging knowledge as composable combinators. This approach has the potential to seamlessly cover the entire range of imaging problems from simple primitive datatypes to complex object structures, and supports a variety of notions about conformance of object and interface types, including the multiple-inheritance single-interface object model of CORBA and the inheritance-averse multiple-interface model of COM.

1 Introduction

One of the benefits expected from object-oriented technology in software construction is the possibility of building open evolving software systems from collections of interoperable components. Potential components include class libraries, single-focus tools of the kind that are expected to be integrated using technologies such as OpenDoc [3] and OLE [4], and system components such as databases and file systems. A fundamental problem in establishing interoperability of components is support for object transport between components residing in different execution environments. The granularity of objects involved in transport may vary greatly. They may be as small as complex numbers or as large as compound documents involved in drag-and-drop operations.

*Department of Mathematics and Computer Science, Clarkson University, Potsdam. NY 13699-5815. E-mail: satish@sun.mcs.clarkson.edu.

The considerably simpler related problem of transporting both simple and complex data values has been addressed in the context of RPC mechanisms with marshalling schemes. Marshalling is concerned with issues like byte order, floating point representations, and data layout in arrays and records. This has been tackled traditionally with external data representation standards like XDR [23], which establish a one-to-one correspondence between standard atomic and aggregate types in different execution environments. Such marshalling schemes are clearly not capable of supporting transport of abstract objects, although they are necessary as a lower-level layer on which object transport can be built in a distributed object system. For various reasons, the leading component oriented object models being developed today, CORBA [17] and COM [15], do not treat object transport as a primary operation. Their default assumption is that objects are passed-by-reference, *i.e.*, only references to objects[1] are transported, not objects themselves. We therefore begin with an explanation of the need for transport of abstract objects, and the shortcomings of current approaches to the problem.

Motivations for Object Imaging

When components exchanging objects reside in different processes, there are many situations in which it is desirable or even necessary to make a copy of an object in the receiving environment rather than merely copying a reference. Examples occur at many levels of granularity:

- It is clearly impractical to expect small objects (such as complex numbers) to be used remotely since such objects tend to be numerous and the operations on them tend to be simple. Using such objects remotely would be prohibitively expensive. This problem is most likely to come up in passing parameters anbd results at the RPC level.

- A large object may be transported because accessing it remotely may impose an unacceptable cost in response time. Transport of multi-media document parts by world-wide web browsers is a good example.

- An object may be part of an aggregate such as a compound document on a remote system where the aggregate needs to be made persistent. Since the object may be transient in its original location, it may need to be transported for "embalming" purposes.

- Remote use may be simply impossible due to platform mismatches. For instance, if a user drags an icon for a document residing on a Macintosh and drops it on an icon representing an application on a machine running Windows, the Windows application will not in general be able to use the document remotely—it will need a local copy produced via (probably both hardware and software) format conversions.

[1] In the case of COM, it would be more accurate to say references to interfaces.

Effective interoperability standards must therefore support some notion of object transport. In the presence of heterogeneity, the only practical proposition *in general* is to transport (state) representations only, not code for methods, since code is extremely difficult to migrate even across software platforms, let alone hardware ones. Specifically, we must assume that when transporting an object O of class C_1 from environment E_1 to environment E_2, there is a *corresponding* class C_2 in E_2 which can provide the same functionality if the representation of O is transported to E_2[2]. It is therefore more appropriate to call this process object *imaging* rather than object transport. Object imaging includes and generalizes the current data transport protocols based on representation standards such as XDR. The imaging of abstract types is a more general problem than the transport of data structure representations—the polar and cartesian representations of complex numbers are identical as data structures. The problem considered in this paper is to identify the kinds of knowledge required to support a general notion of object imaging and to find an appropriate knowledge representation scheme.

Shortcomings of Current Approaches

Some of the essential mechanisms for object imaging are offered by several currently available systems. The specification for the common services layer on top of CORBA reflects current practice by offering standard copying [18] and externalization [19] services (the former as one of the life-cycle services)[3]. The copying service is expected to be a restricted version of what is possible with externalization. The externalization service provides a standard way for an object to produce an external representation of its state, using a flat stream format suitable for persistent storage and network transport. The service also provides a way to internalize such a stream in a different environment using an appropriate "object factory" to produce a corresponding object there. It is obviously possible to combine existing data transport protocols with the externalization service to achieve object imaging. The primary model for externalization is the various wire-format standards used in distributed systems for marshalling data structures. The same idea can be extended to larger objects with proposed standards such as the Rich Text Format (RTF) and the Graphical Interchange Format (GIF). The main current alternative to standard formats is custom conversions when the target type is known, as is often the case in drag-and-drop operations, for instance. This approach has been popular in desktop applications such as word processors.

Although both of these approaches are compatible with the externalization model, they have the disadvantage of making very strong assumptions about

[2]The representation could be treated in either a "shallow" or "deep" manner during transport. The treatment of graph structures is also a well-known problem addressed in the OMG services proposals [18, 19].

[3]The uniform data transfer and structured storage services associated with COM implement related functionality. For expository purposes we focus on the CORBA services. The following remarks can be paraphrased for the corresponding COM services as well.

the relationship between the object types involved in imaging. Many people advocate standards in interfaces and formats as a way of achieving interoperability in open object systems. However, the process of arriving at standards is difficult, both because software technology changes rapidly and because software vendors compete on the basis of product differentiation at least as much as on performance. Standard formats are therefore not likely to be available for most objects. On the other hand, custom conversions assume a degree of mutual knowledge that is contrary to the goals of open systems. For instance, pictures are commonly stored in GIF, JPEG and TIFF formats among others. Consider sending a GIF-formatted picture over the web to remote environments where the available viewers in each support only one of the other two formats. Given the lack of a standard format, custom conversions are called for. Knowledge about all these formats must therefore be available inside each source object in order to externalize appropriate representations for all possible target types. This clearly goes against fundamental principles of object design in open systems since this extraneous knowledge makes such objects fragile and subject to continual change due to the introduction of new picture formats, new requirements such as encryption, and so on. Each object ought to know only how to externalize/internalize its own state in a standard way, and knowledge of transformation to other formats can and should be maintained and applied externally. In short, the complexity of imaging in an evolving heterogeneous environment is difficult to handle *entirely* with *closely coupled* approaches based on custom converters or format standards. We need a *loosely coupled* approach based on a uniform knowledge-representation framework which provides an umbrella under which many approaches can coexist and evolve as object types are introduced, converters added, standards established.

Goals for an Imaging Service

The considerations above provide support for our claim that object imaging is a problem that needs to be investigated in its own right, rather than being seen as a simple application of existing or proposed mechanisms in the realm of distributed objects. Among other things, the imaging process needs an open evolving knowledge-base that registers convertibility relationships between atomic and parameterized data and object types and the corresponding conversions. In fact, one may need multiple knowledge-bases of this kind since imaging occurs at many layers in a distributed object system, and each layer potentially has different requirements. For instance, at the networking level, imaging processes are statically determined, and compiled into RPC stubs, whereas at the user-interface level, imaging tends to be triggered dynamically by spontaneous user actions such as following connections in a hypertext web document or drag-and-drop in a compound document based on OpenDoc or OLE. The knowledge-base for RPC level imaging would therefore be used primarily by a compile-time tool such as RPCGEN, and since performance requirements at this level are often stringent, the use of declarative specifications in conjunction with techniques such as deforestation [27] to optimize the conversions to and from wire-formats is

important. The knowledge-base for document-level imaging, in contrast, would most likely be a part of an *imaging service* similar to naming, security, and other run-time services.

In this paper we focus on knowledge-representation issues that are independent of the specific requirements of any particular layer. For generality, the knowledge-base needs to incorporate the following principles:

- Knowledge representation based on relationships between atomic and parameterized types.
- Seamless integration of primitive and user-defined types[4].
- Composability of imaging knowledge along two dimensions:
  - Part/whole composition based on the use of parameterized types in the construction of type expressions.
  - Sequential composition based on transitivity, *i.e.*, multi-step imaging.
- Complete automation of the reasoning processes involved in imaging.

Leaving aside network transport, the fundamental process in object imaging is type transformation. Both externalization and internalization can be represented as type transformations. External representation transformations such as GIF-to-JPEG can also be seen as type transformations. Moreover, the notion of type covers both primitive data values and abstract objects. A type-centric knowledge-base therefore facilitates the ability to freely combine imaging knowledge along all of the dimensions mentioned above. Such a knowledge-base will in practice consist of a conversion registry for object types, and will have to deal with many systems issues such as distributed reasoning using knowledge from multiple registries, uniform scripting models for conversion scripts, activation protocols for stand-alone converters, and so forth. These issues are not discussed in this paper. which confines itself to *symbolic* knowledge-representation and reasoning.

Motivation for Conformance Considerations

We propose an approach to knowledge representation based on two main elements: an order-sorted conversion logic and a logic of sort conformance, linked by the sorts and signatures of type constants and constructors. The need for a conversion logic is obvious given that the main computational process to be automated in object imaging is the inference and application of conversions between object types. For instance, the isomorphism (symmetric convertibility relation)

[4]Apart from its intuitive appeal, erasing the distinction between primitive and user-defined types is important in practice because user-defined types are often transformed to standard data structures during multistep imaging.

```
class Printable {
  public:
    virtual void print(ostream&) = 0;
};
template<class Key, class Value> class Table
                                : public Printable {
  public:
    void update(Key,Value);
    Value lookup(Key);
    Table merge(Table&);
    void print(ostream& o=cout) { rep.print(o); }
  private:
   struct Pair : public Printable {
     Key index; Value val;
     void print(ostream& o)
               { o << index; o << val; }
     int operator==(Pair& p) { index == p.index; }
   };
   List<Pair> rep;
};
```

Figure 1: A Template Class: `Table`

$$\text{polar(r)} \quad \underset{\text{internalize}}{\overset{\text{externalize}}{\rightleftarrows}} \quad \text{key} \times \text{r} \times \text{r}$$

might express the externalization process for a general notion of polar complex numbers. The product constructor $\times$ is used here to represent `structs`. The type parameter `r` is the real number type (such as `vax_float` or `powerpc_double`) used for representing the real and imaginary parts, and the `key` type is meant to identify the appropriate "object factory" for internalizing the representation. It is best thought of as a type tag. Conversions are essentially rewrite rules. Rewriting logics satisfy the composability requirements mentioned above and are also amenable to automated reasoning. We have used a similar conversion logic in [24] to express type conversions in polymorphic functional languages.

The role and form of the conformance logic are perhaps less obvious. Unlike the primitive universal type constructors such as products and arrays, user-defined parameterized types typically make *assumptions* about the functionality supported by the type parameters they are prepared to accept. The knowledge base for imaging needs to formalize and use these constraints to check that types and conversions are well-formed, and to limit the application of conversions to ensure that functionality constraints on the parameters of type constructors are not violated.

A simple RPC-level example will suffice to illustrate these points. Consider

the C++ container class template `Table` given in Figure 1. The type parameters `Key` and `Value` of `Table` are not free. The way they are used in the implementation structure `Pair` implies that the operation << is defined on all objects of types `Key` and `Value`, and == is defined on all objects of type `Key`. No mention is made of any inheritance relationship between any specific classes and the type parameters—the constraints require only available functionality.

Now suppose we have a type constructor `list`. We might wish to use the conversion from a list of pairs to a pair of lists by declaring the rule

$$\texttt{list(a} \times \texttt{b)} \longrightarrow \texttt{list(a)} \times \texttt{list(b)}$$

where, unlike C++, we use ordinary parantheses to construct type expressions. The symbol $\longrightarrow$ denotes one-step convertibility. The actual conversion function can be used as a tag when it is of interest, as in the isomorphism for complex number types above. Assuming that `int` supports both == and <<, and instances of `list` support <<, the type $\texttt{Table(int,list(int} \times \texttt{bool))}$ is well-formed. Now by the usual laws of rewriting, we should be able to claim

$$\texttt{Table(int,list(int} \times \texttt{bool))} \longrightarrow \texttt{Table(int, list(int)} \times \texttt{list(bool))}$$

However, the C++ `iostream` library does not define << as an operation for arbitrary `structs`, and so, starting with a well-formed type we are in danger of creating an ill-formed type by the application of a perfectly reasonable looking conversion. The point to note is that type conversion does not in general preserve interfaces. Maintaining the integrity of compound types in the conversion process therefore requires consideration of interface types as distinct from object types and specification of conformance relationships between the two.

Designing an Abstract Conformance Framework

One of the difficulties in choosing a symbolic representation for interface types is the variety of specific notions of interface and conformance that must be covered if the choice is to claim generality. For instance, conformance constraints for type parameters in programming languages are based either on subtyping or on functionality. Eiffel [14] uses an object type as an upper bound for each type parameter of a generic type and checks the integrity of compound types at compile-time using (inheritance-based) subtyping. Conformance for the type parameters of templates in C++ [8] is based on available functionality, and is enforced by the linker. In theoretical terms, parametric polymorphism in Eiffel is bounded [6] while in C++ it is F-bounded [5].

Component oriented systems such as OpenDoc [3] and OLE [4] tend to require their components to support *multiple* interfaces. However, as a result of differences in the underlying object models (CORBA and COM respectively), they use radically different notions of conformance. OpenDoc uses objects with single principal interfaces which constitute object types, and a notion of conformance based on interface inheritance, similar to most statically typed languages. Multiple interfaces are merged using multiple inheritance. OLE expects objects

to offer multiple independent interfaces. Moreover, COM strongly discourages the use of inheritance in *defining* interfaces (as opposed to implementing them). Conformance of interface types in OLE therefore reduces to identity, except for the root `IUnknown` interface. On the other hand, object type and interface type are potentially distinct concepts in OLE/COM, since access to one interface can be used to gain access to other interfaces on the same object using the `QueryInterface` operation. Both the actual and the expected type of a COM *object* therefore represents multiple interface types.

The use of `QueryInterface` in COM also implies the existence of *conditional* interfaces, since a container might decide at run-time whether or not to offer an interface based on the availability of certain interfaces in its components. Conditional interfaces are also possible in languages like C++ where "functionality marshalling" for type parameters occurs at link-time based on need. The available interface(s) corresponding to a compound object type therefore need not be fixed. Functionality marshalling also brings up the issue of wrapper generation. When a COM object needs to masquerade as a CORBA object, or vice versa, one could wrap the object with an appropriate veneer to make this possible. In general, an object type is capable of supporting many interfaces not intrinsic to it with some adaptation. Such adaptability relations can be thought of as coercions realized by wrappers. In determining conformance of object types when used as parameters, it is important to take coercibility into account, and the conformance logic should be able to invoke wrapper generation automatically as needed.

It is desirable to use a uniform symbolic framework for all these diverse contexts because conversions often cross conformance-domains. For instance, an RPC call from a C++ client may send object parameters to an Eiffel-based server, or a content object may be dragged from an OpenDoc compound document and dropped into an OLE container. We therefore need an interface formalism that is abstract enough to admit a variety of interpretations, and has enough structure to represent the relation of conformance between object and interface types, provide for inheritance of interfaces and adaptability via wrapper generation, and represent the necessary interaction between the conformance logic and the reasoning algorithms for conversion. We believe that the notion of sorts from order-sorted equational logic can be adapted to fit the bill, if we extend the sort structure with appropriate intersection sorts.

In the rest of this paper, we illustrate the range of imaging problems with examples in Section 2, and present a (preliminary) formal approach to a general solution in Section 3. Related work is reviewed briefly in Section 4.

2 Symbolic Knowledge Representation

The examples used in this section to illustrate knowledge representation issues use two kinds of conversion rules. The standard rule is of the form $t \longrightarrow u$, where t and u are type expressions, possibly containing formal parameters in the form of variables. This defines a one-way or irreversible conversion. A good example

of such a conversion would be `document` $\longrightarrow$ `postscript` which represents a rendering of a document for printing on a PostScript printer. Reasoning with such rules permits inference of multistep relations of the form $t \overset{*}{\longrightarrow} u$. However, very often conversion relations are reversible. We occasionally refer to such relations as isomorphisms although that is not strictly correct since there is no guarantee that, for instance, converting a Word document to WordPerfect form and then back to Word form will reproduce the original. Such reversible relations are common enough to merit the notation $t \longleftrightarrow u$ for a reversible rule, and $t \overset{*}{\longleftrightarrow} u$ for a multistep reversible conversion. Practically all the examples in this section use reversible conversions.

Primitive Types

We start with simple examples involving primitive types, an extension of the problem tackled by systems such as XDR. For clarity, we leave out the conversion functions in convertibility relations. Inference of conversions is not difficult given that we can find a proof for the simpler convertibility relationship between types.

Suppose we are concerned about interoperation of components that may reside on computers based on either the Vax or the PowerPC architecture. We start by expressing the convertibility of arrays between the two environments. However, instead of using a direct equation, we choose to reflect actual practice by equating each array type to a common data format used in the distributed object infrastructure.

```
vax_array(a) ⟷ xdr_array(a)      ppc_array(a) ⟷ xdr_array(a)
```

This allows easy extension of the equivalence to other architectures. We use the convention that wire format type names are prefixed with `xdr_`. Now given additional isomorphisms for atomic types

```
vax_float ⟷ xdr_float        vax_int ⟷ xdr_int
ppc_float ⟷ xdr_float        ppc_int ⟷ xdr_int
```

we can obviously prove equational theorems such as

```
vax_array(vax_float) ⟷* ppc_array(ppc_float)
```

representing marshalling/unmarshalling processes currently used in RPC systems.

The treatment of record types needs more care. Records can often be conveniently represented using cross products, since field labels are not important for portability when it is appropriate to treat records as multisets (or bags). On the other hand, records sometimes need to be treated more like abstract types as in the XDR system (illustrated in the complex number marshalling example below). In the former case, the type

```
struct { int x;  float y;  int z[10]; }
```

can be represented as `int` × `float` × `<int>`. The product constructor in this context is naturally associative and commutative—the same symbolic representation can be used for

```
struct { float y; struct c { int x; int z[10]; } w; }
```

Associativity and commutativity are represented by the conversions

$$a \times (b \times c) \longleftrightarrow (a \times b) \times c \qquad a \times b \longleftrightarrow b \times a$$

and the corresponding pairs of maps can be used to adjust address alignment and similar details if necessary.

As an example of an equivalence that is *not* supported by typical RPC-oriented marshalling mechanisms, tables (finite maps from keys to values) can be represented as a pair of sequences or as a sequence of pairs (among other possibilities). The equivalence of the two representations can be expressed as:

$$\texttt{T\_array}(a \times b) \longleftrightarrow \texttt{T\_array}(a) \times \texttt{T\_array}(b)$$

where `T_array` is treated as a template for array types. We can now prove that the representation of a table mapping strings to floats on a Vax can be transformed to the slightly different representation of a similar table on a PowerPC:

$$\texttt{vax\_array}(\texttt{vax\_string}) \times \texttt{vax\_array}(\texttt{vax\_float}) \overset{*}{\longleftrightarrow} \texttt{ppc\_array}(\texttt{ppc\_string} \times \texttt{ppc\_float})$$

In fact, since the *order* of fields in a record is unimportant

$$\texttt{vax\_array}(\texttt{vax\_string}) \times \texttt{vax\_array}(\texttt{vax\_float}) \overset{*}{\longleftrightarrow} \texttt{ppc\_array}(\texttt{ppc\_float} \times \texttt{ppc\_string})$$

By a simple process of accumulation of obvious knowledge about isomorphism relations, we have reached the point where the equivalence of relatively complex representation types can be inferred, reaching considerably beyond what is possible with traditional marshalling techniques.

Atomic Object Types

Now consider the problem of expressing the standard equivalence between the cartesian and polar representations of complex numbers as part of a framework for interoperation of numerical simulation components. Specifically, we would like to be able to marshall a polar complex number as a parameter in the client and unmarshall it as a cartesian complex number in the server. The two atomic object types are `ppc_polar` and `cray_cartesian`. Given the likely absence of a wire-format standard for complex numbers, the marshalling cannot be done directly unless we tag the data and expect the server to "make it right". We would ideally like to simply express the necessary knowledge symbolically and let a system like RPCGEN use it to insert the appropriate conversions. One

approach to this is to expose the representation of each object type in the form of a structure, except in this case the identity of the structure is important in distinguishing between the polar and cartesian forms.

```
ppc_polar ⟷ ppc_polar_struct
cray_cartesian ⟷ cray_cart_struct
```

We follow the convention of using names appended with `_struct` for structure types with significant identities. The computational cost of such exposure can be made negligible, using private inheritance in C++ for instance. The correspondance between the exposed representations represents local conversions:

```
ppc_cart_struct ⟷ ppc_polar_struct
cray_cart_struct ⟷ cray_polar_struct
```

Finally, the marshalling and unmarshalling of the exposed cartesian representations is symbolized by:

```
ppc_cart_struct ⟷ xdr_cart_struct
cray_cart_struct ⟷ xdr_cart_struct
```

which allows us to prove `ppc_polar` $\overset{*}{\longleftrightarrow}$ `cray_cartesian` and automatically generate the corresponding conversions for marshalling and unmarshalling. This example is meant to illustrate an approach to transport of abstract atomic objects. In the case of complex numbers, given the restriction to two representations, a more direct approach may be viable and attractive.

Compound Object Types

We now consider an entirely different kind of example to demostrate the range of applicability of the symbolic notation. The approach used above is clearly applicable to conversion knowledge regarding atomic document types such as `word_win` (Word for Windows) and `wp_mac` (WordPerfect on a Macintosh). With the advent of object linking and embedding technologies such as OpenDoc and OLE, a document can be seen as a container with many different kinds of content elements, such as text, spreadsheets, drawings, pictures, and even sound and video. A traditional application such as Word plays two conceptually separate roles as both a container widget and a text widget. If we separate these roles and use types to represent pure roles, we would have atomic types for content widgets like `word_mac` and `quattro_win` and type constructors for container widgets like `c_wp_mac` (WordPerfect in its container role on a Macintosh). An actual compound document might be of the compound type `c_wp_mac(wp_mac,excel_mac)`, *i.e.*, a WordPerfect container with WordPerfect text and Excel spreadsheet content elements. Traditional custom conversion utilities (usually bundled with applications) could be represented by relations of the form

```
wp_mac ⟷ wp_win        excel_win ⟷ quattro_win
```

representing cross-platform and cross-widget conversions, respectively[5]. Similarly, container conversions would yield non-atomic relations such as[6]

```
c_wp_mac(a,b)  ⟷  c_word_mac(a,b)
c_wp_mac(a,b)  ⟷  c_wp_win(a,b)
```

which represents a WordPerfect to Word container conversion on the Macintosh platform, and WordPerfect container conversion from a Macintosh to a Windows format, assuming two distinct kinds of content elements. Now if a user drags a document of type `c_wp_mac(wp_mac,excel_mac)` to the Word icon on a Windows machine where Quattro is the only spreadsheet installed, the drag-and-drop manager can derive and use the isomorphism

```
c_wp_mac(wp_mac,excel_mac)  ⟷  c_word_win(word_win,quattro_win)
```

to infer the necessary (multistep) conversions and apply them transparently without user intervention. Actually, this example cheats a little by ignoring the dependence of compound document containers on standard interfaces they expect content objects to support. The symbolic representation of this dependence is an important topic to which we now turn.

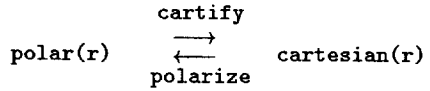
Sorts and Signatures

A full-fledged formalism for interface types is not desirable as part of a reasoning system based on rewriting. As we noted in the introduction, it is sufficient to introduce a notion of *sorts* in the conversion logic. In practical terms, a sort represents an interface type as distinct from an object type—we use the word *type* to refer to an *object* type throughout this section. Formally, a sort has two natural interpretations: an intensional one and an extensional one. The intension of a sort is the set of features supported by any type that possesses the sort. The extension is the set of ground types that possess the sort. Constraining a type *variable* with a sort simply constrains the variable to range over the extension of that sort. Intuitively, a ground type has a sort exactly when it conforms to the interface functionality required by the sort. Each type has at least one sort, and may have more. Multiple interfaces can be directly represented by intersection sorts as we see below. For a CORBA object type τ, the statement that τ possesses a sort s means that the principal interface represented by τ inherits from the interface(s) represented by s. For a COM object type τ, the same statement means that objects of type τ expose the interface(s) corresponding to s.

The conversion logic obviously needs the extensional interpretation of sorts. Introducing sorts there requires adding simple annotations to each variable in a conversion rule. An annotation $\mathtt{a}{:}\mathcal{X}$ constrains the type variable a to sort $\mathcal{X}$. For instance, suppose we want to represent a direct isomorphism

[5] Of course, the availability of these conversions is often restricted to one-way conversions.

[6] There is a technical problem with the number of parameters for container types. The number obviously varies, but this could be handled, by using schemas of relations.



between complex number types on a given platform, using a type parameter to represent the real and imaginary parts, instead of treating each possible combination of architecture, precision, and complex number form as a distinct type. The type parameter cannot meaningfully range over all types, since a type such as, say, `string`, cannot be used to represent the parts of a complex number—it does not support the operations required of such a part. This constraint can be expressed by restricting the range of the type variable. Suppose we let sort $\mathcal{R}$ represent the functionality of real number operations (or the set of real number types such as `vax_float` and `ppc_double`). We could then have:

$$\texttt{polar(r:}\mathcal{R}\texttt{)} \longleftrightarrow \texttt{cartesian(r:}\mathcal{R}\texttt{)}$$

If we let $\mathcal{T}$ represent the sort with no functionality, *i.e.*, the set of all ground types, then $\mathcal{R}$ is a subsort of $\mathcal{T}$ (denoted by $\mathcal{R} \leq \mathcal{T}$) if we use the natural order based on the subset relationship between sort extensions. The conversion theory we are constructing is therefore not only many-sorted, it is naturally *order sorted*: we have a quasi-ordered set of sorts such as $\mathcal{T}$ and $\mathcal{R}$, (type) variables have sorts representing their range, and (type) constants and constructors have signatures representing their relationships to sorts. For instance, suppose we have a sort of "number-like" types (those which support the four basic arithmetic operators) called $\mathcal{N}$. Any type constructed by the type constructor `polar` clearly belongs to $\mathcal{N}$. The signatures of the constructors $\times$ and `polar` therefore could be given as

$$\texttt{polar} : \mathcal{R} \rightarrow \mathcal{N} \qquad \times : (\mathcal{T}, \mathcal{T}) \rightarrow \mathcal{T}$$

and we have the sort structure $\mathcal{R} \leq \mathcal{N} \leq \mathcal{T}$. In general, a type constructor has a signature that shows the relationship between the functionality it expects its parameters to support and the functionality supported by the types it constructs. Note that a type constructor need not have a unique signature. For instance, suppose we use $\mathcal{E}$ to represent the sort of types which support an equality operation. It is not hard to see that $\times$ *could also* have the signature $\times : (\mathcal{E}, \mathcal{E}) \rightarrow \mathcal{E}$ since equality on products can be defined pointwise given that it is defined on the two parts in the product. Moreover, if we assume that the functionality associated with $\mathcal{R}$ and $\mathcal{N}$ does not include equality, then we have $\mathcal{E} \leq \mathcal{T}$ but neither $\mathcal{N} \leq \mathcal{E}$, nor $\mathcal{E} \leq \mathcal{N}$, and hence our sort structure is no longer linearly ordered.

In the terminology used in the literature on order-sorted algebra, operators such as $\times$, which possess multiple signatures, are called *polymorphic* operators. Polymorphic type constructors offer flexible *conditional* functionality.

Signatures influence type composition in two ways. They decide if a type expression is well formed, and if it is, which sort(s) the expression may claim. For instance, suppose `float`$:\mathcal{R}$, `float`$:\mathcal{E}$, but `string` does not possess sort $\mathcal{R}$. Then `polar(string)` is ill-formed, but `polar(float)` is well-formed and

possesses *two unrelated* sorts: $\mathcal{N}$ and $\mathcal{E}$ (the latter using the additional signature for `polar` given below).

The functionality associated with sorts is naturally additive, in particular because an object may support multiple interfaces. Combining two sorts results in an *intersection* of the sorts (seen as sets of types). We use $\sqcap$ to represent the intersection operator for sorts, in keeping with the lattice-like structure being constructed. If $s = s_1 \sqcap \cdots \sqcap s_k$ is an intersection sort, then τ is an instance of s iff it is an instance of each s_i. Of course, $\sqcap$ is an associative and commutative operation, and $s_1 \sqcap \cdots \sqcap s_k \leq s_i$ for $1 \leq i \leq k$. This is an intrinsic subsort relation—no wrapper generation is involved.

Intersection means merger through multiple inheritance for CORBA interfaces, which means that the sort hierarchy parallels the interface hierarchy and subsorts are essentially subtypes. In COM, an object type τ conforms to a sort $s = s_1 \sqcap \cdots \sqcap s_k$ only if it exposes interfaces corresponding to each atomic sort s_i. Note that τ may also expose additional interfaces, in which case its *principal* sort would be a subsort of s. Thus, the sort structure constructs a conformance hierarchy for COM where none existed before—*conformance in COM is based on subsort rather than subtype relations*, specifically on subsorts constructed as intersection sorts.

Even in the absence of multiple interfaces, as in the context of functionality requirements for template type parameters in C++, elevating intersection sorts to the status of first-class sorts helps to avoid a confusing proliferation of overlapping sorts. For instance, if we use $\mathcal{P}$ to represent the sort of printable types, then the signature of the `Table` template of Figure 1 can be given as

$$\texttt{Table} : (\mathcal{E} \sqcap \mathcal{P},\ \mathcal{P}) \rightarrow \mathcal{P}$$

instead of having to declare a new sort that combines the functionality of $\mathcal{E}$ and $\mathcal{P}$. Similarly, `polar` may also claim

$$\texttt{polar} : \mathcal{R} \sqcap \mathcal{E} \rightarrow \mathcal{E}$$

Finally, continuing the hypothetical compound documents example from previous sections, suppose the interfaces `Text`, `SpreadSheet` and `Document` define text, spreadsheet and document objects, respectively, and `RenderMac` represents objects that can be rendered and interactively modified on a Macintosh screen. In this case `c_word_mac` might have the signature

$$\texttt{(Text} \sqcap \texttt{RenderMac, SpreadSheet} \sqcap \texttt{RenderMac)} \rightarrow \texttt{Document}$$

and perhaps a second signature based on the COM persistence framework

$$\texttt{(IPersistFile, IPersistFile)} \rightarrow \texttt{IPersistFile}$$

meaning that if the text and spreadsheet types given to `c_word_mac` are capable of persistence, so is the resulting document type, otherwise not. In this way the dependencies of available interfaces for a `c_word_mac` object can be specified precisely and independently.

Conformance Checking and Wrapper Generation

Simple conformance checking, where the interface/functionality is intrinsic to the object type, is routine and well understood. More sophisticated structural conformance and interface adaptability relations based on subtyping for recursive types have been considered in the literature [2, 26], but we will not be concerned with them here. Our main concern is with conformance based on user-specified methods for generating auxiliary interfaces on demand. The details of how auxiliary interfaces are realized are highly platform-dependent. For instance, a CORBA object could support a new interface by interposition of an auxiliary object that delegates actual execution to the underlying intrinsic interface. This does not affect the identity of the object since CORBA specifically excludes identity checking on references precisely to support such interposition [9][7]. A COM object can achieve the same effect more directly through aggregation, since the problem of interface conflicts does not arise.

The initial examples in the following illustrate sort and signature declarations using a simple applicative notation for the intensional aspects of sorts, *i.e.*, sort definitions. We choose to make the type of `self` explicit in the signatures of operations. In standard object-oriented typing notations, including CORBA IDL, the signatures of intrinsic operations leave out the type of the recipient object. Technically, this is important in establishing subtype relationships corresponding to inheritance—due to the contravariant subtyping for intrinsic operations, making the type of the recipient explicit as an argument type would invalidate structural subtyping and substitutability, and fail to account for the polymorphic nature of such operations. However, the functionality represented by sorts may, in general, be a mixture of intrinsic and extrinsic operations, *e.g.*, in a system like C++-templates, where expected functionality for type parameters is not tightly organized into named interfaces, but is implied by usage. For instance, the printability constraint on the parameters of class `Table` in the introduction refers to an extrinsic (and overloaded) operation `<<` since the object used to select the appropriate implementation is the *second* parameter. We would also like to leave open the possibility of accounting for constructs that do not follow "standard" object-oriented practice, *e.g.*, multimethods in CLOS. Such explicit "self" parameters are also used in CORBA and COM language bindings.

The following sort declaration defines the functionality of number-like types:

```
sort 𝒩 contains T given
 { + : T × T → T, - : T × T → T, * : T × T → T, / : T × T → T }
```

where `T` plays the role of a prototypical instance of $\mathcal{N}$. The operator signatures in a sort declaration use only the prototypical instance and ground types. Every type constant and constructor has one or more signatures which must be declared and validated. For instance, we can establish the signature `int`:$\mathcal{N}$ using a `signature` declaration:

[7] Of course, the support for CORBA's agnostic approach to object identity is by no means universal in the CORBA community [21].

```
signature int : 𝒩 with
    { + = intAdd, - = intSub, * = intMul, / = intDiv }
```

where we assume for the sake of example that `intAdd`, *etc.*, are the integer arithmetic operations, each of which has the type `int×int→int`. Note that the declaration of the signature is accompanied by actual implementations of the required operations which can be (automatically) checked to ensure the validity of the signature. This can be thought of as instructions for the construction of a function table similar to the standard implementation of an interface in COM[8]. The declarations of equality and printable sorts can be given as:

```
sort ℰ contains T given { == : T × T → bool }
sort 𝒫 contains T given { << : ostream × T  → void }
```

where the operation << exhibits multimethod-like flexibility by using the *second* parameter to select its implementation in order to accommodate standard C++ output syntax. As an example of a nonatomic signature, the signature of the type constructor `Table` from Figure 1 might then be declared as

```
signature Table : (ℰ ⊓ 𝒫, 𝒫) → 𝒫 with
     { str << tab = tab.print(str) }
```

If we want to use the expression `Table(int,string)` as an instance of sort $\mathcal{P}$, we must use `signature` declarations to show that `int` is an instance of both $\mathcal{E}$ and $\mathcal{P}$ (and therefore of sort $\mathcal{E} \sqcap \mathcal{P}$) and `string` is an instance of $\mathcal{P}$ as well. The fact that << for `int` may be defined in the iostream library as intrinsic to class `ostream` while << for `string` may be user-defined and extrinsic to both `ostream` and `string` is not a problem since we make no distinction between the two situations.

The rest of the examples are based on compound document types and are meant to illustrate some situations that would trigger wrapper generation and some of the ways signatures and wrapper generation procedures might be specified in an environment based on well-defined named interfaces. Suppose we use `PO` to represent the basic interface for persistence in CORBA, and `P` to represent a similar interface in COM. We may have the following signatures for two containers:

```
wp_win  : (PO, PO) → PO          word_win  : (P, P) → P
```

We may also have the container conversion rule

```
wp_win(x : PO, y : PO) ⟶ word_win(x : P, y : P)
```

where the notation `x : PO` is used to declare that variable `x` can only be instantiated with type that are instances of sort `PO`. Note that the use of identical variable names on the two sides reflects the correspondence between the parts, even though their expected sorts differ. We may then need to infer

[8] CORBA does not define an implementation standard.

$$\texttt{wp\_win(pict\_win, chart\_win)} \longrightarrow \texttt{word\_win(pict\_win, chart\_win)}$$

if we cause a document of type `wp_win(pict_win, chart_win)` to be opened by a `word_win` server. The right-hand side is well-formed only if wrappers can be applied to the type parameters to change their sort. There are two ways such wrapper generation capabilities can be specified. One is to provide declarations analogous to `signature chart_win : P with ..` which directly specifies how to synthesize the required interface based on the underlying object type. This is possible with CORBA but may be inconvenient with COM because COM objects do not have principal interfaces. For COM, a subsort declaration $\texttt{PO} \leq \texttt{P}$ would be more appropriate. In general, a subsort declaration may assume multiple interfaces. For example, if an interface `EQ` is assumed besides `PO` to construct `P` then the declaration would be $\texttt{PO} \sqcap \texttt{EQ} \leq \texttt{P}$. Of course, such an explicit subsort declaration (as opposed to the implicit subsort relations established by intersection) must be accompanied by code that synthesizes the supersort interface given the subsort interface(s).

We conclude this section with an example where the imaging system is used more like a trading system. Suppose we have a different drag-and-drop gesture that causes a dropping of a document of type `wp_win(pict_win,chart_win)` on a `word_win` server icon to be interpreted as the attempted conversion

$$\texttt{wp\_win(pict\_win, chart\_win)} \overset{*}{\longrightarrow} \texttt{word\_win(x : P, y : P)}$$

where the target type is incompletely specified. The imaging system must then find target object types with the specified sorts (in this case `P`) for the types being "traded in": `pict_win` and `chart_win`. This may involve wrapper generation, actual conversion, or both.

3 Reasoning Algorithms for Imaging

In this section we describe a formal approach to reasoning about imaging. The idea is to formalize the problem space illustrated by the examples of the last section. The ideas presented here should not be viewed as finished theoretical results—they are more properly seen as a first attempt at formalizing the problem and an approach to a solution. Among the shortcomings of the present solution is the fact that we pay no attention to heuristics for optimizing either the efficiency of the algorithm or the efficiency of the derived conversions. A naive reasoning algorithm will in general yield unnecessarily long proofs leading to inefficient conversions that proceed through too many intermediate stages. It would be nice if the reasoning algorithm guaranteed a most efficient proof. This is a difficult problem for which theoretically complete solutions are unlikely in the case of unrestricted conversion theories [7]. However, there may be heuristic approaches that yield good results.

We assume a basic familiarity with the ideas and terminology of term rewriting (see [10] for a survey of rewriting concepts and results). Although the discussion in this section is entirely in terms of order-sorted term algebras and

rewriting systems, it is important to keep in mind that terms are object type expressions, and term rewriting corresponds to object conversions.

Delineating the Problem

Conversion rules are rewrite rules and the basic reasoning problem in imaging is a version of the classical word problem in algebra: determining whether two ground terms (in this case type expressions) are equivalent in the conversion theory. Since we allow asymmetrical conversion rules, equivalence must be replaced by reducibility in the corresponding rewriting theory. There are three additional aspects we need to consider:

1. Convertibility proofs must be accompanied by conversion functions/scripts.
2. The algebraic signature being order-sorted, sort checking is necessary and wrapper generation may be needed during rewriting.
3. The conversion target may be incompletely specified, *i.e.*, it may be a term containing variables, as in the "trading service" example above.

As we have noted before, the details of conversion functions and scripts are highly platform and context dependent and the technical problems that will be encountered in synthesizing and using them are likely to be "systems" oriented and cannot be addressed in a general reasoning framework. The mere identification of required conversions given a convertibility proof is a trivial problem rather similar to the problem of constructing a parse tree given a parsing proof. The same comments apply to wrapper generation relative to sort checking. In the formal notation and algorithms developed here, we therefore focus on convertibility proofs and sort checking, and ignore code synthesis. The problem of incompletely specified targets in rewriting, on the other hand, is both very important and amenable to a general solution. It forces us to generalize the problem under consideration to one of *order-sorted matching under reduction.* This specific problem has not been considered in the literature on order-sorted reasoning since it does not occur in the context of standard theorem-proving applications, but the ideas needed to solve it are substantially the same as in the well-understood problem of order-sorted equational unification and matching.

Finally, there is one aspect of practical object systems that is difficult to model formally, though not difficult to deal with in practical reasoning algorithms. Containers in compound document systems are equipped to hold an arbitrary number of parts with possibly different degrees of functionality. The symbol for a container thus has an infinite number of signatures. An infinite algebraic signature is not a problem in itself, but since containers are often involved in transformations, the rule-base also becomes theoretically infinite. In practice this is easy to deal with using rule and signature schemata but such schemata would unnecessarily complicate the formal system so we leave them out and assume that each symbol has a finite number of signatures.

The impact of sorts on the reasoning algorithms goes beyond preserving the well-formedness of type expressions. The fundamental processes in reasoning about imaging is term rewriting, which in turn is based on *matching* of rule-templates with actual expressions. Matching is a special case of unification, and unfortunately, unification in the presence of order-sorted signatures is considerably more problematic than in the unsorted case. To avoid excessive nondeterminism, it would help to ensure that free unification in the algebra of types is *unitary*, *i.e.*, for any set of equations there is a complete set of unifiers with at most one element. Smolka, *et al* [22] show that a finite algebraic signature is unitary if it is regular, coregular and downward complete. Moreover, for unitary algebraic signatures, unification and matching can be implemented in quasi-linear time in a straightforward way [13]. Regularity is simply the property that every expression has a principal (least) sort. This is not a problem since, as we see below, we can transform the signature to ensure a unique signature for each symbol, and therefore a unique sort for each term, at the cost of giving up something we don't have anyhow: sort preservation during conversion. Coregularity is also trivial given this, since it merely requires that there is a least restrictive way to construct a sort using a given function symbol. Downward completeness—the existence of infimums for finite sets of sorts—is more problematic since it requires us to support *arbitrary* sort intersections. In a heterogeneous environment, this might mean, for instance, that objects must be able to support both CORBA and COM interfaces simultaneously, which might not be possible. On the other hand, many practical systems may be homogeneous enough to guarantee downward completeness of the sort structure, allowing for an efficient rewriting procedure.

Notation and Definitions

We work with a finite set of ordered sorts and a finite signature consisting of function symbols with *unique* signatures. There is a denumerable set of variables for use in conversion rules and in the reasoning process. Each variable may possess several sorts, but its sort in each occurrence must be specified. These restrictions are the reverse of the usual ones where function symbols are permitted multiple signatures but variables are restricted to single sorts. In our case, change of signatures for a function symbol (type constant or constructor) must in general trigger wrapper generation to assemble the corresponding functionality, so it is convenient to treat it as a transformation. This amounts to an implicit transformation of the actual algebraic signature. On the other hand, since conversions do not in general preserve interfaces, variables may change sorts as part of a conversion rule. To help keep track of which part in an original compound type corresponds to which part in a transformed one, the identity of the variables must be maintained.

Let $(S, \leq)$ denote a quasi-ordered set of sorts and Σ an algebraic signature of function symbols. Σ consists of a family of subsignatures $\Sigma_{w,s}$. Each $f \in \Sigma_{w,s}$ has the signature $w \rightarrow s$, where w is a possibly empty sequence $s_1 s_2 \cdots s_k$ of sorts. For instance, the signature for `Table` may be $s_1 s_2 \rightarrow s$ where $s_1 = \mathcal{E} \sqcap \mathcal{P}$,

$s_2 = \mathcal{P}$ and $s = \mathcal{P}$. The signature for `int` might be $\Lambda \to \mathcal{N}$, where Λ is the empty sequence. We also have a denumerable set $\mathcal{V}$ of variables. Each occurrence of a variable must be tagged with a sort $s \in S$, but different occurrences of the same variable may be tagged with different sorts. The set of well-sorted terms over Σ and $\mathcal{V}$ is denoted by $\mathcal{T}_\Sigma(\mathcal{V})$, or just $\mathcal{T}$ when Σ and $\mathcal{V}$ are known from the context. $\mathcal{T}_\Sigma$ denotes the set of ground terms, *i.e.*, terms without variables. Every well-formed term has at least one sort. A term t can be an instance of a sort s in one of four ways:

1. $t \in \Sigma_{\Lambda,s}$, *i.e.*, t is a constant of sort s.
2. $t = x^s$, *i.e.*, t is a variable of sort s.
3. $t = f(t_1, \ldots, t_k)$, $f \in \Sigma_{s_1 \cdots s_k, s}$, and each t_i is an instance of sort s_i.
4. t is an instance of a sort s' and $s' \leq s$.

Substitutions are maps from $\mathcal{V}$ to $\mathcal{T}$ which map a *finite* number of variables (the *domain* of the substitution) to terms other than themselves (the *range* of the substitution). Substitutions are extended to apply to terms in the usual way. The application of a substitution ξ to a term t is denoted by ξt. The composition of two substitutions ξ and ζ is denoted by $\xi\zeta$—$(\xi\zeta)\, t = \xi(\zeta t)$. We shall assume that all substitutions are idempotent—a substitution σ is idempotent if and only if $\sigma\sigma = \sigma$. Substitutions must respect sorts, *i.e.*, if $\sigma x^s = t$ for a substitution σ, then t must be an instance of the sort s. A substitution ξ may be restricted to a subset of its domain. We write $\xi|_V$ to denote ξ restricted to V. By definition, $\xi|_V\, x = \xi x$ if $x \in V$, and $\xi|_V\, x = x$ otherwise. A *ground* substitution ξ is said to be *more general* than ξ' modulo R for a set V of variables (written $\xi \leq_R^V \xi'$) if and only if $\forall x \in V.\ \xi x \xrightarrow[R]{*} \xi' x$. The relation $\leq_R^V$ is reflexive and transitive but not necessarily antisymmetric.

The knowledge base, for the purposes of this section, consists of a set R of conversion rules of the form $t \longrightarrow u$, where $t, u \in \mathcal{T}_\Sigma(\mathcal{V})$. R generates the usual rewriting relation $\xrightarrow[R]{*}$, which is just the reflexive, transitive and congruence closure of R. In forming the congruence closure, sorts are assumed to be respected, *i.e.*, well-sortedness takes precedence over congruence.

The general problem we need to solve can be denoted by $t \xrightarrow[R]{?} u$, where $t \in \mathcal{T}_\Sigma$ is a ground term and $u \in \mathcal{T}_\Sigma(\mathcal{V})$ is a linear term (a linear term does not have more than one occurrence of any variable). A solution of such a problem is a ground substitution σ such that $t \xrightarrow[R]{*} \sigma u$. Such a substitution will be called an *R-matcher* for $t \xrightarrow[R]{?} u$. In the process of solving $t \xrightarrow[R]{?} u$, the problem typically gets reduced to a set $\{s_1 \xrightarrow[R]{?} t_1, \ldots, s_k \xrightarrow[R]{?} t_k\}$ of subproblems[9]. It is therefore convenient to consider such a set to be the general form of a reduction problem.

[9] Some of these problems may have terms with variables on both sides. See below.

The notion of an R-matcher extends to the set form naturally—a matcher for a set will solve *all* problems in the set simultaneously. We shall use P to denote such an arbitrary set of reduction problems. Let $M_R(P)$ denote the set of *all* R-matchers of P and $Var(P)$ the set of variables occurring in P. We say that $MM_R(P)$ is a *minimal complete* set of R-matchers for P if and only if

- $MM_R(P) \subseteq M_R(P)$
- $\xi \in M_R(P) \Rightarrow \exists \xi' \in MM_R(P),\ \xi' \leq_R^{Var(P)} \xi$
- $\forall \xi, \xi' \in MM_R(P),\ \xi' \leq_R^{Var(P)} \xi \Rightarrow \xi = \xi'$.

The three conditions can be thought of as *correctness, completeness* and *minimality* conditions respectively. Completeness and minimality also imply that the R-matchers in $MM_R(P)$ are *most general.*

The problem is to find an algorithm that finds a minimal complete set of R-matchers for a general problem P. Such an algorithm can be naturally expressed in the rule-based style of [10]. We give an algorithm in Table 1 that nondeterministically finds most general R-matchers subject to the assumption that the reduction theory is Ω-free, *i.e.*, if $f(t_1, \ldots, t_n) \xrightarrow[R]{*} f(u_1, \ldots, u_n)$ then $t_1 \xrightarrow[R]{*} u_1, \ldots, t_n \xrightarrow[R]{*} u_n$. The function **bridge** used in the algorithm is defined as follows. Given that R consists of the set of rules:

$$\begin{aligned} f_1(t_{11}, \ldots, t_{1m_1}) &\longrightarrow g_1(u_{11}, \ldots, u_{1n_1}) \\ &\vdots \\ f_k(t_{k1}, \ldots, t_{km_k}) &\longrightarrow g_k(u_{k1}, \ldots, u_{kn_k}) \end{aligned}$$

we construct the directed graph $\mathcal{G}_R = (\mathcal{N}_R, \mathcal{E}_R)$ with

$$\mathcal{N}_R = \{f_1, \ldots, f_k, g_1, \ldots, g_k\} \qquad \mathcal{E}_R = \{(f_i, g_i) : 1 \leq i \leq k\}$$

bridge(f, g, R) succeeds if and only if there is a path from f to g in $\mathcal{G}_R$ and in that case it returns the first rule $s \longrightarrow v$ along the path.

A formal proof of the correctness and completeness of the algorithm is boyond the scope of this paper. The following notes mention some of the ideas on which the algorithm is based, and explain the relationship between the constraints on R assumed above and the algorithm.

The algorithm is in the form of a set of rules for transforming the problem set P. The objective is to reduce P to a so-called *solved form* by repeated application of the rules. A solved form consists of units of the form $t \xrightarrow[R]{?} x^s$ where t is an instance of s. This can be interpreted as a substitution, especially since t is always a ground term and hence the substitution is idempotent.

We assume that P starts out as a set $\{t \xrightarrow[R]{?} u\}$ where t is ground. Although **Conflict** introduces nonstandard problem units $t \xrightarrow[R]{?} u$ where t is not ground,

Delete	$P \cup \{s \xrightarrow[R]{?} s\} \Rightarrow P$
Decompose	$P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} f(u_1,\ldots,u_n)\}$
	$\Rightarrow P \cup \{t_1 \xrightarrow[R]{?} u_1, \ldots, t_n \xrightarrow[R]{?} u_n\}$
Conflict	$P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} g(u_1,\ldots,u_m)\}$
	$\Rightarrow P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} s,\ v \xrightarrow[R]{?} g(u_1,\ldots,u_m)\}$, if **bridge** (f,g,R) returns $(s \longrightarrow v) \in R$
	F, otherwise (assuming $f \neq g$)
Instantiate	$P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} x^s\},\ f \in \Sigma_{w,s'},\ s' \not\leq s$
	$\Rightarrow P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} g(x_1^{s_1},\ldots,x_k^{s_k}),\ g(x_1^{s_1},\ldots,x_k^{s_k}) \xrightarrow[R]{?} x^s\}$, if $g \in \Sigma_{s_1 s_2 \cdots s_k, s''},\ s'' \leq s$ and **bridge** (f,g,R) succeeds
	F, otherwise
Propagate	$P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} x^s\},\ f \in \Sigma_{w,s'},\ s' \leq s$
	$\Rightarrow \{x^s \mapsto f(t_1,\ldots,t_n)\}P \cup \{f(t_1,\ldots,t_n) \xrightarrow[R]{?} x^s\}$,

Table 1: SC–MATCH: Rules for R-matching for Simple Conversion Problems

it is possible to ensure that there is always at least one unit of the standard form in P. The basic strategy is to always attack a standard unit with **Decompose**, **Conflict** or **Instantiate** until only **Propagate** is applicable. Applications of **Propagate** can then be used to reduce another unit to standard form, unless P is already in solved form. Note that the actual convertibility proof consists of all the conversion steps generated by uses of **Conflict**.

4 Related Work

Object imaging is recognized as a fundamental problem in the connectivity of distributed object systems, and the Uniform Data Transfer and Structured Storage services associated with COM as well as the Life-Cycle and Externalization

services in the Common Services layer on top of CORBA address the problem to a limited extent. We are not aware of any concrete proposal that goes beyond a simple externalize-transport-internalize model. An interesting overview of the imaging problem is given by Konstantas [11] in the context of a general discussion of interoperability.

The idea of using type isomorphism for representation transformations was used in our earlier work in the context of polymorphic functional languages [24], where the primary reasoning mechanism was equational unification. Although the unification problem for arbitrary equational theories is undecidable, and multiple most general unifiers create semantic problems, we were able to show that for a reasonably broad class of user-defined type equations, implicit conversions among multiple representations can be supported in practice even in the presence of ML-style type *reconstruction* since the corresponding equational theories are unitary [25].

In the broader context of object system interoperability, interface transformations are a related and complementary area of research. Object imaging is limited by the availability of object factories for internalization in the remote environment. Although the required functionality may be supported by the objects produced by the factory the exact interface required may not be supported. In such cases automated interface transforms can be used to produce proxy objects with the right interface. Pintado and Junod [20] discuss a scheme where such proxies are created manually. We were able to show that some recently developed algorithms for subtyping with recursive types [2] can be extended to largely automate the process of interface matching and synthesis of proxy objects [26].

Finally, there has been quite a bit of recent work in extending the very notion of object type to include interaction protocols [1, 12, 16, 28] which is likely to have an impact on object imaging by refining notions of object types and type matching.

5 Conclusions

We have presented an outline of the technical issues involved in an object imaging service, with the main emphasis on knowledge representation ideas. We have made no attempt to give a mapping of our ideas to a specific distributed object technology platform such as CORBA or COM, with detailed specification of service structure and interfaces, and relationship to other services and the basic object model. Such a mapping will result in a conversion registry for object types, and as we mentioned before, will have to deal with many systems issues such as distributed reasoning using knowledge from multiple registries, uniform scripting models for conversion scripts, activation protocols for standalone converters, and so forth. We hope to deal with these problems in another paper.

References

[1] R. Allen and D. Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the workshop on interface definition languages*, 1994. Published as School of CS, Carnegie Mellon University technical report, CMU-CS-94-WIDL-1.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of Eighteenth POPL Symposium*, pages 104–118. ACM Press, January 1991.

[3] Apple Computer, Inc. *OpenDoc Technical Summary*, October 1993. Version 1.0.

[4] Kraig Brockschmidt. *Inside OLE 2.* Microsoft Press, 1994.

[5] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, U.K.* ACM Press, Addison-Wesley, 1989.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), 1985.

[7] D.E. Cohen, K. Madlener, and F. Otto. Separating the intrinsic complexity and the derivational complexity of the word problem for finitely presented groups. *Mathematical Logic Quarterly*, 39:143–157, 1993.

[8] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[9] William Harrison. *The Importance of Using Object References as Identifiers of Objects.* Object Management Group, 1994. OMG Document 94–6–12.

[10] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

[11] Dimitri Konstantas. Object oriented interoperability. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP 93)*, pages 80–102. Springer-Verlag, 1993. LNCS 707.

[12] Doug Lea and Jos Marlowe. PSL: protocols and pragmatics for open systems. Manuscript, August 1994.

[13] J. Meseguer, J.A. Goguen, and G. Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8(4):383–413, 1989.

[14] Bertrand Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[15] Microsoft Corp. and Digital Equipment Corp. *Common Object Model Specification (Introduction), Draft Version 0.2*, October 1994. OMG Document 94.10.9.

[16] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the Eighth Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 93)*, pages 1–15. ACM Press, 1993. Special Issue of SIGPLAN Notices 28(10).

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1993. OMG Document 93–12–43 Revision 1.2.

[18] Object Management Group. *Common Object Services Specification, Volume I*, 1994. OMG Document 94.1.1 (includes specifications for Naming Service, Lifecycle Service, and Event Notification Service).

[19] Object Management Group. *Object Externalization Service*, 1994. Joint Proposal by IBM and Sunsoft, OMG TC Document 94.6.21.

[20] Xavier Pintado and Betty Junod. Gluons: support for software component cooperation. In D. Tsichritzis, editor, *Object Frameworks*. Centre Universitaire d'Informatique, Université de Genève, 1992.

[21] Michael Powell. *Objects, References, Identifiers and Equality White Paper*. Object Management Group, 1993. OMG Document 93–7–5.

[22] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-sorted equational computation. In M. Nivat and H. Aït-Kaci, editors, *Resolution of equations in algebraic structures, Volume 2*. Academic Press, 1989.

[23] Sun Microsystems. *Network Programming Guide*, 1990.

[24] Satish R. Thatté. Coercive type isomorphism. In *Proceedings of the Fifth Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, pages 29–49. ACM Press, 1991.

[25] Satish R. Thatté. Finite acyclic theories are unitary. *Journal of Symbolic Computation*, 15(2), February 1993.

[26] Satish R. Thatté. Automated synthesis of interface adapters for reusable classes. In *the Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*. ACM Press, 1994.

[27] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of Second European Symposium on Programming*. Springer-Verlag, 1988. LNCS 300.

[28] Daniel M. Yellin. Interface, protocols, and the semi-automatic construction of software adaptors. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 94)*. ACM Press, 1994.