The Any Framework
A Pragmatic Approach to Flexibility

Kai-Uwe Mätzel, Walter Bischofberger
UBILAB
Union Bank of Switzerland
{maetzel,bischofberger}@ubilab.ubs.ch

Abstract

During the development of Beyond-Sniff, a distributed multi-user development platform, we were confronted with various, apparently unrelated problems: data, control, and user interface integration of distributed components, system configuration, user specific preferences, etc. Undoubtedly, it is not trivial to find solutions for such issues, but C++ makes it even more challenging due to its static nature and insufficient meta-information. To overcome these shortcomings, we implemented a small and powerful framework called Any[1]. The Any framework augments C++ with a flexible, dynamic, garbage-collected data representation mechanism. It serves as a language-independent data integration vehicle and provides data management and declarative retrieval facilities.

1 Introduction

In 1991, we began to develop the C++ programming environment Sniff [2]. Motivation for this project was that framework-centered software development greatly increases the requirements for development environments [3],[5]. Furthermore, there were no programming environments that scaled and provided the extensive browsing support we needed.

After Sniff was successfully commercialized, we started to work on Beyond-Sniff [4], a platform and set of tools for co-operative software engineering. The goals of this project are to develop a conceptual framework for co-operative software engineering, and to build the development environment needed for its enactment.

A software engineering environment provides support for so many different activities that a monolithic design would make no sense. This is especially true for co-operative software engineering environments, where the environment also provides communication and coordination support. Modern co-operative software engineering environments consist of a number of integrated tools. In the ideal case their integration is so seamless that the user believes him- or herself to be working with a single tool.

1. The name clash of our Any framework with the CORBA any data type is coincidental and does not imply a similarity.

A Beyond-Sniff environment consists of a set of tools, which use several shared services in order to provide functionality, as depicted in Fig. 1. These tools are relatively lightweight because much of their functionality is already implemented in the shared services.

Beyond-Sniff has the following characteristics which are especially relevant to this paper:

- It is a distributed environment built with a focus on scalability.
- It provides data, control and user interface integration mechanisms [15].
- New tools and services can be integrated at runtime, thus making their functionality immediately available.
- Tools and services can be substituted. Whether tools or services are substitutable depends upon their capabilities - not upon their implementation details, e. g. the implementation language used.
- A user can tailor his or her environment without affecting other users.

Beyond-Sniff has been designed and implemented using object technology, especially framework technology. Starting points were the application framework ET++ [17], C++, and the current implementation of the Sniff programming environment running on various flavors of the UNIX operating system. We decided to start under these conditions because they allow us to profit from years of experience as well as the results of our former work. Further information about Beyond-Sniff can be found in [4].

1.1 What are the problems we faced?

During the development of Beyond-Sniff we were faced with the following problems:

Data integration and message exchange between heterogeneous, distributed components. All components of Beyond-Sniff (tools and services) have to be able to exchange data amongst each other. Whether the data is exchanged or physically shared depends on its size and

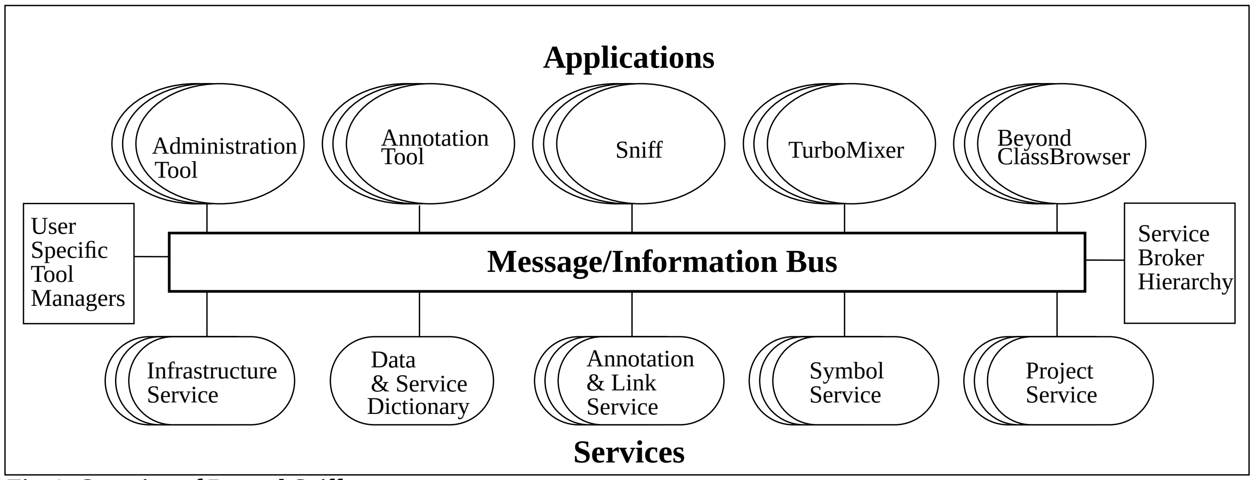


Fig. 1: Overview of Beyond-Sniff

the purpose of its use. In both cases, all components involved must agree on the meaning of the data. Its format cannot be predefined due to the openness of Beyond-Sniff. Furthermore, it should be possible to exchange data even if the receiver knows the sender's data model only partially and different languages are used.

Declarative data access. If large amounts of data have to be shared and therefore made accessible, the access should be declarative (e. g. through pattern matching or query languages) and scalable. This makes it impractical to use the data representation system of the host language.

User interface integration mechanism. Integrating a set of services and tools in a way that they provide the same experience as a stand-alone tool.

Reconfiguration and extension of open extensible environments. If new components can be integrated at runtime, they must be able to inform the rest of the system about their capabilities. Obviously, capabilities cannot be predefined. But to make the new functionality accessible for each user (for instance by means of a new menu entry), tool or service, all system entities must be ready to deal with them.

Schema evolution. The system has to be capable to handle various data model versions of a certain service or tool.

User specific tailoring. Tailoring of a user's environment requires a sophisticated preference system. Again, the format cannot be predefined because the adjustment features of future tools cannot be predicted. Preferences have to be persistent beyond session boundaries.

All these problems are not specific to Beyond-Sniff. They are common to a wide range of large systems. In order to tackle these problems, the following requirements have to fulfilled. They are independent from a specific system or application, especially independent from Beyond-Sniff.

R1 a language-independent, extensible, self-describing (i. e., semantic, according to Sims [16]) data representation mechanism, which can be smoothly integrated with arbitrary programming languages,

R2 fast, change-tolerant, alphanumeric, and binary IO mechanism,

R3 declarative or rule-based data access,

R4 persistent data storage, and

R5 robustness due to explicit data modeling and runtime type checking.

If a semantic data representation, a mechanism that fulfills R1, additionally meets R2, e. g. the listed I/O features, it can be used as a sophisticated message exchange format. This is crucial especially for distributed systems, since it eliminates the difference between external and internal data representation. A mechanism that meets all requirements R1 - R5 is called extended semantic data representation mechanism.

Obviously, there are systems that provide some of the listed features (namely certain dynamically typed languages and 4GLs). But there is no working system under our given external constraints[2] providing all of them. Therefore, we have had to invent a solution that meets the requirements as closely as possible and fits well into our development environment.

Note, that the external constrains do not reduce the applicability of our mechanism but rather makes it a

2. These are the implementation language C++ and that the mechanism has to be integrable with the existing code base of Sniff.

Anything usage

```
Anything employees, anEmployee, address;

anEmployee["LastName"]= "Weinand";
anEmployee["FirstName"]= "André";

address["City"]= "Mountain View";
address["Street"]= "High School Way";
address["State"]= "CA";
address["ZIP"]= 94041;

anEmployee["Address"]= address;

employees.Add(anEmployee);
employees.Add(anotherEmployee);
```

Anything import/export format

```
# array with 2 elements
        { # 1st array element; a dictionary
                /LastName "Weinand"
                /FirstName "André"
                /Address {
                        /City "Mountain View"
                        /Street "High School Way"
                        /State "CA"
                        /ZIP 94041
                }
        }
        { # 2nd array element
                /LastName ...   ...
        }
}
```

Fig. 2: Anything examples

generic solution which could be very useful in many other cases.

Before stepping into the Any Framework, we sketch some related work to show our sources of influence and inspiration. Then, we describe design and implementation of the Any Framework - our approach to an extended semantic data representation mechanism.

2 Related work

The presented systems and mechanisms are selected according to their contributions and ideas to meet the given requirements R1 - R5.

2.1 "Anythings"

André Weinand introduced a semantic data representation mechanism called Anythings. He was motivated by the following problems he had to solve with C++:

Dynamic Extensibility. In object-oriented systems, many objects of the same class exist playing slightly different roles in the context where they are used. These roles require the various objects to carry slightly different information. Usually, this results either in the definition of a large number of classes, which actually should belong to the same type, or in the definition of a comprehensive set of data members, which is used only partially by most instances.

The first approach is unacceptable because it results in an explosion of the number of classes. The second approach wastes a lot of memory. Both approaches make the system harder to understand and force the objects to carry only predictable information. But frequently, developers are faced with problems where this is too inflexible. Instances should therefore be dynamically extensible with arbitrary information.

An interesting application of extensible objects is information “piggybacking”. The different parts of a framework are coupled through a sophisticated information flow that usually involves dozens of mediators. Assuming that different parts want to extend the information they exchange, then the originally transmitted object only has to be extended with that information. This does not affect any of the mediators.

Attaching arbitrary information reveals the necessity for a semantic data representation mechanism (R1) because this information has to be interpretable for the receiver.

Streaming and configuration. Similar problems as with information piggybacking occur with object streaming. In order to rebuild an object from a stream the knowledge of how to do so is required. Due to information hiding, this is usually encapsulated in the object’s class. Therefore, the creator needs access to this code. To encode the object into a semantic data representation, would free the stream mediators from their dependence on this code. Note, that this actually decouples the receiver of such a stream from its source. If additionally the stream format is human-readable, it can be used for debugging purposes as well as a simple but flexible configuration or preferences mechanism.

Data structure mining. Legacy information systems sometimes export data in more-or-less structured formats. The problem is to parse the format and to dynamically construct a data type, whose instances represent the different results in a uniform way. The evolved data type could be for instance a class in which the set of data members correspond exactly with the set of occurred keys.

Anythings were designed to support solutions for the above problems. An Anything example is shown in Fig. 2. Their most important attributes:

- scalar data types such as integers, strings etc.,
- composed data types, e. g. arrays and dictionaries,
- automatic conversion between all scalar types, and from scalar types to arrays,
- meta-information about the structure of dictionaries,
- garbage-collection based on reference counting,
- human readable streaming format.

These data types are mainly derived from languages like AWK or Perl which incorporate highly flexible data structures like arbitrary nested associative arrays.

Anythings are highly flexible and well suited for a many tasks in small-scale development. They meet R1 and partially R2. However, their high convertibility violates R5 and makes them dangerous to use as an exchange mechanism in large distributed systems like Beyond-Sniff. There, the components have to negotiate and use a common data model. The data itself, not the access operation, should decide about the type of the accessed data.

Furthermore, to represent entire data models it is necessary to have additionally higher level data types, such as classes., A type system that provides classes and as much of the Anything's flexibility as possible, proved to be better than the pure Anythings. Such a system is just as applicable as Anythings - but safer, because it meets R5.

Anythings were not only the inspiration for the Any Framework, but were also our first code base.

2.2 Smalltalk Meta-Classes

Reflection is indispensable for a semantic data representation mechanism. It is a prerequisite for dynamic type checks, related operations, and therefore the best way to make the data representation self-describing and robust (R1, R5). Reflection can be found in many dynamically and statically typed languages or systems. Usually, the implementation of these features is quite similar to their realization in Smalltalk [10]. For each structural entity there is an object describing it.

2.3 NewtonScript

Beside Anythings, we consider NewtonScript another source of inspiration, especially in consideration to the requirements R3 and R4. NewtonScript is the prototype-based object-oriented programming language for the Apple Newton. The basic entities in NewtonScript are frames. A frame consists of various named slots which can hold either a scalar or composed data item or a functional block known as method. NewtonScript introduced a structuring concept called soups.

> "... soups are persistent storage objects that hold collections of related data items called entries, which are frames ..." [1].

Soups do not define an internal structure, they only serve as a grouping mechanism. They are object pools and may not be recursive.

Soups are very useful in their role as natural scopes for retrieving particular frames or objects, which is of great importance for R3. Some systems like ET++ already have a technically quite similar mechanism frequently called the “object table”. In contrast to NewtonScript's soups, the object tables are usually used as a hidden, internal book-keeping mechanism to implement reflection and not as a grouping concept.

2.4 ODMG Object Database Standard

The ODMG Object Database Standard defines an intuitive object query language. We found the declarative aspects (R3) of OQL [7] convincing. It is well designed for class based object systems and is based on the proven concepts of a query language.

2.5 CORBA and RPC

RPC [6] and CORBA [13] are widely accepted and used. These techniques prove a good fit for distributed systems solving exactly defined tasks. However, their design does not take adequately into consideration the facts that systems have to exchange large amounts of arbitrary structured data and that they evolve over time.

Evolution. With both techniques, the developer implementing the changes has to rebuild even those clients that do not use the new extensions or are not affected by the changes of their servers. This shows that this kind of static interface definition is not as evolution-friendly as we expect it to be for Beyond-Sniff, or generally, large distributed systems.

In contrast to RPC, CORBA reveals less resistance to changes, but only if either

- interfaces are regularly extended by means of inheritance or
- the IDL data type "any"[3] is used.

Necessary changes can be realized by interface inheritance only in few cases. More often, many small changes have to be made, for example few signatures

3. IDL-any is used for the data type any in IDL-scripts as well as for their related structures in the actual implementation language as defined by the CORBA standard.

have to be changed at just one position of the parameter list.

Using IDL-any, the signature changes could be encapsulated by shifting changes from the type level to the runtime structure of data items. Numerous approaches are possible: a) each parameter list consists just of one IDL-any, b) each parameter list includes an IDL-any at its last position, c) only parameter lists which are intended to be extensible include an IDL-any. Changes can then be performed entirely on the structure of the IDL-any, which does not affect the original signature.

Data exchange. A related problem to evolution is data exchange. The physical exchange of structured data is limited within CORBA to the IDL data types or, within RPC, to the data types provided by the RPC's data representation mechanisms like XDR. In many cases this does not satisfy the needs of the applications built on top of them. IDL-any can be used to encapsulate the actual exchanged data structure.

Problems with IDL-any: Evolution as well as data exchange can be tackled by means of IDL-any. But in both cases there are several risks. Followingly, we describe two problems of the application of IDL-any.

- In large systems the IDL-any can only encapsulate an extended semantic data representation mechanism because of its lack of abstraction and flexibility. Why?

  An IDL-any instance can represent an arbitrary data graph. The leafs are either IDL's basic types, constructed types, templated types, or arrays.

  The CORBA standard defines just the basic operations needed to deal with such data graphs. There are no general mechanisms for comparing, sub-graph matching, iteration and similar operations. Furthermore, there is no concept of structural equivalence of particular data graphs, which reveals the lack of means for classification. This implies that each IDL-any has to be individually treated and that the basic operations have to be implemented by everybody, who intends to use IDL-any intensively.

  Beside these lacks of abstraction IDL-any is completely limited to such data graphs. There is no abstraction for dynamic data structures like dictionaries. Although there are ways[4] to emulate dynamic data structures, the usage of these constructions would rely on various conventions and therefore be clumsy to use.

  IDL-any meets requirement R1 only partially because of its lack of extensibility, whereas it meets R2 almost completely (except the alphanumeric representation). I/O mechanisms are intrinsics of the concrete CORBA implementations. Furthermore, the intended usage of IDL-any is limited to CORBA applications which makes it hard to use in other applications.
- The CORBA-defined broker algorithm bases on statically defined interfaces. Signature changes have therefore strong impact on the actual behavior of this algorithm. This is no longer true as soon as IDL-any are used to encapsulate signature evolution.

  Under this point of view, the integration of an extended semantic data representation mechanism with IDL-any can only be a pragmatic solution. In the long term we consider it indispensable to turn IDL-any into a semantic data representation mechanism itself. It should meet at least R1, R2, and R5 and considered accordingly by the object broker algorithms.

4. An IDL-any could be used to represent a particular value of a dictionary. It cannot be used to represent the concept dictionary. This would be possible through a combination of CORBA objects, which would represent the concept and use IDL-any to transfer values between them.

2.6 NEWI

NEWI [11] is an integration environment for co-operative business objects (CBOs). It has to cope with some of the problems listed above: Flexible data and control integration between distributed, heterogenous components. It tackles the problems in a manner quite similar to Anythings. The NEWI equivalent to Anythings are so called Semantic Data Streams (SDS). NEWI prefers semantic data streams instead of interface definition languages such as CORBA-IDL, for reasons similar to the problems mentioned in section 2.5 [16].

3 The design of the Any Framework

In the following chapter we present Anys, our design of an extended semantic data representation mechanism, that has the features (R1-R5) we presented above. Some implementation details of the various language bindings follow. The usability of Anys will be demonstrated with two real world examples.

3.1 Introduction

The Any Framework comprises scalar and compound data types. Additionally, it contains a object-based data type called AnyFrames. Object-based means that AnyFrames do not support methods. The structure of an AnyFrame is defined by its class. Single inheritance between classes is supported. Frames are rather similar to objects, except that they do not have attached functionality. Frames can be grouped by AnySoups.

```
AnyContext *gContext;

void InitSymtab() {
	AnyFrameDesc value(gContext, "AnyNodeValue",
				"Description",		new AnySlotDesc("AnyString"),
				"PropertyList",		new AnySlotDesc("AnyDict"),
				"Deletable",		new AnySlotDesc("AnyBool"),
				0);
	AnyFrameDesc node(gContext, "AnyTree",
				"Name",		new AnySlotDesc("AnyString", eSingleValue | eMustHave),
				"Value",	new AnySlotDesc("AnyNodeValue"),
				"Children",	new AnySlotDesc("AnyTree", eMayNotRemove),
				0);
}

AnyFrame MakeTree() {
	AnyFrame rt(gContext, "AnyTree", "Name", new AnyString("Root"), 0);
	AnyFrame n1(gContext, "AnyTree", "Name", new AnyString("Node1"), 0);
	AnyFrame n2(gContext, "AnyTree", "Name", new AnyString("Node2"), 0);

	AnyDict pl;
	pl.Append("A", 1);
	pl.Append("B", 2);
	pl.Append("C", 3);

	AnyFrame v(gContext, "AnyNodeValue",
		"Description", new AnyString("Value of Node 1"),
		"PropertyList", &pl,
		"Deletable", new AnyBool(TRUE),
		0);

	rt.Append("Children",n1);
	rt.Append("Children",n2);
	n1.Append("Value", v);
	n2.Append("Value", v);

	return rt;
}

main() {
	gContext= new AnyContext();
	InitSymtab();
	AnyFrame tree= MakeTree();
	cerr << "The name of the root node is " << tree.At("Name").AsString() << "\n";
	PrettyAnyWriter().WriteAny(cout, tree.ContAt("Children"));
}
```

Fig. 3: Any code example

Soups provide declarative retrieval facilities as appropriate means to describe sets of frames. Each soup can manage an arbitrary number of indices. Indices are used to control and speed up frame retrieval. They are well known from database technology.

Beyond all the features listed in section 1.1, we gave Anys a few more to make their usage more convenient:

- Anys are very flexible and dynamic. At lifetime, the relations between them usually tend to be so complex that it would require tremendous efforts from a programmer to care about their memory management. Therefore, Anys are garbage-collected. This enhances their fit with languages like Smalltalk and makes the C++ programmer's life easier.
- Anys are closely integrated into the host language.

Smooth bi-directional conversation facilities between the host language's data world and the Any data world exist. Each Any implementation has to provide a comprehensive set of built-in transformations.
- Furthermore, it should be possible to consider data items of the host language's world as parts of the Any world without having to transform them in advance. For instance, it should be possible to plug a C++ object into an Any. This requirement conflicts with language independence. A pragmatic solution is that the plug, like the C++ object, is only visible inside the Any for the component that created it. It can be neither streamed out nor streamed in.

3.2 Remark to the C++ code examples

In order to illustrate the definition, the text includes concrete C++ code examples. To understand these examples some knowledge about the C++ implementation of Anys is indispensable. Anys are implemented according to the handle/body class idiom; more exactly, the reference counting idiom. In [8] Coplien explains: "Use of two (or potentially more) classes where an instance of one serves as a manager for instances of the other is called the handle/body class idiom. ... When the body class contains a reference count manipulated by the handle class, such use is called reference counting idiom." The Bridge Pattern [9] describes a more general form of the handle/body idiom. The code examples only show the handle class.

3.3 Basics

3.3.1 Scalar and composed types

Anys comprise the following scalar data types: AnyBool, AnyInt, AnyDouble, AnyString, AnyProxy, and FlatAnys:
- AnyStrings are symbols in Smalltalk terminology. If several AnyStrings have the same value then they refer to the same string. Indeed, strings exist only once.
- AnyProxies are used to bind any kind of object or data of the host language's world into a composed Any.
- FlatAnys are specialized AnyStrings. FlatAnys represent composed Anys that are streamed into a string. This makes it possible to partially delay the reconstruction of Anys when reading them from a stream. This is crucial, if large amount of data are transferred. Furthermore, comparing two FlatAnys checks the structural equality of the represented composed Anys.

The composed data types are AnyArray and AnyDict:
- An AnyArray is a flexibly growing array of Anys.
- An AnyDict, is an array of slots addressed with AnyStrings, containing zero, one, or more Anys. Inserting an Any at a slot which does not yet exist results in its creation.

3.3.2 Frames

AnyFrames are like specially typed AnyDicts. The types define classes over the structure of the AnyFrames. A class defines the names of all slots, their types, properties, and values, as well as their number and sequence.

In the Any system, classes are represented by AnyFrameDescs, an abbreviation for AnyFrame descriptor. An AnyFrameDesc is a sequence of AnySlotDescs. The explicit representation of classes and their structure makes it possible to extend the type set or certain types at runtime and to dynamically check the type of a frame.

Runtime errors occur when accessing an AnyFrame at a non-existent slot or inserting an Any at a slot for which no AnySlotDesc exists in the frame's AnyFrameDesc.

AnyFrameDescs are shared between AnyFrames within a soup or context. Soups and contexts are described in section 3.4.1. Fig. 3 shows a simple and instructive Any example.

3.3.3 Semantic streaming

Anys have only a very limited built-in streaming facility. The produced stream does not ensure that a structurally identical Any can be reconstructed. If the case demands it, the semantic streaming facilities provided by AnyWriters and AnyReaders must be exploited. An AnyWriter is a visitor [9] that travels over the structure of a given Any and writes a self-describing output format onto a given stream. Consequently, an adequate AnyReader can reconstruct an identical Any from that stream.

The following reader/writer pairs must be provided by an implementation:

- schema-tolerant, ASCII-based reader/writer,
- schema-intolerant, ASCII-based reader/writer,
- binary-reader/writer.

A schema-tolerant, ASCII-based reader/writer pair is the ideal instrument for semantic streaming. The writer also streams the complete meta-information, e. g. all necessary type information and AnyFrameDescs. The reader can then reconstruct even AnyFrames whose classes were previously unknown to it.

Schema-intolerant reader/writer pairs include the basic type information but not the AnyFrameDesc. If a reader is forced to rebuild an AnyFrame without knowing its AnyFrameDesc, this results in a runtime error as opposed to the alternative of constructing an AnyDict. Fig. 4 shows the semantic streaming output of the schema-tolerant, ASCII-based PrettyAnyWriter of Fig. 3.

```
//Any:PrettyAnyIO
[2
   <
     "AnyTree"
     "Name"
       [1  "Node1" ]
     "Value"
       [1
          <
            "AnyNodeValue"
            "Description"
              [1  "Value of Node 1" ]
            "PropertyList"
              [1
                 {
                   "A"
                     [1  1 ]
                   "B"
                     [1  2 ]
                   "C"
                     [1  3 ]
                 } &10
              ]
            "Deletable"
              [1  T ]
          > &5
       ]
     "Children"
       [0 ]
  > &0
   <
     "AnyTree"
     "Name"
       [1  "Node2" ]
     "Value"
       [1 *5
       ]
     "Children"
       [0 ]
  > &20
]
```

Fig. 4: Semantic Streaming

3.4 Advanced features

Additional features are necessary to sensibly use Anys, especially AnyFrames. These features are soups and a declarative data access within soups.

3.4.1 Soups

Soups have been introduced mainly for two reasons:

- An easy to use utility to structure AnyFrames into disjunct, retrievable, changeable sets is considered indispensable. AnyArrays could serve a similar purpose with the drawback that the programmer has to insert each AnyFrame manually. An AnyFrame should be inserted automatically into a given soup at the point of its creation and removed at destruction time.
- Soups in the Any system play the same role as name spaces in C++. All AnyFrames of a soup share their AnyFrameDescs, which are only valid in their soup. This mechanism enables the use of efficient, schema-intolerant AnyReader/Writer pairs. Assume that all AnyFrameDescs between two soups are identical. It is then sufficient to transmit the raw data of an AnyFrame without additional meta-information.

The two purposes of soups are so fundamental that we defined two abstractions. An AnyContext maintains a table of AnyFrameDescs. Therefore, an AnyFrame can only be created within an AnyContext to be able to locate its AnyFrameDesc. An AnySoup, which is derived from AnyContext, additionally manages a collection of AnyFrames and provides query based retrieval and locking support on it.

3.4.2 Querying and indexing

AnySoups support declarative access to their AnyFrames. In contrast to navigating access, declarative access allows a set of AnyFrames to be described. It avoids the need to iterate over all of them and to check if they match a certain pattern. AnySoups incorporate an OQL query processor [7] which can evaluate any OQL query and delivers an AnyArray with all the matching AnyFrames. We chose OQL because it is an object-oriented standardized query language that matches well with our AnyFrame concept.

Query evaluation can only be performed reasonably fast on large AnySoups if they support comprehensive indexing. Indices are managed by AnyIndexManagers. Each soup can have an arbitrary configurable set of index managers.

An implementation has to provide at least the following index managers:

- the OnlyOneIndexManager, which actually provides no index, or with other words all frames are in the same index,
- the StandardIndexManager, which manages a separate index for each AnyFrame type.

Fig. 5 demonstrates two OQL queries. Note, that slots in AnyDicts as well as in AnyFrames are multi-valued. To access all values at once, the All method has to be used. By convention, ALL_<name> is always the name of an index.

```
# Returns an array of AnyDicts containing class name/metho
# name-pairs for all classes except class Any and all
# “Impl”-classes where the method names are not Get or Set.

select  struct(ClassName: x.Name,MethodName: y.Name)
from
        x in ALL_Class,
        y in x.All("MethodDef")
where
        x.Name != "Any"
        and not x.Name.Match("Impl")
        and y.Name.Match("^[GS]et")

# Returns type and name of all classes, method
# implementations, and includes

define symbols as
(       select  struct( type: "Class", Name :sym.Name)
        from    sym in file.All("Class")
) + (   select  struct( type: "MethImpl", Name :sym.Name)
        from    sym in file.All("MethodImpl")
) + (   select  struct( type: "Include", Name :incl.Name)
        from    incl in file.All("Include"),
);
select  symbols
from    file in ALL_SymtabFile
```

Fig. 5: An OQL-query for Anys

3.5 Discussion of language binding problems

Implementations of Anys exist in C++, Smalltalk and Python. This allows us to exchange Anys between programs written in these languages.

In C++ as well as in Smalltalk, we developed frameworks to implement Anys. Different tasks require different degrees of exploitation of a used mechanism. To make the use of a solution always profitable, the interface has to provide various layers of abstraction. Frameworks are an appropriate tool to realize this.

Obviously, the design of, and the effort needed to implement, these frameworks depend upon the underlying platform; just have a look at garbage collection! In C++ the reference counting idiom had to be implemented. The handle and body classes and all assignment operators had to be implemented. In Smalltalk, garbage collection is for free.

3.5.1 Smalltalk

We shall shortly present the two main alternatives to integrate Anys with Smalltalk. We had the choice of implementing Anys according to their definition, or to integrate them seamlessly with the language. The excellent dynamic features of Smalltalk would make this possible.

All scalar data types as well as AnyArray and AnyDicts could be mapped to Smalltalk intrinsics. For each AnyFrame type, the system could dynamically generate a structurally equvivalent Smalltalk class. In order to be able to distinguish these classes from ordinary Smalltalk classes, which is necessary for streaming, all these classes had to be derived from a single special root class. This seems to be a promising approach, but raises various problems.

If class definitions are changed at runtime, which has to be possible with Anys, then maybe large parts of the application have to be recompiled.

The programmer is no longer conscious that he or she is working with data that is possibly shared between various components in a distributed system. While designing and programming with Anys, we found it very helpful to be constantly reminded of the different purposes of Anys and ordinary data. The programmer is much more aware of the fact that the components exchange messages between each other.

4 Application of the Any Framework

Next, we will demonstrate the usage of Anys by the example of the implementation of Beyond-Sniff’s service architecture. First, we describe how Anys ease the implementation of an infrastructure for distributed services. Second, we show how we used Anys and Python [14] to make services more flexible and adaptable at runtime.

4.1 Beyond-Sniff’s service framework

Anys by themselves are only an extended data representation mechanism. In order to make them sensibly applicable in the intended context of distributed systems, they must be embedded in a proper infrastructure.

As depicted in Fig. 1, Beyond-Sniff follows a service architecture. All services, independent of their actual task, share common characteristics. They provide synchronous and asynchronous point-to-point or multicast communication and request handling facilities. Consequently, messages are encoded as AnyFrames, which include the transmitted information as well as the usual information like sender, receiver, sequence number, and so on. This information of course is also an

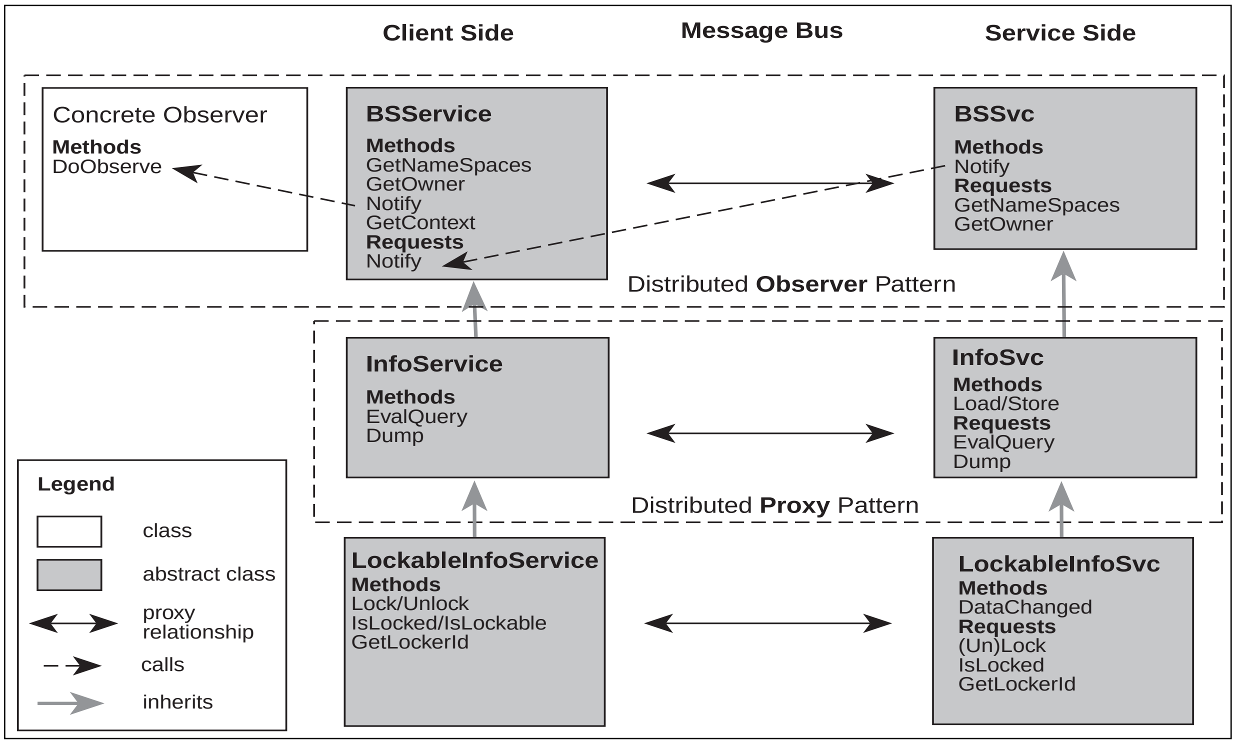


Fig. 6: The Beyond-Sniff client-service framework

Any. NEWI [11] uses a rather similar mechanism to make co-operative business objects (CBOs) communicate language-independently.

Clients and services have to communicate efficiently. Therefore, they negotiate the actual communication context in respect to their capabilities and technical environment. The context mainly consists of a common data model and reader/writer pair that appears to be most efficient.

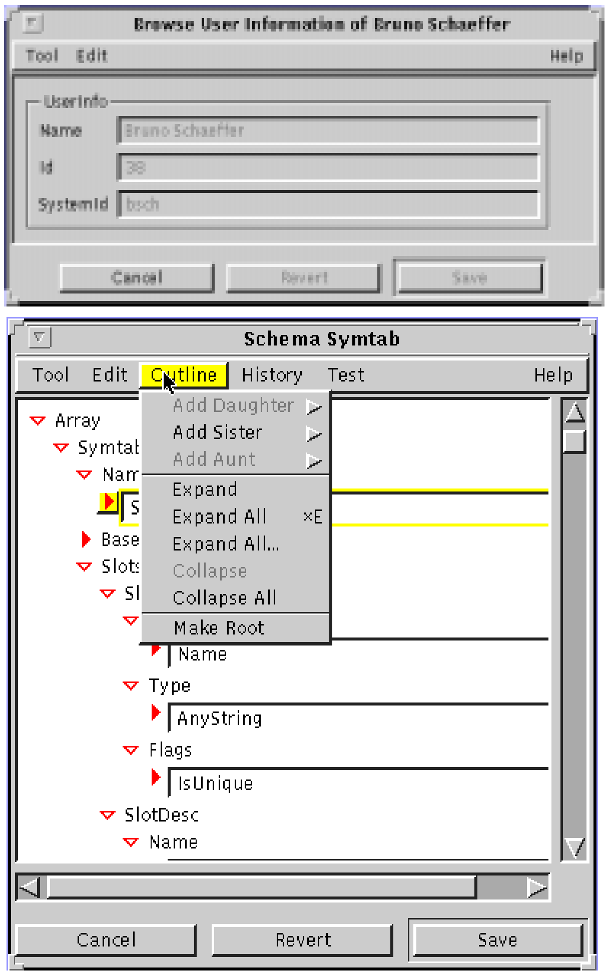
The Any message stream format is determined by the kind of underlying computer architecture of the involved parties. They use binary streaming if both are sitting on top of the same architecture. Beside the negotiated reader/writer pair, more descriptive pairs can always be used. If a message in a more tolerant format arrives, the receiver dynamically selects an appropriate reader.

Data model negotiation is rather simple. The client downloads the server's AnyContext, e. g. its set of AnyFrameDescs, and uses it to create service requests. The service is committed to serve such requests. Furthermore, a client can send any arbitrary request which might not be provided by the server. In this "trial and error" case, the client has to use schema-tolerant streaming.

Service functionality can be invoked dynamically by sending a request directly to the service or by using a service proxy. From the client's point of view, the proxy provides the service although it is just a front end to the real service. This mechanism is provided by a service framework in C++, depicted in Fig. 6. Additionally, it provides indispensable functionality for a distributed infrastructure such as service management, including finding and launching of appropriate providers, monitoring, load balancing, and more.

So far, the service framework solely exploits the basics of Anys. AnySoups with their querying facilities make large-scale data sharing and integration feasible without major effort. Therefore, we introduced information services. An information service manages an AnySoup and lets its clients manipulate and ask queries about the soup's content. In the service framework we provide different abstractions to the concept of an information service, as depicted in Fig. 6.

Visualizing and interactive editing of Anys is almost trivial due to their reflective nature. We exploit this to build generic message-monitoring tools and information service inspectors. Each message can be graphically displayed and manipulated, if necessary. The same is true for the soup managed by an information service. Fig. 7 shows the Generic AnyOutliner and the Generic AnyFrameEditor.


Browse User Information of Bruno Schaeffer

UserInfo

Name	Bruno Schaeffer
Id	38
SystemId	bsch

Fig. 7: AnyOutliner and AnyFrameEditor

The AnyFrameEditor can also be used to create any kind of form-based input dialog cheaply. The data structure to be filled up by using the form, has to be modeled only as a particular AnyFrame. The frame's content can be manipulated with the AnyFrame Editor, which dynamically creates the needed form.

4.2 Introducing scriptability into services

Services, especially in a software development environment like Beyond-Sniff, should be highly flexible and adaptable at runtime. Consequently, services have to have a reflection and a behavior manipulation component. The latter is responsible for providing the demanded flexibility. This can be achieved most comfortably by incorporating a scripting language with the services. We chose Python for this purpose.

The integration of Python with our C++ service framework was fairly straightforward. We extended the service interface with two new requests: EvalScript and InstallRequestHandler. EvalScript is used to evaluate a given script in the receiver's context. With InstallRequestHandler the authorized client can overwrite a particular request handler of a service with a Python script. For information about the integration of Python and C++ consult [14].

As already mentioned, Anys work as a data integration mechanism between different languages. The integration of Anys into Python enables the scripts to work with the data of the C++ service, such as the AnySoup of an information service. Fig. 8 shows this.

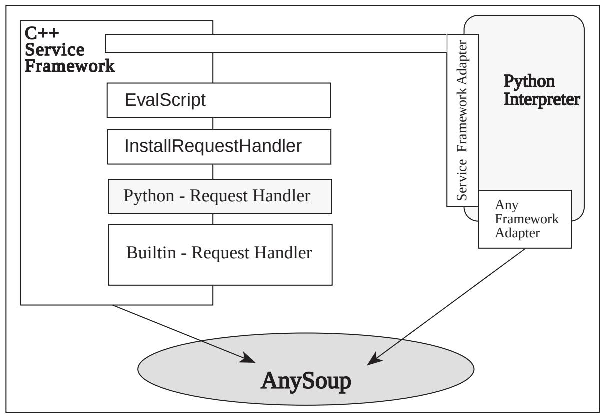


Fig. 8: A scriptable Beyond-Sniff service

5 Experience and conclusions

Anys are a language-independent semantic data representation mechanism which provides semantic streaming, soups, and OQL-based data access. They help the programmer to combine the strength of statically-typed languages like C++ with the convenience of dynamic, extensible, and object-based data types. Anys are widely applicable. Undoubtedly, they are most profitable in the context of distributed systems and language integration. We presented this by the example of the Beyond-Sniff service framework, written in C++, as well as the integration of Python into those services. Both examples represent heavily used mechanisms which are crucial to the Beyond-Sniff project. We can state that Anys proved their usefulness and applicability in large-scale development.

Anys do not provide facilities to attach functionality to certain slots as NewtonScript or Self do [18]. In the context of language-independent data integration, this would allow the introduction of data encapsulation and information hiding. Currently, either each client has to know how the provided data must be used to compute certain functions, or their results have to be part of the data. Function-slots would avoid such redundancies. Although we consider this feature useful, we decided not to implement it, since the integration of a scripting language has sufficiently met our requirements.

Further work on Anys will have to be done in the field of internationalization. Probably, we will provide two different AnyString implementations; one working with unicode characters, the other supporting character code pages.

6 Acknowledgment

We would like to thank André Weinand for his ideas, work, and discussions about Anythings and Anys, as well as for his valuable contributions to this paper. Furthermore, we thank Bruno Schäffer, Michael Scharf and Alexander Schwab for their helpful input. Bruno implemented AnyOutLiner and AnyEditor. Michael and Alexander worked on the integration of Python with the Any and the Beyond-Sniff service framework.

7 References

[1] Apple Computer Inc.: The NewtonScript Programming Languages: PIE Technical Publications, 1993

[2] Bischofberger,W.R.: Sniff - A Pragmatic Approach to a C++ Programming Environment. in Proc. USENIX C++ Conference, Portland, Oregon, Aug. 1992.

[3] Bischofberger,W.R., Kofler,T., Schaeffer,B.: Object-Oriented Programming Environments: Requirements and Approaches. In Software - Concepts and Tools, Vol. 15 No. 2, Springer Verlag, 1994

[4] Bischofberger,W.R., Kofler,T., Maetzel, K.-U., Schaeffer,B.: Computer Supported Cooperative Software Engineering with Beyond-Sniff. In Proc. Software Engineering Environments 1995, IEEE Computer Society Press, Los Alamitos, California, April 1995.

[5] Bischofberger,W.R., Kleinferchner,C.F., Maetzel, K.-U.: Evolving a Programming Environment Into a Cooperative Software Engineering Environment. In Proceedings of CONSEG'95, Tata McGraw-Hill, New Dehli, February 95

[6] Bloomer J: Power Programming with RPC; O'Reilly & Associates, Sebastopol, 1991

[7] Cattell, R.G.G. (Ed.): The Object Database Standard: ODMG - 93. Morgan Kaufman Publishers, San Mateo, California, 1994.

[8] Coplien J: Advanced C++ Programming Styles and Idioms, Addison-Wesley Publishing Company, Reading Massachusetts, 1992

[9] Gamma E, et. al.:Design Pattern - Elements of Reusable Object Oriented Software. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994

[10] Goldberg A., Robson D.: Smalltalk-80 The Language and its Implementation. Addison-Wesley Publishing Company, Reading Massachusetts, 1983.

[11] Integrated Object Systems Limited: Introducing Newi: Integrated Object Manual, Berkshire, 1994

[12] McKeehan J, Rhodes N: Programing for the Newton - Software Development with NewtonScript. AP Professional, Boston, 1994

[13] Object Management Group: The Common Object Request Broker: Architecture and Specification, 1992

[14] Rossum v. G.: Extending and Embedding the Python Interpreter, Stichting Mathematisch Centrum, Amsterdam 1995

[15] Schefstöm D, van den Broek G: Tool Integration–Environments and Frameworks. John Wiley & Sons, 1993

[16] Sims O.: Business Objects - Delivering cooperative objects for client-server; McGraw-Hill Book Company, London, 1994

[17] Weinand, A., Gamma, E.: ET++ - a Portable, Homogenous Class Library and Application Framework. Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, Sept. 1994. Universitaetsverlag Konstanz, Konstanz, 1994, pp. 66-92

[18] Ungar D., Smith R. B.: The power of simplicity; Proceedings of the OOPSLA '87 Conference, Orlando, 1987