Pragmatic Type Interoperability[*]

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

Swiss Federal Institute of Technology in Lausanne
[a] *Distributed Programming Laboratory*
[b] *Programming Methods Laboratory*
{Sebastien.Baehni, Patrick.Eugster, Rachid.Guerraoui, Philippe.Altherr}@epfl.ch

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 dynamic and distributed environment. Our approach is based on an optimistic transport protocol, specific serialization mechanisms and a set of implicit type conformance rules. We experiment the approach over the .NET platform which we indirectly evaluate.

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 environment where objects of new types are introduced and exchanged in the system (either remotely or directly). Typically, different software modules might need to be assembled in a distributed application. Some of these modules represent a single logical entity.

Motivation. Type interoperability has been studied in centralized applications [5]. However, as we discuss in Section 2, the proposed solutions are too rigid for a dynamic distributed environment. In short, such solutions assume a priori knowledge of the type hierarchy. Our aim is to provide a transparent solution to this problem in a distributed environment. Basically, we are interested in devising a flexible scheme to allow objects of different types, that aim at representing the same module, to be remotely exchanged (not only passed-by-reference, but especially also passed-by-value) 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.

Contribution. This paper presents our approach to "distributed" type interoperability based on an optimistic transport protocol (that saves network ressources) as well as serialization mechanisms (that guarantee the efficiency of type comparison between objects). To experiment our approach in a concrete setting, we have implemented it over

[*]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.

a popular object-oriented platform: .NET[1]. This platform has been chosen because it provides the highest level of interoperability "underneath" type interoperability: language 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 a type conformant object received from a remote host, and we precisely measure this overhead through our prototype implementation. Indirectly, through our performance measures, we evaluate the .NET serialization (XML (*eXtended Markup Language*), SOAP (*Simple Object Access Protocol*) and binary) mechanisms together with its reflection capabilities.

Roadmap. Section 2 puts our work in the perspective of general approaches to language and type interoperability. Section 3 overviews the problem of type interoperability in a distributed environment and our approach to address 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 some performance measurements of our prototype. Finally, Section 8 draws some conclusions. More details on our .NET based prototype can be found in [1].

2 Related Work

2.1 Safe structural conformance for Java

Type interoperability was addressed for a centralized context in [5] through the notion of *structural conformance*. The structural conformance rules are based on the Java type hierarchy (a type is conformant with another type if it implements each method of the second type) 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, neither to be tagged as being structural conformant enabled.

[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.

2.2 Compound types for Java

The idea of providing compound types for Java [2] aims 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: *[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 the implemented methods of a type are taken into account instead of only its name. However these compound types are more about composition than about structural conformance i.e., making type interoperable.

2.3 Interoperability in Corba

CORBA [9] addresses the language interoperability problem through an *interface definition language* (IDL). 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. 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. Note that CORBA implementations provide various mechanisms, such as the *dynamic skeleton interface* and *dynamic invocation interface*, as well as 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 because of the lack of a general protocol for transferring efficiently objects as well as type interoperability rules.

2.4 Interoperability in 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 (bytecode) 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 [7, 10] is problematic outside of Java. Like CORBA, this dynamic linking mechanism could be used for implementing type interoperability, but again, to our knowledge, no efficient protocol an type interoperability rules have never been proposed.

2.5 Microsoft .NET

Just like CORBA, .NET aims at unifying several object-oriented languages through a *common type system* (CTS). The advantage here is that the programmer does not need to reimplement the type of interest in all 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 aim 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 [8] 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 is not bound to any intermediate language but rather to the type system of the platform itself. Moreover, our approach encompasses 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. Consider a type `Person` [1] with 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, 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 the translation rules can be hardcoded into the system.

However, when the system is distributed and "dynamic", i.e. where new events of new types can be put into the system through remote locations at runtime, this problem is not trivial. A set of general rules that can be compatible with every type must be created and implemented into the system. This implementation must be compatible with the pass-by-reference and the pass-by-value semantics in order to achieve full distribution 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. This is out of the scope of this paper. Our 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. This protocol is optimistic in the sense that the code of the object as well as its type representation are not always send with the object itself, but only when needed. The general protocol of Figure 1 can be decomposed into three distinct subprotocols, namely (1) object serialization, (2) type description creation and (3) type conformance checking.

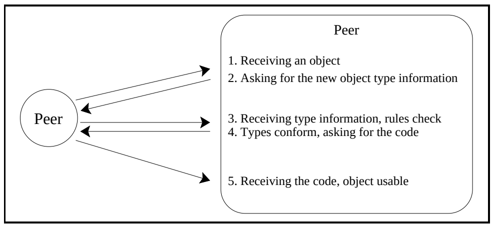
Peer
Peer
1. Receiving an object
2. Asking for the new object type information
3. Receiving type information, rules check
4. Types conform, asking for the code
5. Receiving the code, object usable

Figure 1. Ensuring 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 before giving our specific conformance rules.

4.1 Conformance categories

Hardware conformance 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.

Type conformance focuses on the interoperability between types. This category gathers two subsets: *implicit structural* type conformance and *implicit behavioral* type conformance. Implicit structural type conformance encompasses what we call *explicit* type conformance. Namely, explicit type conformance takes 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 rather tricky. Finally, the implicit structural conformance 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. For presentation simplicity, we say implicit structural conformance instead of implicit structural type conformance.

4.2 Type conformance rules

We first introduce here several basic notations and definitions that will help us explain the different aspects of conformance. Finally we present the implicit structural conformance rule.

[2]Structural conformance has been studied in [5] and is in between what we call explicit type conformance and implicit structural type conformance.

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 2 presents those terms, notations and the implicit structural conformance rules. First some notations that are used in the rules are defined. Then 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 2 presents the implicit structural conformance rules.

Decomposing implicit structural conformance. We define different aspects of conformance as follows:

Name (i): This aspect takes into account 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) [6] 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 (i.e. supertypes), 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} denote the superclass and the set of interfaces of type T respectively.

Methods (iv): Conformance between methods is a bit more tricky. First, the modifiers of the methods are supposed to be the same (this assumption is implicitly assumed in the rule). Then, to describe the corresponding rule, three parameters for each method are taken into account: the name of the method, the arguments of the method and the return type of the method. To understand this rule, one must think which uses the (1) return parameter

[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.

$Notations:$

$T\ denotes\ a\ type\ T$

$T_{GUID}\ denotes\ the\ globally\ unique\ identifier\ of\ type\ T$

$m\ denotes\ a\ method\ m$

$cons\ denotes\ a\ constructor$

$Conformance:$

$T \leq_E T' \Rightarrow T \leq T'$

$T \leq_I T' \Rightarrow T \leq T'$

$T \leq_{Is} T' \Rightarrow T \leq_I T'$

$Equality:$

$T == T'\ iif\ T_{GUID} == T'_{GUID}$

$Equivalence:$

$T \leq T' \wedge T' \leq T \Rightarrow T \equiv T'$ (case for $\leq_E$ iff T'==T)

$Superclass:$

$T^{super} \equiv (T' \mid T'\ superclassof\ T)$

$Interfaces:$

$T^{inter} \equiv (\{T' \mid T'\ interfaceof\ T\})$

$Name:$

$(name(x) \mid x \in \{T, m, cons\}) \equiv name\ of\ x$

$(as\ a\ case\ insensitive\ string\ representation)$

$Name\ conformance\ (i):$

$T \leq_{Is}^{name} T' \Rightarrow LD(name(T), name(T')) = 0$

$Field\ conformance\ (ii):$

$T \leq_{Is}^{field} T' \Rightarrow \forall f' : T_{f'} \in T'\ \exists f : T_f \in T \mid T_f \leq_{Is} T_{f'}$

$Supertypes\ conformance\ (iii):$

$T \leq_{Is}^{hier} T' \Rightarrow (T^{super} \leq_{Is} T'^{super} \wedge T^{inter} \leq_{Is} T'^{inter})$

$Method\ conformance\ (iv):$

$T \leq_{Is}^{meth} T' \Rightarrow \forall m'(Perm(a_{1'} : T_{1'}, ..., a_{n'} : T_{n'})) : T_{r'} \in T'$

$\exists m(Perm(a_1 : T_1, ..., a_n : T_n)) : T_r \in T \mid$

$name(m) == name(m') \wedge$

$\forall i \in [1, n](T_{i'} \leq_{Is} T_i) \wedge T_r \leq_{Is} T_{r'}$

$Constructor\ conformance\ (v):$

$T \leq_{Is}^{cons} T' \Rightarrow \forall cons(Perm(a_1 : T_1, ..., a_n : T_n)) \in T$

$\exists cons(Perm(a_{1'} : T_{1'}, ..., a_{n'} : T_{n'})) \in T' \mid$

$name(cons) == name(cons) \wedge \forall i \in [1, n](T_{i'} \leq_{Is} T_i)$

$Implicit\ structural\ conformance\ (vi):$

$T \leq_{Is} T' \Leftrightarrow (T \leq_{Is}^{name} T' \wedge T \leq_{Is}^{hier} T' \wedge T \leq_{Is}^{field} T' \wedge$

$T \leq_{Is}^{meth} T' \wedge T \leq_{Is}^{cons} T') \vee T == T' \vee T \leq_E T'$

Figure 2. Conformance rules

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 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.

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).

Implicit structural conformance (vi). We now describe the implicit structural conformance ($\leq_{Is}$). A type T implicitly structurally conforms to a type T' iff 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'.

One could think of having a weaker 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 might lead to receive 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 do 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 reflective capabilities of the object-oriented platform are used as a basics, as they provide some useful mechanisms that help 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 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. To create such instances of our introspection classes, the introspection classes of the chosen object-oriented platform are used.

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 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 or the 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.

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.

6.1 Overview

For the same reasons presented in Section 5, it is not desirable to send the type representation of the object with the object itself. Indeed, objects of the same type might have already been received before, and there might not be any 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 XML-SOAP approach

We describe here how 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). The XML serialization is used to provide a human readable type description of the sent object as well as download paths information to get the code of it. The SOAP or binary serializations are used to serialize efficiently the whole object (including the private fields). For connecting objects remotely, our approach assumes and uses the remoting mechanisms of the choosen object-oriented platform.

Pass-by-value semantics. As presented in Section 5, 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 mechanisms are used. That is, an 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 an XML message is received, it is deserialized in order to get the corresponding type information. A check is done 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. To deal with such conformant objects, dynamic proxies are used (see [1]). Figure 3 illustrates the serialization of an object of type `A` containing an object of a type `B`.

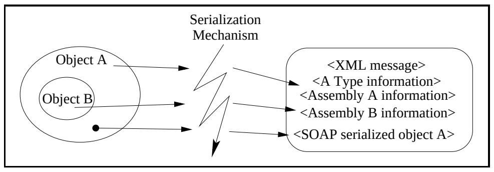

Figure 3. A hybrid serialization scheme

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* 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 mechanisms and dynamic proxies are used and for the the pass-by-reference approach, the basic remoting mechanisms enhanced with dynamic proxies (see [1] to have a look at our .NET implementation) are used.

7 Performance

We present here some performance results of our prototype implementation. All our results are based on simple types (see [1]) 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.

7.1 Invocation time

We first consider the invocation time taken to invoke a method using a dynamic proxy and compare it with a direct invocation. The method called is `getName()` of the type `Person`. This type is described in [1]. The testbeds were the following: 100 repetitions of 1000000 invocations to the method either directly or indirectly (using a dynamic proxy). We have made repetitions to see if, over the time, the overall slope was constant or not. 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, type descriptions and assemblies.

7.2 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`). 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, the time depends upon the serialized type. 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.

7.3 Serialization and deserialization of an object

We have measured the time taken to serialize and deserialize an instance of type `Person`. 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 it of 1.32 milliseconds. This difference could be explained by the fact that creating a SOAP structure from an object is more complex than the opposite.

7.4 Conformance testing

Finally, we also measured the cost of the verification of the conformance rules. These tests were done on very "simple" types [1]. We have performed 100 times 1000 verifications. 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.

8 Concluding Remarks

This paper addresses the issue of type interoperability in a distributed environment. We 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, possibly developed by different programmers, on different languages and systems, need to be unified. Our approach can also be used to extend CORBA or Java RMI with type interoperability capabilities.

One obvious application of type interoperability is type-based publish/subscribe (TPS) [4]. With TPS, subscribers express their interest in events of a given type and publishers produce events of specific types. The main issue with TPS is that the subscribers and the publishers must agree a priori on the types they want to transfer/receive. Enhancing TPS with type interoperability would simply alleviate this problem. Another possible application of this form of interoperability is the borrow/lend (BL) abstraction [3]. In this application *lenders* can lend *resources* to *borrowers* via specific criteria. A possible criterion is *type conformance*, for a type T_B with which the lent resource's type T_L must conform.

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 objects and the SOAP/binary serialization has been used to serialize objects themselves. 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.

Finally, we have pointed out 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

[1] S. Baehni, P. Th. Eugster, R. Guerraoui, and P. Altherr. Pragmatic Type Interoperability. Technical Report EPFL/IC/TR-200308, Swiss Federal Institute of Technology-LAUSANNE, February 2003.

[2] 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.

[3] 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.

[4] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

[5] 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.

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

[7] 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.

[8] 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.

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

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