Pragmatic Type Interoperability*

Sébastien Baehni[a] Patrick Th. Eugster[a] Rachid Guerraoui[a] Philippe Altherr[b]

[a] Distributed Programming Laboratory
[b] Programming Methods Laboratory
Swiss Federal Institute of Technology in Lausanne

Technical Report IC/2003/08

Abstract

Providing type interoperability consists in ensuring that, even if written by different programmers, possibly in different languages and running on different platforms, types that are supposed to represent the same software module are indeed treated as one single type. This form of interoperability is crucial in modern distributed programming.

We present a pragmatic approach to deal with type interoperability in a distributed system. Our approach is based on an optimistic serialization mechanism and a set of implicit type conformance rules. We experiment the approach over the .NET platform which we indirectly evaluate.

Keywords: Distributed Programming, Types, Objects, Events, Interoperability, Conformance, Serialization, Middleware, Dynamic Proxies, .NET.

1. Introduction

Context. There are different forms of interoperability and these differ according to their abstraction level. *Interoperability at the hardware level* is typically about devising an operating system, e.g., Linux, that runs on different machines, e.g., Pcs, Laptops, Pdas, Macs. *Interoperability at the operating system level* ensures that the programming language, e.g., Java through its bytecode and virtual machine, is independent from the underlying operating system, e.g., Linux, Unix, Windows, MacOS. *Interoperability at the programming language level* guarantees that a class written in a specific language, e.g., C++, can be used in another language, e.g., Java, transparently. This is for instance what .NET aims at offering.

This paper focuses on an even higher level of interoperability: *type interoperability*. The goal is to make transparent for the programmer the use of one type for another, even if these types do not exactly have the same methods or names, as long as they aim at representing the same software module. These types might be written in the same language but by different programmers, they might be written in different languages, or even running on different platforms.

We address the issue of type interoperability in a distributed system where new types of messages are sent/received dynamically (more particularly, we are thinking of pass-by-value objects). Typically different software modules need to be assembled in a distributed application. Some of these modules might aim at representing a single logical entity.

*The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

Motivation. Type interoperability was first introduced in centralized applications [LBR96]. However, as presented in Section 2, those solutions are too rigid to apply them as is in a dynamic distributed environment. Our aim is to provide a transparent solution to this problem in a distributed environment. Basically, we are interested in devising a pragmatic scheme to allow objects of different types, that aim at representing the same module, to be remotely exchanged (not only pass-by-reference, but especially also pass-by-value semantics) as if they were of the same type, even if these types (a) have different methods or names, (b) are written in different languages, or (c) running on different platforms. The challenge here is to provide this transparency with acceptable performance.

Contributions. This paper presents a general approach to deal with type interoperability in a distributed setting. We focus on the representations of the types and we use an optimistic transport protocol (that saves network ressources) as well as different serialization mechanisms to guarantee the efficiency of type comparison between transferred objects. To experiment our approach in a concrete setting, we have implemented it over a popular object-oriented platform: .NET[1]. This platform has been chosen because it provides language interoperability. That is, it provides the highest level of interoperability "underneath" type interoperability. We extend .NET to allow for type conformance and we provide associated structural conformance rules, themselves implemented via .NET dynamic proxies. Our approach requires a small overhead for invoking an object, though locally invoked, received from a remote host and we precisely measure this overhead through our prototype implementation. Indirectly, we evaluate the .NET serialization (binary, XML (*eXtended Markup Language*) and SOAP (*Simple Object Access Protocol*)) together with its reflection capabilities.

Roadmap. Section 2 relates our work to other work on type interoperability in a centralized context. Section 3 overviews the problem of type interoperability in a distributed environment and our approach to solve it. Section 4 presents our type conformance rules. Section 5 describes how types are represented and Section 6 presents our mechanisms for serializing objects. Section 7 gives a brief background on .NET and describes how we have implemented our type comparison mechanism in the .NET platform. Section 8 gives some performance measurements. Finally, Section 9 draws some conclusions on our use of .NET and describes some applications that can make use of type interoperability.

2. Related Work

2.1. Safe structural conformance for Java

Type interoperability was addressed for a centralized context in [LBR96] through structural conformance. The structural conformance was defined as follows: "Any class or interface that declares (or implements) each method in a target interface *conforms structurally* to the interface, and any expression of the source class or interface type can be used wherever a value of the target interface type is expected". To that end, several rules for structural conformance were defined. However these rules are based on the Java type hierarchy (a type is conformant if it implements each method in a target interface) which narrows the scope of structural conformance. Moreover only types that are tagged as being structural conformant can pretend to do so, meaning that legacy interfaces can never be used with structural conformance. Our approach has the aim to extend the structural approach in a decentralized environment such that structural conformant types do not need to share the same type hierarchy and neither need to be tagged as being structural conformant enabled.

2.2. Compound types for Java

Compound types for Java [BW98] aim at simplifying the composition and reusability of Java types without having to change them or agree on a common design. A new way to express a type was introduced:

[1] Of course the choice of the .NET platform implicitly fixes the operating system (Windows^{TM}) and runtime environment (*common language runtime*–CLR) respectively, while the set of programming languages is fixed through our choice of supporting only those supported by .NET. However, our approach could be implemented in another platform like CORBA or Java RMI.

[TypeA,TypeB,...,TypeN]. This new notation defines all the types declared to implement *TypeA*, *TypeB*,..., *TypeN*. With compound types the programmer can express a "kind" of structural conformance as we take into account the implemented methods of a type instead of only its name. However these compound types are more about composition than about structural conformance and making type interoperable.

2.3. Corba

CORBA [OMG01] addresses the language interoperability problem through an *interface definition language* (IDL) that unifies different programming languages. This IDL provides support for pass-by-reference semantics which make it possible to call a specific method from one language to another. Pass-by-value semantics for object types have been added to CORBA quite recently through *value types* to enable the passing of invocation arguments [OMG01]. The adopted solution is rather tedious to use, as developers are required to implement such types in all potentially involved languages. In particular, this makes it hard to add value (sub)types with new behavior at runtime. Even if this mechanism could be used for solving the type interoperability problem, to our knowledge, the problem has never been addressed in CORBA. Note that CORBA implementations provide various mechanisms, such as the *dynamic skeleton interface* and *dynamic invocation interface*, but also the concept of *smart proxies* found in many ORB implementations, which enable to some extent the realization of implicit structural conformance. Pass-by-value semantics with object types would however be strongly limited, as pointed out in Section 5.

2.4. Java RMI

Java RMI enables the transfer of objects by value as arguments of remote invocations, thanks to its built-in serialization mechanism. By virtue of subtyping, an instance of a new class can be used as invocation argument, provided that it conforms to the type of the corresponding formal argument. By transmitting the corresponding class (byte code) to an invoked object previously unaware of that class, one can implement a scheme where new event classes are automatically propagated. The underlying dynamic code loading and linking ensured by the Java virtual machine would also make it possible to extend/alter the behavior of existing resource types at runtime. Though the Java virtual machine has been used to run code written in various languages, the exploiting of its type safe dynamic code loading and linking [LB98, Sun99] is problematic outside of Java. Like CORBA, this dynamic linking mechanism could be used for implemeting type interoperability, but again, to our knowledge, type interoperability has never been implemented in Java RMI.

2.5. Microsoft .NET

Just like CORBA, .NET aims at unifying several object-oriented languages through a *common type system* (CTS). The advandage here is that the programmer does not need to reimplement the type of interest in all the programming languages in order to use the pass-by-value semantics. Nevertheless, .NET does not address the issue of transparently unifying types that are not identical but that are aimed at representing the same module, i.e., types which conform to each other implicitly (see section 4 for our definition of implicit conformance).

2.6. Renaissance

The *Renaissance* system [MR95] implements an interesting RPC scheme where types with different methods or names can be invoked as if they were the same type, as long as they conform implicitly to each other. The idea is based on *structural conformance* rules as means to compare such types. The approach is however limited in that it relies on an explicit type definition language called *lingua franca* (even though mainly for the purpose of generating typed proxies), and does not support pass-by-value semantics with object types. Our approach for type interoperability has the aim of not being bound to any intermediate language but rather to the type system of the platform itself. Moreover, our approach focuses on pass-by-value semantics as well as pass-by-reference semantics.

3. Overview

This section overviews the problem of type interoperability in a distributed environment and our approach to address it.

3.1. The problem

Usually, types are implemented either through interfaces or classes. Let us imagine a type `Person` that has a field `name`. A first programmer can implement this type with a setter method named `setName()` and a getter method named `getName()`. Another programmer can implement the same type with the following setter and getter respectively: `setPersonName()` and `getPersonName()`.

Clearly even if the two implementations provide the same functionalities, they are not compatible with each other, i.e. the programmers cannot use the two implementations transparently. In "static" environments (where all the types of objects are known at the start of the system) this problem is easy to solve because one can a priori hardcode the translation rules in the system.

However, when the system is "dynamic", i.e. where new events of new types can be put into the system at runtime, this problem is not trivial anymore. To solve it one must create a set of general rules that can be compatible with every type and implement these rules into the middleware. This implementation must be compatible with the pass-by-reference but also with the pass-by-value semantics in order to achieve full distributed interoperability.

3.2. Our approach

We implement the general rules needed for ensuring type interoperability as well as a set of corresponding serialization mechanisms. A distributed and "dynamic" environment is assumed. We do not tackle the problem in a local setting, because it raises static type safety issues that are difficult to resolve without proving the type soundness of the solution which is not the aim of this paper. The general protocol to achieve type interoperability in a distributed environment is depicted in Figure 1.

When the middleware receives an object, it tries to check for the type information of this object. Once the middleware obtains the type information it can check for type conformance with respect to types of interest. If the check is successful, the code of the object is downloaded in order to deserialize the object. The object can then be used as if it were of the type of interest.

The general protocol of Figure 1 can be decomposed in three distinct subprotocols, namely (1) object serialization, (2) type description creation and (3) type conformance checking. This paper describes the general case and then focuses on an implementation using the .NET platform. This implementation gives a practical viewpoint as well as an experiment of making our rules more concrete.

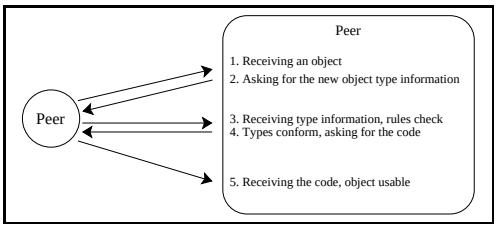

Figure 1. General protocol used to ensure conformance between two types

4. Type Conformance

This section presents our type conformance rules. We first make a classification of the different categories of conformance and then we give our conformance rules.

4.1. Conformance categories

We give here a complete classification of the different categories of conformance. It explains the different categories and recall the existing terminology.

The first category is *hardware conformance* that aims at devising an operating system to work on different computers. *Operating system conformance* ensures that the programming language is independant from the underlying operating system. Another category, now provided by the .NET platform allows to use a type described in one programming language (C# for example) in another language (VB.NET). We call this category *language* conformance.

The next category is called *type conformance* and focuses on the interoperability between types. This category contains two subsets called *implicit structural* type conformance and *implicit behavioral* type conformance. Implicit structural type conformance encompasses what we call *explicit* type conformance. Namely, explicit type conformance take into account the type hierarchy to which a type belongs, i.e. subtyping issues. The combination of the implicit structural type conformance and the implicit behavioral type conformance results in a "strong" *implicit* type conformance.

The implicit behavioral type conformance is based on the behavior of the type, i.e., based on the result of its methods. This type of conformance is very difficult to analyse in the sense that the body of the methods cannot just be compared but these methods must also be executed in order to compare their results for corresponding inputs. That should be feasible for types dealing only with primitive types but for more complex types it is not so trivial. Finally, the implicit structural conformance (the one this paper is about), strictly relies on the structure of the type. By structure, we mean the type name, the name of its supertypes, the name and the type of its fields and the signature of its methods[2].

In this paper we focus on implicit structural type conformance only (and for simplicity purpose we will denote, in the following of this paper, implicit structural conformance instead of implicit structural type conformance). Figure 2 summarizes our classification of the type conformance.

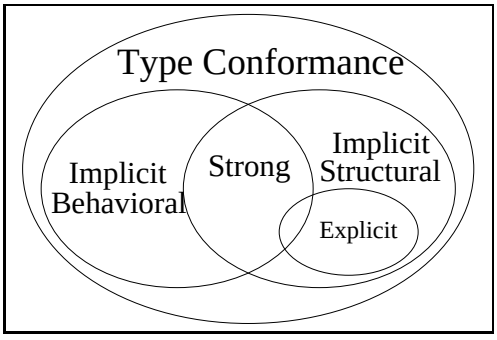

Figure 2. The type conformance classification

4.2. The type conformance rules

In the following, we present the type conformance rules. We first introduce several basic notations and definitions that will help us to explain the different aspects of conformance. Finally we present the implicit structural conformance rule.

4.2.1. General definitions and notations

To make things clearer, and in order to be able to describe the different aspects making all together the implicit structural conformance rule, several terms are defined. Figure 3 presents those terms, notations and the implicit structural conformance rules. First some notations that are used in the rules are defined. Then

[2] Please note that in the literature, *structural* conformance has been studied in [LBR96] and is a mixin between what we call explicit type conformance and implicit structural type conformance.

a definition of the general conformance rules is given[3]. The second definition describes the equality of two types. The third definition explains the equivalence between two types. The fourth and the fifth definitions denote the notation for the superclass and the interfaces of a certain type. The sixth definition defines the *name*() method used in the conformance rules. Finally Figure 3 presents the implicit structural conformance rules.

$$
\begin{array}{l}
\textit{Notations}: \\
\quad T \text{ denotes a type } T \\
\quad T_{GUID} \text{ denotes the globally unique identifier of type } T \\
\quad m \text{ denotes a method } m \\
\quad cons \text{ denotes a constructor} \\
\\
\textit{Conformance}: \\
\quad T \leq_E T' \Rightarrow T \leq T' \\
\quad T \leq_I T' \Rightarrow T \leq T' \\
\quad T \leq_{Is} T' \Rightarrow T \leq_I T' \\
\\
\textit{Equality}: \\
\quad T == T' \ iif \ T_{GUID} == T'_{GUID} \\
\\
\textit{Equivalence}: \\
\quad T \leq T' \wedge T' \leq T \Rightarrow T \equiv T' \text{ (case for } \leq_E \text{ iff T'==T)} \\
\\
\textit{Superclass}: \\
\quad T^{super} \equiv (T' \mid T' \ superclassof \ T) \\
\\
\textit{Interfaces}: \\
\quad T^{inter} \equiv (\{T' \mid T' \ interfaceof \ T\}) \\
\\
\textit{Name}: \\
\quad (name(x) \mid x \in \{T, m, cons\}) \equiv name \ of \ x \\
\quad\quad (as \ a \ case \ insensitive \ string \ representation)
\end{array}
\qquad
\begin{array}{l}
\textit{Name conformance } (i): \\
\quad T \leq_{Is}^{name} T' \Rightarrow LD(name(T), name(T')) = 0 \\
\\
\textit{Field conformance } (ii): \\
\quad T \leq_{Is}^{field} T' \Rightarrow \forall f' : T_{f'} \in T' \ \exists f : T_f \in T \mid T_f \leq_{Is} T_{f'} \\
\\
\textit{Supertypes conformance } (iii): \\
\quad T \leq_{Is}^{hier} T' \Rightarrow (T^{super} \leq_{Is} T'^{super} \wedge T^{inter} \leq_{Is} T'^{inter}) \\
\\
\textit{Method conformance } (iv): \\
\quad T \leq_{Is}^{meth} T' \Rightarrow \forall m'(Perm(a_{1'} : T_{1'}, ..., a_{n'} : T_{n'})) : T_{r'} \in T' \\
\quad \exists m(Perm(a_1 : T_1, ..., a_n : T_n)) : T_r \in T \mid \\
\quad name(m) == name(m') \wedge \\
\quad \forall i \in [1, n](T_{i'} \leq_{Is} T_i) \wedge T_r \leq_{Is} T_{r'} \\
\\
\textit{Constructor conformance } (v): \\
\quad T \leq_{Is}^{cons} T' \Rightarrow \forall cons(Perm(a_1 : T_1, ..., a_n : T_n)) \in T \\
\quad \exists cons(Perm(a_{1'} : T_{1'}, ..., a_{n'} : T_{n'})) \in T' \mid \\
\quad name(cons) == name(cons) \wedge \forall i \in [1, n](T_{i'} \leq_{Is} T_i) \\
\\
\textit{Implicit structural conformance } (vi): \\
\quad T \leq_{Is} T' \Leftrightarrow (T \leq_{Is}^{name} T' \wedge T \leq_{Is}^{hier} T' \wedge T \leq_{Is}^{field} T' \wedge \\
\quad T \leq_{Is}^{meth} T' \wedge T \leq_{Is}^{cons} T') \vee T == T' \vee T \leq_E T'
\end{array}
$$

Figure 3. Our rules of conformance together with the general definition

4.2.2. Decomposing implicit structural conformance

We define different aspects of conformance as follows:

Name (i): This aspect takes into account only the name of the different types to compare to. A name of a type T is said to conform to the name of a type T' if the names are the same (i.e. the Levenshtein distance (LD) [Lev65] is equal to 0). The names are considered to be case insensitive. In order to be more general, wildcards could be allowed but this is not the aim of this paper.

Fields (ii): A field f of type T_f ($f : T_f$) defined in a type T is said to conform to a field f' of type $T_{f'}$ defined in a type T' if T_f and $T_{f'}$ are implicitly structurally conformant.

Supertypes (iii): This aspect takes into account the supertypes of the type and its interfaces (if any)[4]. A type T is said to conform to a type T', with respect to T''s type hierarchy, if the superclass and the interfaces of T conform respectively, in the implicit structural sense, to the superclass and the interfaces of a type T'. T^{super} and T^{inter} denotes the superclass and the set of interfaces of type T respectively.

Methods (iv): Conformance between methods is a bit more tricky than the other aspects of conformance. First, the modifier of the methods are supposed to be the same (this assumption is implicitly assumed in the rule). Then, in order to describe the corresponding rule, three parameters for each method are taken into account: first the name of the method, second the arguments of the method and finally the return type of the method. To understand this rule, one must think of which uses the (1) return parameter and the (2) the arguments of the method: the instance of the type expected to be received (depicted as the "real" object) or the object received that must implicitly structurally conforms (depicted as the implicitly structurally conformant object). In (1) the "real" object uses the return parameter, meaning that the

[3]Implicit conformance is noted $\leq_I$, while explicit conformance is noted $\leq_E$and the implicit structural conformance is noted: $\leq_{Is}$. Finally, $T \leq T'$ denotes the fact that instances of T can be used safely whenever an instance of T' is expected.

[4]The distinction between the type and its supertypes is done in order to make things clearer.

return parameter T_r of the method m must implicitly structurally conform to the return parameter $T_{r'}$ of the method m'. In (2), the implicitly structurally conformant object uses the parameter given by the "real" object. In this case, the argument $T_{i'}$ of the method m' must implicitly structurally conform to the argument T_i of the method m. Note that the permutations of the arguments of the methods (denoted by $Perm(a_1, a_2, ..., a_n)$) are taken into account.

In other terms, contravariance is acceptable for the arguments of the methods and covariance is acceptable for the return values. Indeed, assuming contravariance for the arguments means that if the user uses the methods of the conforming object with its own parameters, these parameters must be acceptable for these methods. As the type of the parameters are static (in the sense that they are fixed by the user), the methods of the object to conform to must either accept these types or supertypes of them. The return types can deal with covariance. Indeed, if the user is expecting an object of type T, and receives an instance of a subtype of it, there is no problem, but if it receives a supertype, he might try using a specific method of T that the supertype does not define.

Constructor (v): The final step before defining the implicit structural conformance rule is to describe the conformance rule for the constructors. This rule is quite the same as the one for the methods except that there are no return values (hence no return type). Again, contravariance for the arguments of the constructors is assumed.

4.2.3. The implicit structural conformance rule (vi)

Implicit structural conformance ($\leq_{Is}$) can now be described. A type T implicitly structurally conforms to a type T' (in a complete way) iif T conforms to type T' in all the aspects defined before or if T and T' are equivalent or if T conforms explicitly to T'.

The word "complete" is introduced here for implicit structural conformance for the first time because one could think of having a more weaker (incomplete) rule taking into account only the name of the types for example. However, not taking into account the whole set of aspects breaks the type safety and result in receiving an error while trying to call a specific method onto the object.

What if a field, a method or a constructor of a type T match several fields, methods or constructors of a type T' of which it implicitly conforms (e.g., a method with a single argument x_1 of type T_1 in T can match an arbitrary number of methods with a single argument $x_{1'}$ for as long as $T_1 \leq T_{1'}$)? In this case, the rules does not impose any criterion, it is up to the programmer to decide what is more suitable.

5. Type Representation

This section discusses the representation of types. Our objective is to make the comparison between two types possible, according to the rules described in Section 4, without having to transfer the implementation of them. To achieve this goal, we rely on introspection mechanisms (that are provided in platforms like Java or .NET).

5.1. Overview

Once the object, as well as a handle to its type description (see Section 6), are received on a given host, a test must be performed to check if this object can be used as is within a given variable. This means that the type description of the received object must conform to the type of the variable. Downloading directly the package/assembly containing the type of the object is not an option, because this would consume too many network and memory resources, especially if it appears that the types do not conform. For that reason, only a type description is downloaded.

To create such a type description, the reflection mechanisms of the object-oriented platform are used as a basics, as they provide some useful mechanisms that help in order to achieve our goal. Those reflection classes help us to get information about the variables, the methods and the attributes of the type to represent.

5.2. Our approach

This section shows how types are represented in order to be effeciently serialized and sent through the network.

5.2.1. Types as XML messages

Types in our system are represented as XML structures. One obtains the information necessary to "construct" a type description by means of introspection. Such a type description includes explicit supertype information as well as signatures of methods, attributes, and type *identity*[5].

Recall that the serialization mechanisms of the main object-oriented platforms we think of (.NET or Java) are not able to serialize/deserialize an object (even its reflection fields) without knowing in advance its type. For that reason our own introspection for representing fields, methods, constructors, interfaces and superclass of objects need to be created and serialized. Figure 4 represents our general description of a type. In order to create such instances of our introspection classes, one must rely on the introspection classes of the chosen object-oriented platform (see Figure 6).

Since it is not feasible (for resource reasons) to send these introspection objects trough the network, one by one, a special type (called `TypeDescription`) which implements the `ITypeDescription` interface (see Figure 5) is introduced. The `ITypeDescription` interface presents the methods necessary to acquire the information about the type of the object to serialize. Two specific methods (`equals()` and `conforms()`) are defined and are used to test the conformance between types. In order to serialize a new `TypeDescription` object a basic XML serialization mechanism is sufficient.

As mentioned before, the `TypeDescription` class gives a description of the type it reflects (i.e. its fields, methods including the arguments of the methods, constructors, etc). But there is no description of the fields, methods, ... of the types of the formal *arguments* of the methods or of the fields themselves. There is no recursion in the type description for two main reasons, namely (1) for saving time during the creation of the XML message and (2) for keeping this message small because a subtype description might already be available at the receiver side, so there is no need to transport redundant information. Figure 7 gives a XML description of the `Person` class presented in Figure 9. This XML description has been shortened for space reason.

```
public class TypeInfo
{
  public string attributes;
  public string typeName;
  public string assemblyName;
  public string downloadPath;
  public string GUID;
  public TypeInfo(string attributes,string typeName,
                  string assemblyName,string downloadPath,
                  string GUID) {...}
}
```

Figure 4. `TypeInfo` **introspection class**

```
public interface ITypeDescription
{
  ClassInfo[] getClassInfo();
  ConstructorInfo[] getConstructorsInfo();
  MethodInfo[] getMethodsInfo();
  SuperClassInfo[] getSuperClassesInfo();
  InterfaceInfo[] getInterfacesInfo();
  FieldInfo[] getFieldsInfo();
  bool conforms(ITypeDescription itd);
  bool equals(ITypeDescription itd);
}
```

Figure 5. `ITypeDescription` **interface**

6. Object Serialization

In this section, we elucidate how objects are represented, conveyed (pass-by-value semantics), and invoked (pass-by-reference semantics) between components in our approach. We first introduce the different issues addressed in this section and explain our pass-by-value and pass-by-reference approaches.

[5] We rely on the concept of type identity provided by the underlying platform. As a matter of example, .NET provides globally unique identifiers (GUID) of 128 bits long for types.

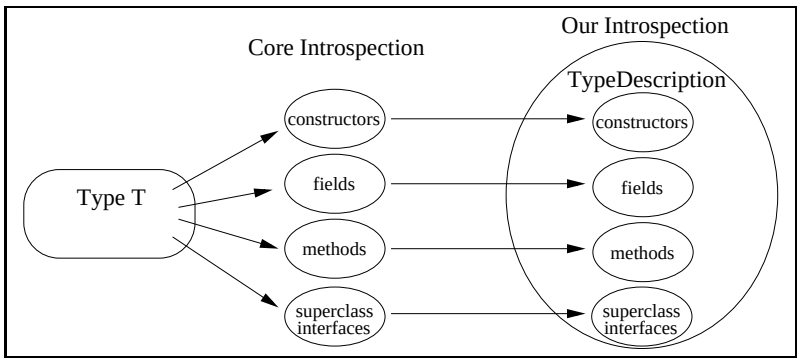

Figure 6. Introspection mechanism

```
<?xml version="1.0"?>
<TypeDescription ..>
  <class>...</class>
  <superClasses>...</superClasses>
  <fields>...</fields>
  <methods>...</methods>
  <constructors>...</constructors>
</TypeDescription>
```

Figure 7. XML description of `Person`

6.1. Overview

For the same reasons presented in Section 5 it is also not feasible to send the type representation of the object with the object itself. This is because it might happen that objects of the same type have already been received before and there is no need to download again the type representation of the object. For that reason, when an object is sent through the network it is sent only with a description of the download path where to get the complete type representation of it.

6.2. A XML-SOAP approach

This section presents the approach we have chosen to send/use an object through the network. Our approach is hybrid in the sense that to send objects, we rely on a combination of the XML and some other serialization mechanisms (SOAP or binary). For connecting objects remotely, our approach assumes and uses the remoting mechanisms of the chosen object-oriented platform.

6.2.1. Pass-by-value semantics

As presented above, it is not possible to serialize an object and send it through the network just as is. This is because the receiver of the object may not have the necessary information used to deserialize the object (if this is the first time he receives the object). To prevent knowing the type of the object sent, specific serialization and deserialization mechanism are used. That is, a XML message encompassing the object is sent instead of only the object itself. This XML message consists of information about the types of the object (type names and download paths of their implementations) and includes the SOAP or binary serialized object.

When such a XML message is received, it is deserialized in order to get the corresponding type information. A check is done in order to know if the corresponding classes or interfaces implementing the types are locally available. If this is the case, the deserialization of the object can be easily achieved. Otherwise, the type description of the object must be downloaded with the help of the information of the download paths. If the type of the object and the type of interest conform, the different classes and interfaces that implement the types can be downloaded and loaded into the memory in order to deserialize cleanly the object. In order

to deal with such conformant objects, dynamic proxies concept will be used (see next section for further information).

Figure 8 illustrates the serialization of an object of type `A` containing an object of a type `B`, while Figure 9 depicts the type `Person` (that is a subtype of interface `IPerson` for some reasons explained in Section 7) and `Name` (a field of `Person`). Finally, the corresponding XML message is described in Figure 10[6].

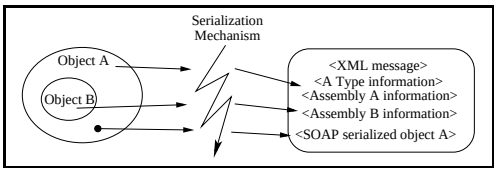

Figure 8. A hybrid serialization scheme

```
public class Person : IPerson {
  private Name name;
  private int secureID;
  public Person(Name name, int secureID) {...}
  public Name getName() {...}
  ...
}
public class Name {
  private string firstName;
  private string lastName;
  public Name(string firstName, string lastName) {
       ...
  }
  ...
}
```

Figure 9. `Person` and `Name` class

```
<?xml version="1.0" encoding="utf-8"?>
<ObjectDescription>
  <TypeInformation> ... </TypeInformation>
  <SOAPInformation>
      <SOAP-ENV:Envelope ...>
        <SOAP-ENV:Body>
          <a1:Person  ...>
            <name .../>
            <secureID ...>22</secureID>
          </a1:Person>
          <a1:Name ...>
            <firstName ...>Seb</firstName>
            <lastName ...>Baehni</lastName>
          </a1:Name>
          ...
        </SOAP-ENV:Body>
      </SOAP-ENV:Envelope>
  </SOAPInformation>
  </ObjectDescription>
```

Figure 10. XML representation of `Person`

6.2.2. Pass-by-reference semantics

Though the main object-oriented platforms (e.g. .NET), already provide several mechanisms for pass-by-reference semantics (.NET *remoting*), they are in our case, just like basic serialization mechanisms, not usable as such. Indeed, the current implementation can not be applied straightforwardly, due to our desire for interoperability not only at the programming language level, but also at the type level, meaning that we want to be able to provide some flexible form of type conformance. Imagine a component querying a type T_B, and T_B happens to match a lent remote server's type T_L *implicitly* (only), i.e., T_L is not a subtype of T_B. The invocation of T_B can not be performed straightforwardly on a remoting proxy; the interposing of a *dynamic proxy* (see Section 7) as a wrapper is necessary since T_B and T_L are not explicitly compatible. This mismatch increases with the depth of the matching of the two types T_B and T_L (requiring similar wrappers on the sharing component as well). This concept of dynamic proxies is available in object-oriented platforms like .NET by extending the the `RealProxy` class (even in Java by extending the `java.lang.reflect.Proxy` class and in using the `java.lang.reflect.InvocationHandler` interface). Our pass-by-reference approach is, in fact, quite the same as the pass-by-value approach, in the sense that for both approaches, the concept of dynamic proxies is massively used. The only difference between the two approaches is that for the pass-by-value approach our own serialization mechanism and dynamic proxies are used and for the the pass-by-reference approach, the basic remoting mechanisms enhanced with dynamic proxies (see Section 7 for a description of how to use dynamic proxies with .NET) are used.

[6]For presentation purpose the object has been serialized using the SOAP format.

7. Implementation issues

The concepts and rules presented in Section 4, Section 5 and Section 6 have been implemented on the .NET platform.

We have chosen this platform mainly because it provides language interoperability. However, relying on such a platform brings some limitations in terms of genericity (i.e. current lack for templates) and operating system interoperability (.NET has been designed to run on Windows). For the first issue, a very new *Gyro* "patch" has been released in order for .NET to be able to support genericity for the C# language [KS01]. However, even this patch does not allow to use genericity for the other languages supported by .NET, meaning that the language interoperability is broken for the types using such a patch. Concerning the second issue, some efforts have been done to port .NET to Linux. An open-source project, called *mono* (www.go-mono.com) has been launched.

The mechanisms that help us to implement the different concepts as well as the new type representation and the new implicit structural conformant rules (narrowed to the .NET needs) are presented. This section finishes with the explanations how such mechanisms are used to implement our concepts and rules.

7.1. Background: Overview of .NET

This section gives a brief overview of the .NET architecture as well as an explananation of the different components .NET is made of. We also present the serialization, networking and dynamic proxies mechanisms as our prototype was built on top of these.

7.1.1. .NET architecture

.NET [TL01] is Microsoft's framework for unified services and application development, mainly on the Windows platform. Figure 11 gives an overview of the .NET architecture.

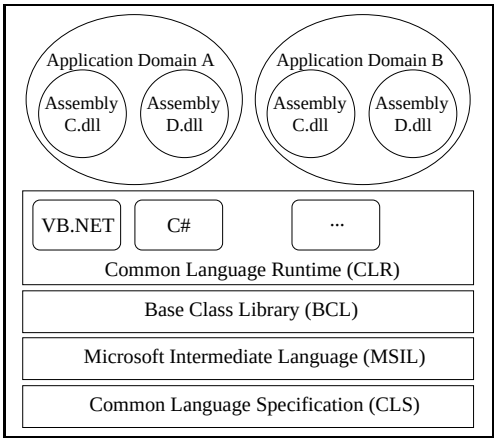

Figure 11. Overview of the .NET platform

7.1.2. Common language runtime (CLR)

The CLR is the soul of .NET. It is its runtime, comparable to the Java virtual machine, which is layered between an operating system and .NET applications. The CLR loads application code, manages it, runs it, and provides a number of support services. Some of these vital support services include resource management, thread management, remoting, as well as enforcing of code safety and security constraints. Code that is loaded and running under the control of the CLR is referred to as *managed code*.

7.1.3. Intermediate language (IL)

Compiled code in .NET does not contain assembly language instructions. Rather, code is compiled into assemblies that contain declarations in *intermediate language* (IL)[7]. Together with *meta-data* (used to provide a generic description of the types), this managed code makes up the abstract intermediate representation of a .NET application. IL is hence a low level language, similar in idea to Java byte code, i.e., represented as a series of opcodes. The IL is *always* JIT-compiled into native machine code.

7.1.4. Common type system (CTS)

If IL represents a general code format, able of expressing a multitude of languages, the CTS represents a general type system, able of capturing type schemes of these languages. CTS is currently limited to single class inheritance (though multiple subtyping can be achieved in interfaces), and is not able of representing *genericity*.

The CTS is divided in two categories: value types and reference types[8]. Value types directly contain their data. These value types are traditionally known as "primitive types" (like `int` in Java). In .NET, it is possible to create new value types. To this end, a new type must inherit from the `System.ValueType` type.

In contrast, reference types maintain a reference on the value's memory address. This kind of types are either interface types, pointer types or self-describing types. The latter type can be divided into several categories: arrays, user-defined classes, boxed value types and delegates.

7.1.5. Common language specification (CLS)

IL and CTS have been designed to express a variety of languages. The CLS on the other hand is simply a specification that defines the rules, basically a set of features that can be supported by all languages, which ensure that compliant IL code fragments can be executed in any language. These rules (over 40) are documented in [Mic02]. Roughly, a CLS-compliant definition (type, type member, or entire assembly) is tagged as such if its constituents (e.g., members of a type, formal arguments of a method, types in an assembly) visible outside the assembly are CLS-compliant, and it references itself only other compliant definitions. Namely, the `CLSCompliantAttribute` is set to `true` in an assembly definition if this is ensured. An assembly which is not tagged as so is, by default, not CLS-compliant.

7.1.6. Assemblies

IL declarations are regrouped in *assemblies*. They can be viewed as collections of types and can work together to create a logical unit of functionality. An assembly gives to the CLR the necessary information about the type(s) it owns. From the runtime point of view, a type does not exist outside an Assembly. An assembly is responsible for the security access, the versioning, and the reference scope of the type(s) it contains. Assemblies can be (1) static, or (2) dynamic. Static assemblies are compiled and saved to disk into *portable executable* (PE) files or *dynamic link libraries* (DLL). Dynamic assemblies are loaded directly into the memory and are not saved to disk (the `System.Reflection.Emit` package provides this functionality to the programmer). Assemblies are the successors of the well-known *dynamic link libraries* (DLL, known through the typical file postfix, e.g., *mylib.dll*) and of the conventional windows executable files. The type of the assembly (DLL or executable) depends on whether it contains an entry point (main function). In this last case, the assembly is a PE, otherwise it is an DLL. Each assembly has its own version number, meaning that two assemblies can have the same name.

The assemblies are constitued of four parts: an assembly manifest (containing metadata), type metadata, IL and resources (pictures, files, ...). Metadata are used to make programming language interoperability easier as it is essentially supplementary information about data types. Each assembly consists generally of several so-called *modules* which in turn consist of several type declarations.

[7] Sometimes also termed Microsoft IL (MSIL) or Common IL (CIL).

[8] Which have however nothing to do with our previous mentions to pass-by-value and pass-by-reference semantics in remote interactions.

7.1.7. Object serialization

.NET provides several mechanisms which might be useful in the context of sending objects throughout the network. Pass-by-value semantics or pass-by-reference semantics can be used.

Pass-by-value semantics .NET provides three ways of serializing objects, namely (1) using a binary format, (2) using an XML serialization, and (3) using SOAP.

Binary serialization: This form of serialization has the advantages of preserving type fidelity, meaning that the *entire* state of an object is serialized/deserialized (i.e., including non-public fields), and being efficient. Its implementation is comparable to serialization in Java.

XML serialization: The XML serialization, in contrast, neither preserves type fidelity (only public fields are considered) nor is efficient. However this mechanism is very useful for applications which are not to be restricted by the data types they use, because it enables the use of objects without being forced to have the type describing it. Indeed, as the XML serialization serializes the object in a human-readable format, you do not have to know at runtime the type of the serialized object you receive in order to take benefit of its public fields. On the other hand, that means that in order to take advantage of those fields, one must provide an own deserialization mechanism. In any case, to deserialize an object in the proper sense, i.e., recreating an object from a serialized representation, its assembly is required. However, with access to public fields without deserialization, objects can be efficiently (pre)filtered while in transit [EFGH02].

SOAP serialization: This third serialization mechanism combines the benefits of both previously presented mechanisms, by being essentially an XML serialization, yet providing the possibility of exploiting the .NET advanced serialization mechanisms, i.e, *customizable* serialization and deserialization. Moreover, the SOAP serialization allows to serialize an object and its fields without imposing any design guidelines (special constructor, etc). But in spite of its advantage, the SOAP serialization does not provide the deserialization of an object without having the assembly of it.

As discussed in Section 6, our prototype uses the XML and the SOAP serialization in order to transfer objects. The XML serialization is only used to serialize the type representation.

Pass-by-reference semantics .NET *remoting* is .NET's mechanism for invoking methods on objects across the network as if these were local to the invoker.

.NET remoting uses the well-known concept of proxies in order to mimic the object to call, and uses the serialization mechanisms introduced above for marshalling arguments. .NET provides several variants of remote invocations, among which the programmer chooses by constructing classes of remotely invocable objects accordingly. Furthermore, .NET supports multiple protocols for carrying out the communication underlying remote invocations (currently HTTP or TCP/IP) and provides several utilities in order to configure the access to a remotely invoked "server" object, which can be of the following:

Server-activated: As suggested by its name, this mode is characterized by having server objects' activation and deactivation guided by the server itself. Two variants exist:

Singleton: This kind of server object is automatically activated at startup of the server CLR. It's lifetime is hence equal to that of the server, and a single instance is used to handle all remote invocations.

Single call: New, automatically created, copies of the server object handle individual incoming invocation.

Client-activated: In this mode, activation and deactivation of server objects is triggered by clients. To avoid the problem of server objects being kept alive despite inactive clients, latter ones must *lease* server objects, which after expiration of all leases are garbage collected.

Our prototype uses the .NET remoting mechanism presented above enhanced with dynamic proxies. The "type" of activation is chosen depending on the needs of performance of the prototype.

7.1.8. Transfering data

In order to transfer datas through the network, .NET proposes a `Socket` class. This class is usable to create TCP, HTTP, UDP or even IP multicast channels. The choice of the channel type really depends on the Quality of Services wanted. All these channels send serialized messages. For the implementation of our prototype none of these mechanisms had to be changed. Simply the appropriate channel type had to be choosen.

7.1.9. Dynamic proxies in .NET

A *dynamic proxy* class is a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class can be encoded and dispatched to another object through a uniform interface. Thus, a dynamic proxy class can be instantiated to create a typed proxy object for a list of interfaces without requiring any code pre-generation, such as with compile-time tools.

This mechanism is available in the .NET platform [Hus01] under the `RealProxy` abstract class. Overloading this class lets the programmer define a new dynamic proxy class. In order to do so, a subclass of the `RealProxy` class must define a special constructor and implement a specific `Invoke()` method. The constructor to implement takes only one parameter which is the type to mimic. The `Invoke()` method takes and returns an `IMessage` object. This method is performed everytime a method is called on the dynamic proxy. The return type contains the expected return value (which is casted to the real type the invoker expects to receive, that implies the returned value must be an instance of the expected return type). With the help of the `Invoke()` method arguments, it is possible to retrieve the name and the arguments of the real method the callee invoked. Figure 12 shows an implementation of such a dynamic proxy class.

```
public class MyProxy : RealProxy
{
  public Object o;
  public MyProxy(Type t, Object o): base(t) { this.o = o; }
  public override IMessage Invoke(IMessage msg)
  {
    IMethodMessage m = (IMethodMessage)msg;
    MethodInfo method = o.GetType().GetMethod(m.MethodName);
    Object retValue = method.Invoke(o,m.Args);
    return new ReturnMessage(retValue,null,
           0,m.LogicalCallContext, (IMethodCallMessage)m);
  }
}
```

Figure 12. Example of a dynamic proxy

```
public class TestProxy
{
  public static void Main(String[] argv)
  {
    MyProxy mp = new MyProxy(typeof(IPerson),
          new Person(new Name("Bob","Morane"), 12));
    IPerson ip = (IPerson)mp.GetTransparentProxy();
    Console.WriteLine(ip.getName());
    ...
  }
}
```

Figure 13. Class using a dynamic proxy

To obtain a dynamic proxy, `TransparentProxy` is requested via the `getTransparentProxy()` method. This method returns an object which can be cast into the type provided to the constructor of the dynamic proxy class. Figure 13 shows a class that uses such a dynamic proxy. Please note that the `Person` type (Figure 9) is a subtype of the `IPerson` interface (as we can only use interfaces with dynamic proxies in .NET).

7.2. Adaptation to .NET

This section presents the modifications to the type representation and the implicit structural conformance rules (presented in Section 4) in order to make them compliant to .NET

7.2.1. .NET type representation

To represent a type of a given object, .NET uses introspection classes: `Assembly`, `Module`, `ConstructorInfo`, `FieldInfo`, `MethodInfo`, `ParameterInfo`, `PropertyInfo`, etc. All these classes give information about the type they reflect, as for example, the name of the types they reflect, the modifier(s) of the methods of the types they reflect, etc.

As it is not possible to use such a description remotely (see Section 5) and as the dynamic proxies of .NET only allows to use interfaces or types that are marked as *MarshalByRef*, one must take this into account for our type representation. Our approach only focus on dealing with interfaces as one wants to have a complete control over the marshalling of the objects (which is not accomplished with the *MarshallByRef* tag).

In .NET, interfaces only allows specific declarations. For instance there is no possibility to include fields into an interface. However, it is possible to include abstract "properties" which are special fields with getter and setter methods. It is also possible to include events (in order to deal with events) and indexers (the equivalent of properties but for arrays). No modifier is acceptable for methods and properties, events or indexers.

To circunvent these limitations, some modifications have to be done in the previously presented classes and interfaces (see Section 5). First, new `PropertyInfo`, `EventInfo` and `IndexerInfo` classes must be created. Those classes subclass the `TypeInfo` class (as one need the same kind of information). Some changes must also be done in the `ITypeDescription` interface, leading to a new interface described in Figure 14.

This new interface does not give information about the constructors (as there are no constructors in interfaces) neither about the superclass (because interfaces cannot subtype a superclass but only other interfaces). However it gives now information about the `Property`, `Event` and `Indexer` fields. Of course the implementation of the type `TypeDescription` now mimics these transformations but the general behavior of the class remains the same.

```
public interface ITypeDescription
{
  ClassInfo[] getClassInfo();
  MethodInfo[] getMethodsInfo();
  InterfaceInfo[] getInterfacesInfo();
  PropertyInfo[] getPropertiesInfo();
  EventInfo[] getEventsInfo();
  IndexersInfo[] getIndexersInfo();
  bool equals(ITypeDescription itd);
  bool conforms(ITypeDescription itd);
}
```

Figure 14. .NET compliant `ITypeDescription` interface

7.2.2. .NET implicit structural conformance rule

As mentioned previously, one must deal with the .NET specific limitations, i.e. dealing with abstract types rather than types. To that end, the rules described in Section 4 must be rewritten. Most of them remain the same (with some minor modifications), except the constructors rule that is removed and the supertypes rule that is changed. The .NET specific rules are presented in Figure 15.

Name: This aspect remains the same. Please see Figure 3 for its description.

Fields: This rule remains the same. The only modification is that now fields refer to either `Property`, `Event`, `Indexer` types as only those fields are acceptable in a .NET interface.

Method (i): This rule remains quite the same. But particular care must be taken to the arguments of methods as well as their return types. Indeed, if the arguments/return type of a method conforms "only" implicitly, those arguments then must be interface type, because in this case, dynamic proxies have to be used (and only interfaces are usable with dynamic proxies). $T_i\ isinterface$ denotes the fact that T_i is an interface.

It may be possible that the arguments (or the return type) are equivalent or explicitly conformant. In this case they do not need to be interfaces. To take into account this property without changing all the different rules, the tests of equivalence ($T == T'$) and explicit conformance ($T \leq_E T'$) have been added. Those tests are already present in the implicit structural conformance rule, but as in the method conformance rule (i) their are combined with the test of being an interface or not, they cannot cover the equivalent and explicit conformance cases, why the redundancy.

Supertypes (ii): This rule changes slightly as there is no need anymore to care about the superclass of a type.

$$
\begin{array}{l}
\textit{Method conformance (.NET version) } (i): \\
\quad T \leq_{Is}^{meth} T' \Rightarrow \forall m'(Perm(a_{1'} : T_{1'}, ..., a_{n'} : T_{n'})) : T_{r'} \in T' \\
\quad \exists m(Perm(a_1 : T_1, ..., a_n : T_n)) : T_r \in T \mid \\
\quad name(m) == name(m') \wedge \\
\quad \forall i \in [1, n](T_{i'} == T_i \vee T_{i'} \leq_E T_i \vee (T_{i'} \leq_{Is} T_i \wedge T_{i'}\ isinterface)) \wedge \\
\quad (T_r == T_{r'} \vee T_r \leq_E T_{r'} \vee (T_r \leq_{Is} T_{r'} \wedge T_r\ isinterface))
\end{array}
$$

$$
\begin{array}{l}
\textit{Supertypes conformance (.NET version) } (ii): \\
\quad T \leq_{Is}^{hier} T' \Rightarrow (T^{inter} \leq_{Is} T'^{inter}) \\
\\
\textit{Strong implicit structural conformance (.NET version)}: \\
\quad T \leq_{Is} T' \Leftrightarrow (T \leq_{Is}^{name} T' \wedge T \leq_{Is}^{hier} T' \wedge T \leq_{Is}^{field} T' \wedge \\
\quad T \leq_{Is}^{meth} T') \vee T == T' \vee T \leq_E T'
\end{array}
$$

Figure 15. .NET compliant rules

7.2.3. Dealing with equivalent or with explicitly conformant types

In the case of equivalency, no particular treatment is required. Indeed, the assembly of the type is already available at the receiver side implying that the object can be used immediately without having to get through all the protocol (Figure 1).

What if the received type is explicitly conformant to one of the types the peer is interested in (i.e. is a subtype)? First, how is it possible to check that a type is explicitly conformant with another one? There are two possibilities: (1) either download the assembly containing the type in order to use the .NET reflection mechanisms or (2) use recursively our type description of the object in order to try to find an intersection between the supertypes of both types.

Let us assume that the two types are explicitly conformant. In this case two possibilities are available. Either a `Binder` can be specified to our formatter in order to deserialize our message into the supertype directly, but that implies creating our own deserialization mechanism for the supertype (in order to drop the fields that are not interesting). The second possibility consists in downloading the assembly and deserializing the object.

It is currently not clear for us which solution is better for all cases. In one case, a deserialization mechanism must be provided and in the other case the assembly need to be downloaded only once. The final implementation will certainly be a tradeoff between bothering the user in writing special serialization mechanisms and between reducing the network load.

7.2.4. Dealing with types that conform "only" implicitly

If the type of an object received is neither the same nor a subtype of the type of interest, the implicit rules defined in the previous section are applied. A success means that the received object is implicitly conformant with the interested type (if the test fails, the object is simply dropped). During the rules check, the conversion between one method and another (same for the fields) is kept in memory (in a hashtable).

Once this check is done, it is possible to use the object as we discuss below. The next step is to deserialize the object in order to use it (invoke methods on it). For that purpose, the assembly of the type of the object must be downloaded. Finally the object must look like if it were of exactly the same type as the one the user requested. As this is not true (the type of the object only implicitly conforms to the type the user is interested in), the object must be wrapped and a reference to this wrapper must be given to the user. In order to create such wrappers, the dynamic proxy paradigm is used.

7.2.5. Using .NET Dynamic Proxies

In all the cases (methods, fields, supertypes), a dynamic proxy for the type of interest is created and the `Invoke()` method dispatches the calls on the dynamic proxy to the received object. However, this is not enough, one must take care of return types, methods arguments types, and fields types.

Implicitly structurally conformant methods: Here, two cases must be taken into account: (1) The return types are implicitly structural conformant, (2) the arguments are implicitly structural conformant. For these two cases, one must keep in mind that the programmer is interested in a specific type he designed. So he must expect to deal only with instances of this type.

Return types: The programmer expects to receive an instance of a specific type in return of his method call, but instead he receives an instance of a type that is "only" implicitly structurally conformant. In order to give him what he wants, a specific dynamic proxy must be created for the expected type. The `Invoke()` method of this new dynamic proxy is designed in order to dispatch the calls of the returned object to the implicitly structurally conformant one.

Argument types: The types of the arguments, provided upon invocation, of the received object do not match. In that case, a dynamic proxy is created for each argument of the received object. The `Invoke()` method in this case dispatches the calls to the arguments the programmer provided.

Recursion: In the above two cases, specific dynamic proxies in order to impersonate a specific type are defined. But what if the impersonating type has also methods that have arguments and/or return types that only implicitly structural conform to the impersonated type, i.e. if the implicit structural conformance has several levels? In that case, the rules must be applied recursively up to the point where the different types are equivalent or explicitly conformant.

Implicitly structurally conformant fields: Let us focus here on either properties, events or indexers (called here "fields"). All these fields can in turn declare methods. This implies, for example, that `a.myProperty` calls the `get()` method of the property (the same applies for indexers) and if the following `a.myProperty += aValue` call is done, the `set()` method is in turn called. For that reason, special care must be taken about the return type of the `get()` method and of the value given to the `set()` method, because there could exist "only" implicit conformance between the fields of the objects of interest and the ones of the conforming object. To achieve "complete" transparency, the same mechanisms described above for the methods are used.

Implicitly structurally conformant supertypes: This case does not imply any special care as if two types have super-abstract types that are implicitly structurally conformant it implies, according to the rules, that they have methods or fields which are implicitly structurally conformant. We fall back then into the above two cases, i.e. dealing with several dynamic proxies.

8. Performance

We present here some performance results of our prototype implementation. We measure the time taken by the different serialization mechanisms, the invocation time taken for calling a method using a dynamic proxy and compare it to a direct invocation, and finally the time taken to check conformance rules. All our results are based on simple types (precisions follow) and obtained with a HP Omnibook XT6050, Pentium 3, 256 MB Ram, HDD 30GB, Windows 2000 SP2, Visual Studio .NET Enterprise Architect 2002 version 7.0.9466.

8.1. Invocation time

We first consider the invocation time taken to invoke a method using a dynamic proxy and compare it to a direct invocation. The method called is `getName()` of the type `Person`. This type is described in Figure 9. The testbeds were the following: 100 repetition of 1000000 invocations to the method either directly or indirectly (using a dynamic proxy). We have made a repetition in order to see if, over the time, the overall slope was constant or not. The results are presented in Figure 16.

The average direct invocation time is about 0.000142 milliseconds. The average indirect invocation time is about 0.03 milliseconds. A huge difference can be seen in comparing these two results. Moreover, the overall time for making an indirect call depends upon the number of indirect calls performed on the dynamic proxy as well as its implementation. However, this amount of time still remains negligible with respect to the time taken for checking type conformance or for transfering objects, types descriptions and assemblies.

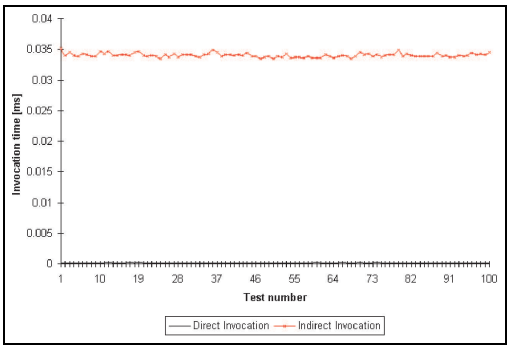

Figure 16. (In)direct Invocation time

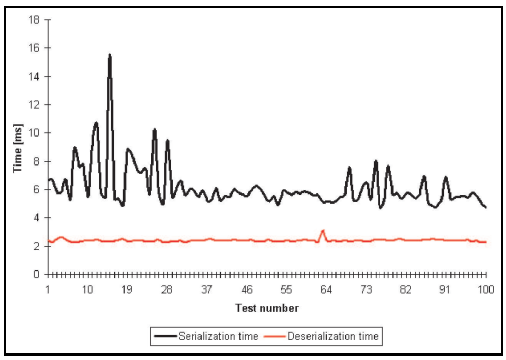

Figure 17. Type description (de)serialization

8.2. Creation, serialization and deserialization of type descriptions

We consider here the time taken to create a type description of a simple type (in this case the type `Person`, presented in Figure 9). Figure 17 presents our measurements obtained when creating and serializing as well as deserializing a type description of a type `Person`. Here, the type description of an instance of `Person` was (de)serialized 1000 times. We also average over 100 runs.

The average time for the creation and the serialization into an XML message of a `Person` description is about 6.14 milliseconds and the time taken to deserialize such a message is 2.34 milliseconds. Even if this cost is small, we must note that, again, these times depend much upon the serialized type. For a more complex type, the creation and the serialization of the message will take of course more time. However, we must also note that this serialization is done only once for a specific type. The time taken to send many objects of the same type will not be significantly affected by this (de)serialization time.

8.3. Serialization and deserialization of an object

We have measured the time taken to serialize and deserialize an instance of type `Person`. Results are shown in Figure 18. More precisely, we have measured the duration of serializing and deserializing this instance 1000 times. The average time to serialize the object is of 16.68 milliseconds and to deserialize of 1.32 milliseconds. This difference between the serialization time and the deserialization time could be explained by the fact that creating a SOAP structure from an object is more complex than creating an object from a SOAP structure. Again, we can expect a longer time if our object is a bit more complex.

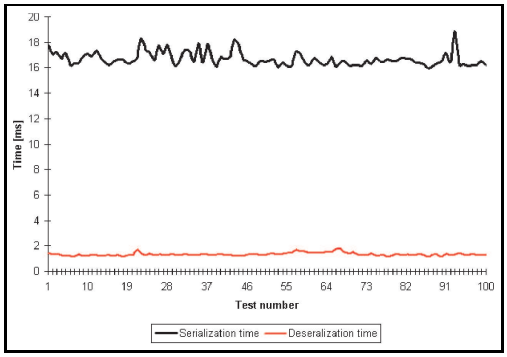

Figure 18. SOAP (de)serialization of an object

```
public class ClassA
{
    private string aString;
    private int anInt;
    private Person aPerson;
    public ClassA(Person aPerson, string aString, int anInt)
    {...}
    public string getString() {...}
    public void setString(string aString) {...}
    public int getInt() {...}
    public void setInt(int anInt) {...}
    public Person getPerson() {...}
    public void setPerson(Person aPerson) {...}
}
```

Figure 19. Class to conform to

8.4. Conformance testing

Finally, we also measured the cost of the verification of the conformance rules. These tests were done on very "simple" types. Those types are "simple" in the sense that they are composed principally of fields of primitive types and there is no recursion needed in order to test the implicit structural type conformance (the arguments of the methods, return types of the methods and properties are either equivalent or conform explicitly, see Figure 19 and Figure 20).

```
public class classa
{
    private string s;
    private int i;
    private Person p;
    private string h;
    public classa(string s, int i, Person p)
    {...}
    public string getstring() {...}
    public void setstring(string s) {...}
    public int getint() {...}
    public void setint(int i) {...}
    public Person getperson() {...}
    public void setperson(Person p) {...}
    public string gethello() {...}
    public void sethello(string h) {...}
}
```

Figure 20. Class that conforms implicitly to the one presented in Figure 19

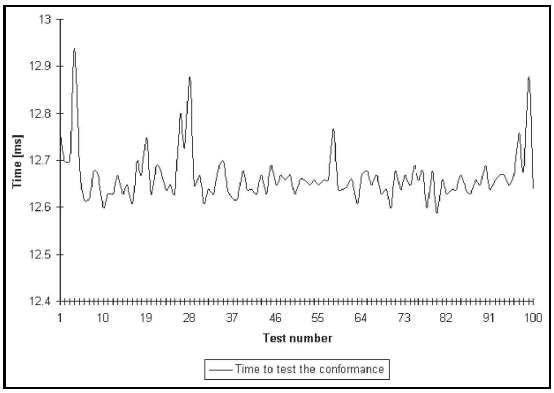

Figure 21. Time to test the conformance between two simple types

Though these measurements do not give information about the time it takes to test the conformance between two "complex" types, one can easily use them to deduce the time taken for a more "complex" conformance checking (by induction).

We have performed 100 times 1000 verifications. Figure 21 shows the different results. One can see that the average time to test the implicit structural type conformance is of 12.66 milliseconds. Even if this time does not reflect the overall time for all types, it gives, in some sense, a lower bound.

9. Concluding Remarks

This paper addresses the issue of type interoperability in a distributed environment. The goal is to make transparent for the programmer the use of one type for another, even if these types do not exactly have the same methods or names, as long as they aim at representing the same software module. We believe this form of interoperability to be crucial in modern distributed computing where several software modules need to be assembled. Our approach can also be used in a CORBA or Java RMI environment to extend their capabilities with type interoperability.

One possible application could be type-based publish/subscribe (TPS) [Eug02]. In the TPS scheme, event consumers subscribe their interest in objects of specific types and producers publish events of specific types. TPS ensures type safety and preserves event encapsulation with application-defined event types: in the original pattern, the subscriber knows in advance the type of events it receives (type-safety) and subscriptions operations of the type can be used for content-based filtering (encapsulation). The main issue with TPS is that the subscribers and the publishers must agree on the type they want to transfer/receive. Enhancing TPS with type interoperability would simply alleviate this problem. Another possible application could be borrow/lend (BL) [EB02]. Components can make objects, called resources, available to other components by indicating that they are willing to *lend* those objects. Conversely, components requiring resources can *borrow* such resources. Interaction between components hence takes place indirectly and anonymously, nevertheless explicitly, through first class resoure objects. A *borrower* can describe the resources it requires based on several *criteria*, which must be met by a lent resource to make interaction possible. A possible criteria is *type conformance*, for a type T_B with which the lent resource's type T_L must conform. Different levels

of conformance ranging from explicit (name) conformance over different "depths" of implicit (structural) conformance to a form of completely dynamic typing are possible.

In general, combining type interoperability with language interoperability makes the use of object-oriented middleware systems more attractive. One of the main issues in such systems is indeed that the different programmers must agree on a common type system or, at least, on a common way of describing types. This kind of assumption is far from being trivial in distributed dynamic systems where new types can be defined and exchanged on the fly, which changes the type hierarchy continuously.

Our approach is based on implicit structural type conformance rules and rely on an optimistic transport protocol as well as serialization mechanisms for marshalling the type description and the object itself. In our prototype, the XML serialization has been used to describe the type representation of the object and the SOAP/binary serialization has been used to serialize the object itself. Our approach focuses on the structure of the types instead of their behaviour. The implicit structural type conformance we have defined relaxes the strong assumptions of a type system. However, even if our rules have been written in a general way, we are aware that we cannot ensure complete conformance for all the possible cases.

We have demonstrated that the price for having type interoperability in a distributed system is not so high in comparison with the possibilities offered by such an enhanced system.

References

[BW98] M. Büchi and W. Weck. Compound Types for Java. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 362–373, October 1998.

[EB02] P.Th. Eugster and S. Baehni. Abstracting Remote Object Interaction in a Peer-2-Peer Environment. In *2002 Joint ACM Java Grande - ISCOPE Conference*, November 2002.

[EFGH02] P.Th. Eugster, P. Felber, R. Guerraoui, and S.B. Handurukande. Event Systems: How to Have Ones Cake and Eat It Too. In *Proceedings of the IEEE International Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 625–630, July 2002.

[Eug02] P.Th. Eugster. *Type-Based Publish/Subscribe.* PhD thesis, EPFL, 2002.

[Hus01] K. M. Hussain. *Microsoft .NET Programming Tutorial.* Dotnetrox web site, 2001.

[KS01] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 1–12, June 2001.

[LB98] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 36–44, October 1998.

[LBR96] K. Läufer, G. Baumgartner, and V.F. Russo. Safe Structural Conformance for Java. Technical Report CSD-TR-96-077, Department of Computer Sciences, Purdue University and West Lafayette, December 1996.

[Lev65] V. I. Levenshtein. *Binary codes capable of correcting deletions, insertions and reversals*, volume 163, chapter 4. Doklady Akademii Nauk SSSR, 1965.

[Mic02] Microsoft. *Common Language Infrastructure (CLI). Partition I: Concepts and Architecture (ECMA TC39/TG3)*, 2002.

[MR95] P. A. Muckelbauer and V. F. Russo. Lingua franca: An IDL for structural subtyping distributed object systems. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'95)*, pages 117–133, June 1995.

[OMG01] OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, February 2001.

[Sun99] Sun. *Java Core Reflection API and Specification*, 1999.

[TL01] Th. Thai and H. Lam. *.NET Framework Essentials.* O'Reilly and Associates, Inc., June 2001.