Realization of a TMN Java Management API

Heiko Dassow
Dassow@tzd.telekom.de
Deutsche Telekom AG
Technologiezentrum Darmstadt
Germany 64307 Darmstadt

Christian Hubert • Hubert@ese-metz.fr • Student at Supélec Paris / Technische Universität Darmstadt
Birgit Frohnhoff • Frohnhoff@tzd.telekom.de • Deutsche Telekom AG
Gerd Aschemann • Aschemann@Informatik.TU-Darmstadt.de • Technische Universität Darmstadt

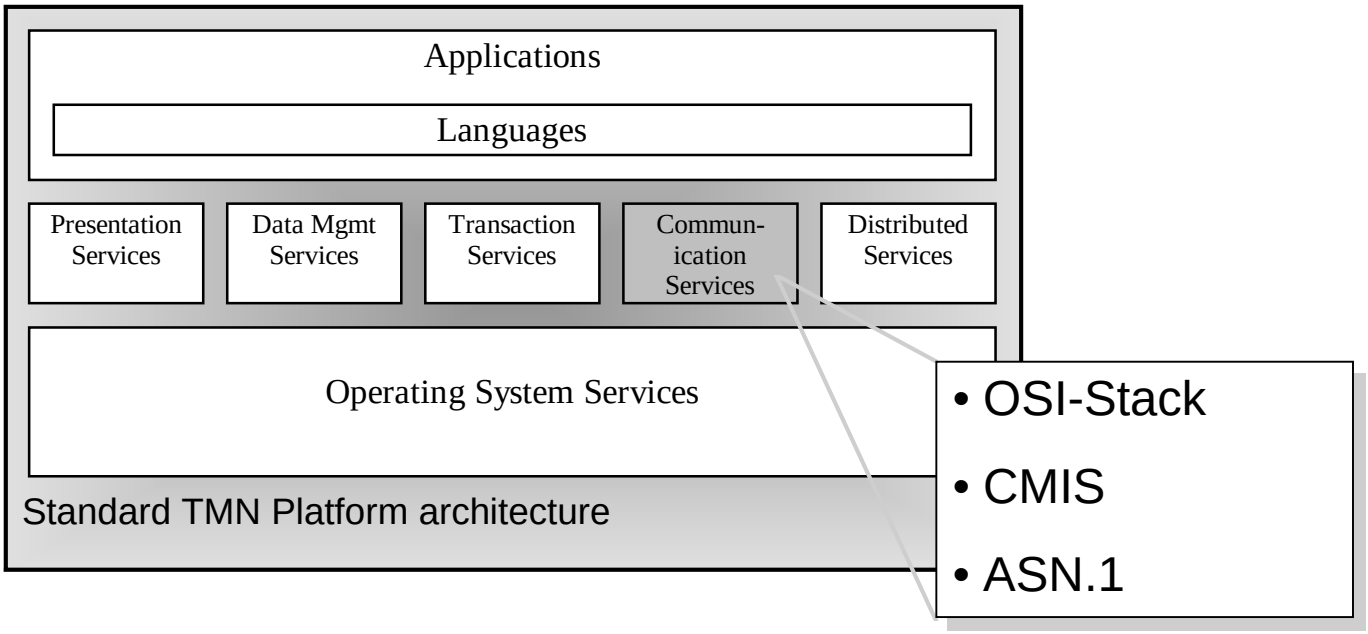
Several middleware products are dealing with the representation of ASN.1 values. This paper proposes a novel approach which realizes the object oriented representation of ASN.1 values completely in Java. A prototype implementation of this API has been realized in parallel to the definition process in order to evaluate the practicability of the paper based specification. The main difference of this implementation to existing C++ wrapper classes is, that the Java-ASN.1 API does not encapsulate existing native C code and can therefore be applied on several processing environments. This paper includes also a presentation on a completely new method of building ASN.1 value representations without referencing the corresponding type definitions (metadata) during runtime. This approach which implies the separation of the ASN.1 API into two functional components enables a totally new kind of architecture for TMN management platforms. This can be used to design a small independent client part and a powerful server part for BER value encoding.

Introduction

About 1.000 programming languages are currently available. Most of them are used for special purposes. The newly specified Java language, however, seems to be applicable to several applications and platform types due to the fact that the runtime is specified vendor independently. A further benefit of the Java language is the fact, that it can be easily integrated into existing Internet technology. This will enable new types of software architecture, like the mobile agent approach. Regardless of all these features and advantages of the Java language, however, another reason for the great success of Java is the similarity to the well-known C++ language and the exclusion of concepts like pointers and manually memory management. There can be no doubt, that the usage of pointers and manual assignment of memory implies the possibility of runtime errors.

We will describe the advantages of a TMN management platform and the necessity of machine independent programming languages for those platforms. Therefore we will propose to use Java in a specific area of TMN application. In order to consider already existing platform dependent implementations, an overview of several migration techniques for those implementations to Java will be given. The Java-ASN.1 API will be introduced by a comparison of the already existing APIs with this new specification. Afterwards, a specification of a Java based ASN.1 programming interface will be given. In order to illustrate the experience and the advantages of this Java API, a prototype application implemented at Deutsche Telekom will be described in this paper. Finally, an architecture for a customer network management scenario will be introduced, which is based on a prototype implementation of the Java-ASN.1 API.

General Requirements for a TMN-Platform



In order to reduce the complexity of TMN-applications as much functionality as possible should be encapsulated by a management platform. Common functionality should not be implemented individually for each application. The corresponding code should be shared among several applications. The basis of a management platform is a standard operating system like UNIX. Several middleware components, e.g. presentation services, transaction service, etc. are running on this platform and will be shared by the applications.

Due to the benefits of Java which are already mentioned it might be feasible to use this language also in a Telecommunication Management Network (TMN) environment. In order to assess this proposal, it is required to analyze the demand for platform independent management applications and their specific requirements in more detail.

It is not necessary to discuss the advantages of a programming language which is easy to use for the implementation of a management application. Another fact which can not be rejected is the advantages of a vendor independent hardware infrastructure for these applications. In particular, for small applications Java seems to be an optimal solution. The management of a large public network, however, requires a considerable amount of processing power and optimized software algorithms. Especially the fault management requires extremely short reaction times and it is obvious that, for example, a reconfiguration task should not wait until memory has been rearranged by a Java garbage collection. That is why we do not recommend to completely replace reliable high performance management systems by Java based implementation. On the other hand, it is feasible to perform non time critical operations on a separate machine. It is not sensible to restrict the usage of this sort of software to only one kind of machine. In particular, a customer network management solution requires that a management application will run on the different customers machines. Thus, the Java language seems to be well suited for the realization of such an application.

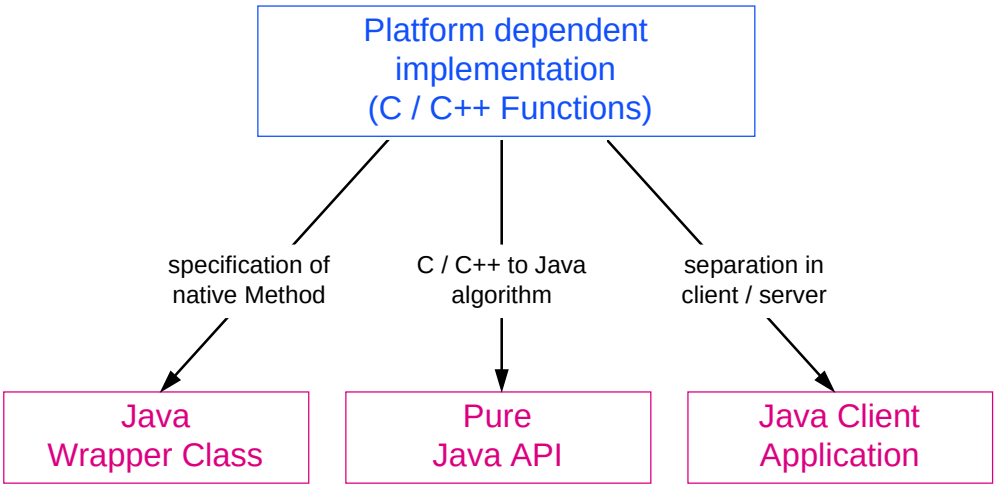
Obviously, the most specific and important component for an TMN platform is the Communication Service. This component deals with Communication Services via Q3- or X-interfaces that conform to Open System Interconnection (OSI). For this purpose, values described in Abstract Syntax Notation One (ASN.1)[9] [1] have to be exchanged between management applications based on OSI conformance. In order to include Java components in a TMN platform, OSI based Communication Services have to be provided into Java. For this purpose, further enhancements of the Java libraries in addition to the standard Java runtime environment are required on account of the Common use of ASN.1 in this area.

---

[9] "Specification of Abstract Syntax Notation one (ASN.1)", ITU-T Recommendation X.208, Melbourne, 1988

[1] "Abstract Syntax Notation one (ASN.1): Specification of basic Notation", ITU-T Rec. X.680, Melbourne, 1994 | ISO/IEC 8824-1 : 1995

Concepts for the Implementation of a Java API



The required ASN.1 Java API should be flexible enough to adopt several mapping approaches from existing management platforms. Therefore we have to consider several migration approaches from existing C or C++ APIs to a Java based API before we take a detailed look at the Java-ASN.1 API. For the migration of an existing implementation into Java we have identified three different approaches, which already have been used successfully for several migrations in the past. These approaches will be illustrated in more detail in the following paragraphs.

The Java language provides a technique which easily allows the definition of Java wrapper classes for already existing implementations. This technique has been successfully applied for the integration of vendor specific database access into Java before. The underlying concept is called native method call in Java. A native method is a Java method (either an instance method or a class method) whose implementation is written in a different programming language such as C. Therefore, an existing API can be migrated into Java without re-implementation of existing methods. When a native method call has to be performed during runtime, a shared library which contains the native implementation code will be referenced by the Java virtual machine. It should be mentioned, however, that in this case the application is not platform independent any more. In addition, the ability to load dynamic libraries is subject to approval by the security manager. Therefore, applets may not be able to use native methods depending on the browser or viewer they are running in. Thus in order to apply this approach one has to take care that the required native code is available on all destination platforms and that the security limitations of applets are to be considered carefully.

In order to avoid the restriction of native methods, a migration of the existing API code into pure Java can be performed. A good example for this approach is the new Java "Swing"-library, which enables a more powerful and platform independent user interface. Due to the similarity between C++ and Java, the effort for the migration is limited. In addition, there are already tools available which can automatically translate most of the existing C++ code into Java. This, however, implies for the purpose of an ASN.1 API some additional engineering effort to integrate the Java code with the communication software which is in most cases based on third party products. In this particular case, support of a full size OSI-stack or a simple RFC 1006 connection can not be found in the standard Java libraries. Thus one has to take into account the need to integrate Java with these protocols and the required processing power for such an implementation.

Additionally, with respect of dynamic download of an application across a network, the code size should be limited to avoid long periods of time for application startup. That is why an application with complex functionality should not be completely migrated into Java. For example, a vendor has used this approach to provide a Java version of his office software. Only a small Java application has to be performed on the local machine which is dynamically interconnected with a server part on a central machine. Due to the not Java based implementation of the server part no performance problem has to be expected. In addition, most of the operations will be performed completely in the Java environment and only for some complex methods native routines will be called on the server.

Considering all the advantages and disadvantages of this three approaches the realization of Java-ASN.1 API that is presented in the following is mainly based on a the client/server approach. This specification is not restricted to client/server approach.

Comparison of the Java API with other APIs

	XOM	AIDA	ASN.1++	Java-ASN.1
Originator	X-Open	Siemens	NMF	T··· Telekom
Language	ANSI-C	C++	C++	JAVA
Concept for Implementation	standalone ASN.1-API	wrapper-class for XOM	pure C++ or wrapper-class	pure Java or client / server
Type Reference	Pointer	encapsulation	encapsulation	encapsulation
ASN.1-Metadata	required	required	required	optional

Several kinds of ASN.1 API are currently available, but none of them supports Java[0] . In order to give an overview on the functionality of our Java API we will describe the individual characteristics each of the existing APIs by illustrating their features.

A well-known API for the representation of ASN.1 values has been specified by the X-Open Group and is called XOM[1] Even if the terms "object", "package", and "attribute" are extensively used in this specification, this interface is completely implemented in ANSI C. Due to the complex manual pointer handling this interface is not recommended for manual implementation of large applications.

Regarding this drawback of the XOM-Specification, several C++ wrapper classes have been defined to encapsulate this complexity. The most popular realization of wrapper classes which have nearly the same functionality, are ASN.1++ specified by the Network Management Forum[2] and AIDA from Siemens[3] . Both APIs are easy to use. The advantage of these APIs is, that every XOM structure is complete encapsulated by the objects of the wrapper classes. It should be mentioned, however, that sometimes the underlying C data structure requires a lot of functional compromises in the wrapper classes realization. Therefore a basic understanding of XOM is still required.

The current version of the Java-ASN.1 API is a first draft. It is possible - similar to the NMF approach in C++ - to implement a pure Java version of this API. Even a wrapper class concept with native method calls can be adopted. As already mentioned, a client/server approach is used for our prototype implementation. In order to get an API which is easy to use we have tried to hide as much complexity of ASN.1 as possible at the programmers interface. Due to the advantages of Java language, which implies automatic memory management and reference handling, one has not to consider these problems when implementing a Java-ASN.1 based management application.

For encoding and decoding of ASN.1 values basic knowledge of the ASN.1 type specification during runtime is necessary. For this reason every ASN.1 API (even the Java-ASN.1 API) requires a database with the ASN.1 type information which is called metadata information. In order to keep the usage of the ASN.1 API as simple as possible, we have decided to give an optional access to this required metadata information. Thus, ASN.1 values in Java can be constructed without the necessity to look up each type definition in a database and without the necessity to know platform dependent identifiers for the ASN.1 type definitions. Only when the dynamically constructed ASN.1 value should be encoded, an access to the metadata information is required. In this case, the knowledge of the ASN.1 type structure is completely hardcoded in the application. On the other hand it is feasible, to build generic applications which are completely based on metadata information. Thus, before a value will be assigned to a type, its definition structure can be determined by looking up the metadata information. It is obvious, that a violation of a ASN.1 value constraint could only be detected, when metadata information is available. Thus, only if the optional access to the metadata information has been used one will be notified about a value constraint violations before the value will be encoded.

---

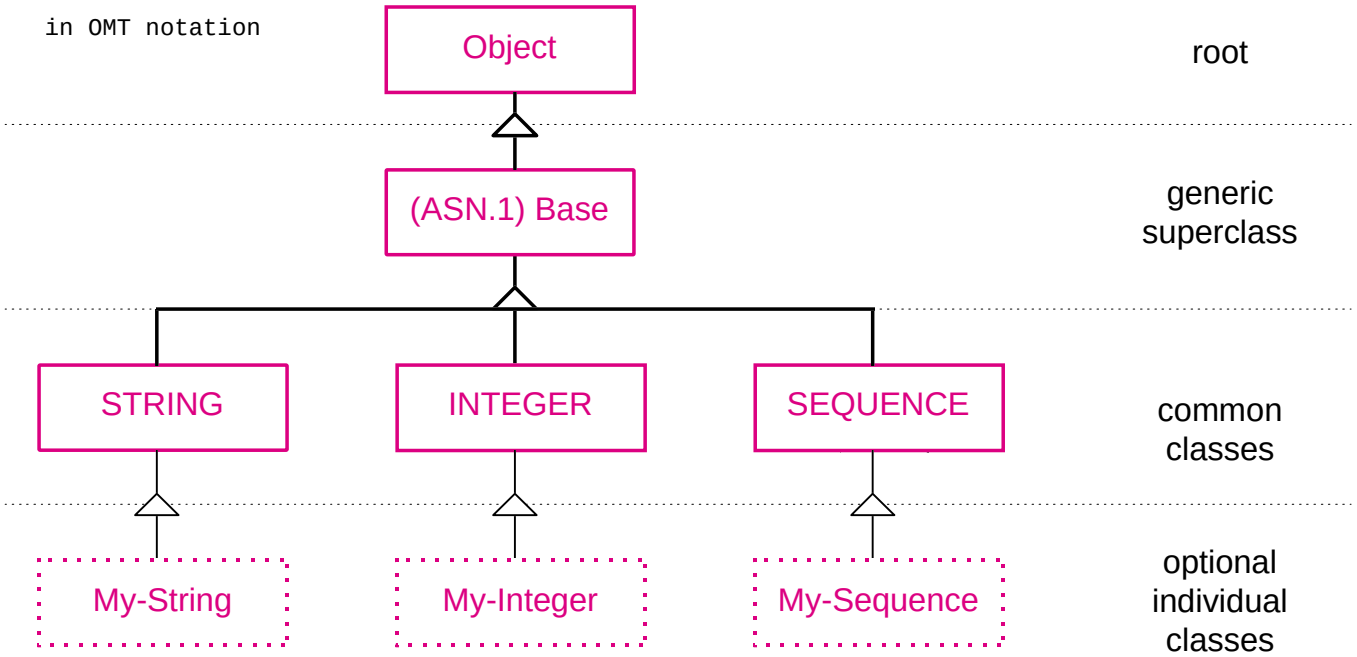
[0] "Java Development Kit Documentation (Vers. 1.1.4)", Sun Microsystems, 1997

[1] "OSI-Abstract-Data Manipulation API (XOM) Issue 3", X/Open CAE Specification, May 1996

[2] "ASN.1/C++ Application Programming Interface", NMF 040-1, Issue 1.0, July 30, 1997

[3] "Reference Manual for MOAS - AIDA - MIBRS - NE Configuration" O.N.M.S Base, Siemens, 1997

Inheritance Tree for the Java-ASN.1 classes



The inheritance tree of the Java-ASN.1 API is similar to the C++ based NMF API approach. In general, one has to consider objects at three different levels of inheritance which can be used by the application. The most common level is the generic superclass which represents common functionality of ASN.1 data types. A less generic interface to access type individual functionality is provided by common classes, which are defined for each ASN.1 universal data type individually. The most specific level of inheritance are classes which have to be generated individually from a specific ASN.1 module for each ASN.1 type definition in order to consider for example individual label assignments for subcomponents. These three levels of inheritance and their specific advantages will be illustrated in more detail in the following paragraphs.

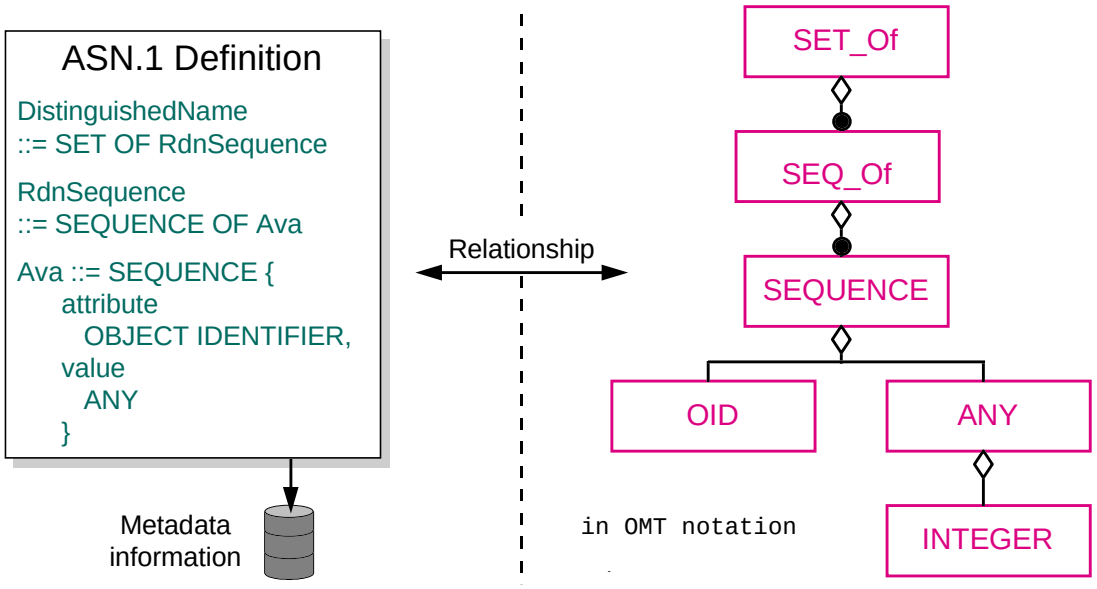
The ASN.1 BASE class does not contain the complete type individual functionality. A lot of virtual methods, however, are defined in this generic superclass, which can be specialized individually for each type in the common classes. Thus, most of the common functionalities of ASN.1 types - like encoding and decoding - are already considered in the BASE class.

Each ASN.1 type which is defined in the ASN.1 standard is represented by a common Java object class. All these classes are inherited from the generic super class BASE which in turn is inherited from OBJECT. Complex representation of values is realized by encapsulating contained elements. All required functionalities dealing with an ASN.1 value representation can be invoked directly via methods which are defined in the corresponding common class. From the API internal point of view, however, this classes might be wrapper classes, which directly invoke native code on the same or on a remote machine. Thus, depending on the selected implementation approach, the internal realization of this class could be differ in a wide range of scope. It has to be pointed out, however, that it should not be visible to the programmer, which internal implementation approach has been selected for a specific Java-ASN.1 realization.

In addition to the common classes, individual object classes might be generated for each module specific ASN.1 type. These classes are directly inherited from the corresponding common class definitions. The main advantage of this additional layer in the inheritance tree is that the common class functionality will be enhanced with individual convenience methods. One example would be a method RELATIVEDISTINGUISHEDNAME() which performs the same functionality as the standard POSITION() method in the ASN.1-sequence class. The implementation of these individual classes should be performed by an automatic algorithm.

It has to be mentioned that the generic superclass BASE can not be instantiated directly. It is feasible, however, to assign a reference to an ASN.1 class to a BASE type. Due to the reference handling concept of Java, every operation of the BASE class can be invoked on the referenced instance directly without the necessity of casting. It is also possible to cast an object which is referenced by the BASE class back to its original type. Thus the assignment of references between individual, common, and generic classes is possible in both directions without any additional engineering effort.

Representation of an ASN.1-Value in Java-ASN.1



Up to now, we have only considered the Java-ASN.1 class definitions. In order to represent the value of an individual ASN.1 type definition it is required to define the relationship between Java-ASN.1 object instances and ASN.1 type definitions. In particular, one has to consider that an ASN.1 value can be represented by a type individual class as well as with the common class definition.

Applying individual classes for type representation is quite simple and has not to be illustrated graphically. The class name is exactly the same as the type name in the ASN.1 definition. In some special cases e.g. when using a hyphen "-" in an ASN.1 type definition some minor adaptations are required. In the case of constructed type definitions -like an ASN.1 SEQUENCE e.g. - a reference to each component of this constructed value can be directly returned via methods which has the same name as the component identifier in the ASN.1 definition. Furthermore, each instance of an individual class does contain a mandatory type descriptor object, which stores additional information about the ASN.1 type definition.

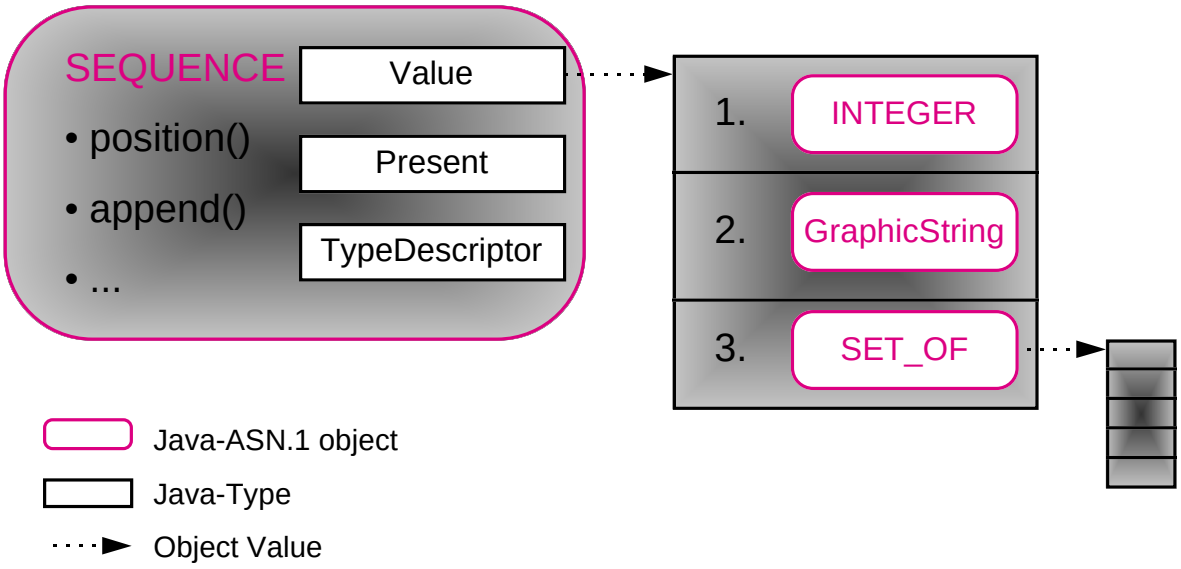
For representing ASN.1 values by common classes two slightly different approaches have to be considered: An informationless approach and an optional metadata approach. Common to both approaches is the representation of an ASN.1 value by common Java-ASN.1 classes. Both approaches differ, concerning the knowledge of the underlying ASN.1 type definitions and therefore in the possibility to detect and to avoid ASN.1 constraint violations during construction time. One could easily determine the applied approach for a Java-ASN.1 object instance by accessing the contained type descriptor object. If a null pointer will be returned, the informationless approach has been chosen.

An example for these slightly different approaches is the representation of the distinguished name. One can create and concatenate all the required object instances in two different ways. If the informationless approach is chosen, no additional type information will be required. The application has to know by itself which components are part of the type specification and how they have to be arranged. In case of the metadata approach, type information is available during runtime. Therefore, one can ask for the exact structure of the type before performing each creation and concatenation step.

Nevertheless, in both cases the application must have knowledge about the values which are represented by a specific type. Therefore, type specific information has to be coded in the application anyway. Thus, to keep the specification of the API simple, the metadata approach is only an optional feature, which has not been realized in our prototype implementation.

In order to avoid confusion about several approaches for value representation, it has to be pointed out that each Java API should support at least the easy to use informationless approach. If the metada approach is supported by an Java-ASN.1 API it has to be provided in parallel to the informationless approach. On application level, the metadata approach should be applied where appropriate for the individual task to be performed. Thus objects with and without type descriptors can exist simultaneously and the internal API realization should not rely on a type descriptor for each instance. In addition, one can not rely on the fact, that each decoded value does have a type descriptor.

Example for the usage of the Java-ASN.1 class "SEQUENCE"



For a more detailed understanding of the functionality of the Java-ASN.1 API we will investigate the internal structure of each class in more detail in the following section. Due to the similar structure of each classes inherited from BASE, it is possible to explain these functions exemplary for one type. Each Java-ASN.1 class contains several members. The most important objects and methods which are externally visible will be briefly introduced in this section.

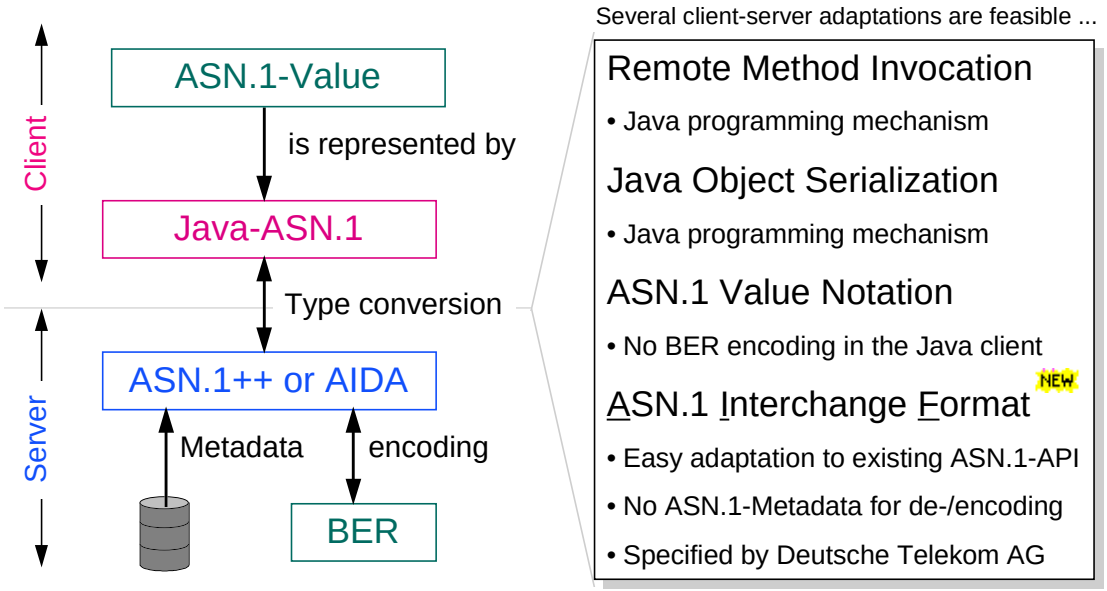
As already discussed in detail, it is not mandatory that an ASN.1 value representation in Java is always related to a specific ASN.1 type definition. Therefore, the contained element TYPEDESCRIPTOR (which represents a reference to the related type definition in the ASN.1 module) will not always be instantiated. In case of the informationless approach the member TYPEDESCRIPTOR contains only a NULL reference.

Up to now, we did not explain how a value can be stored internally in Java-ASN.1 classes. This is performed with aid of the flexible Java type VECTOR which is used for the contained object VALUE. This class is able to store one or more classes in any order. The values of simple ASN.1 types are directly contained via a appropriate Java object in this vector. In case of a structured ASN.1 data type definition - which might contain more than one component - this VALUE vector contains exactly one instance of the Java-ASN.1 class for each component which is specified in the related ASN.1 type definition. Thus in case of a valid value representation the size of the vector will be equal to number of components in the corresponding type definition. In addition, the order of the instances in the vector value should be exactly the same as the order of components in the ASN.1 type definition. For example, the value representation the following sequence type is illustrated in the figure above:

```
MYEXAMPLE ::= SEQUENCE {
        I       INTEGER,
        G       GRAPHICSTRING,
        S       SET OF INTEGER }
```

In addition, we have to consider that the ASN.1 keyword "optional" can be used in an ASN.1 type definition for a specific component. It has to be mentioned, that an omitted optional component should not be represented via a Java null reference. In order to consider this aspect in the API specification, the member PRESENT was defined. The value TRUE indicates, that this element represents a present optional type or even a mandatory component. In contrast, the value FALSE indicates that an optional component has been omitted. The main advantage to use the PRESENT attribute instead of a null reference is, that an object could be temporary set to "non present" without deleting its internal value representation. It is obvious, that the user application has the complete responsibility for the correct representation of the value. Only when type information is available during runtime (metadata approach) additional checks can be performed by the Java-ASN.1 API. Finally, it has to be noted that a non present instance of an ASN.1 ANY type is defined assignment compatible with every other non present instance due to implementation aspects of the API.

Encoding of ASN.1 values with Java-ASN.1



One major disadvantage of existing ASN.1 APIs is the requirement to load ASN.1 type information (metadata) during runtime. Even, if this information is stored in a dynamic database, a lot of additional code and system memory is required for this purpose. On the other hand, this type information can not be omitted completely due to the functional specification of the basic encoding rules (BER)[0] [1] . In order to omit the type information partially, we have split up the representation and the encoding of an ASN.1 value into two separate functional components which will be called ASN.1-client and ASN.1-server.

The client component of the ASN.1 API is completely engineered in Java. All kinds of value representations will be handled by Java objects which do not necessarily encapsulate information about the specific ASN.1 type definition. In addition, no native code is contained in our implementation of this client part. Therefore, all kinds of operations, e.g. assignment of values, will be performed completely in Java.

In our approach, the functional ASN.1-server component responsible for encoding the ASN.1 values is based on an already existing AIDA implementation. It is also feasible, to use the XOM-Interface directly or to use another C++ based API like ASN.1++. Even if the decision to use this client/server approach did originally not rely on migration aspects, it is the easiest way to reuse already existing code. Thus, this client/server approach does support migration of existing TMN APIs into Java.

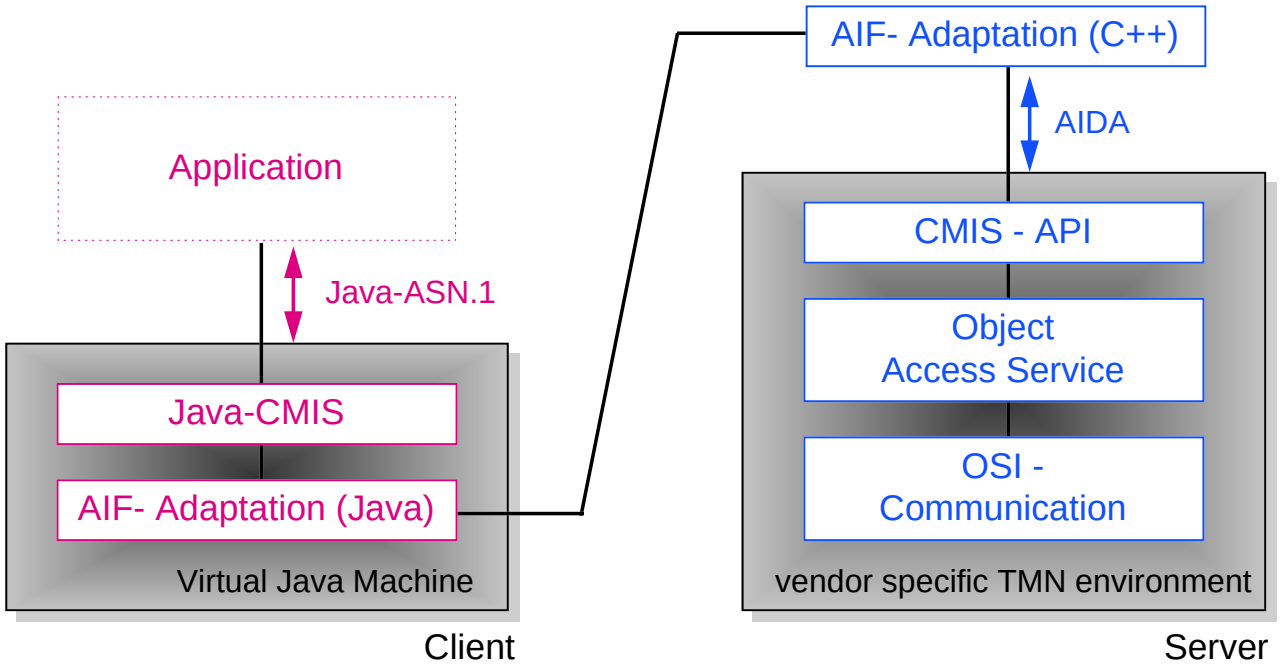
The AIDA-interface, as already mentioned, requires a detailed ASN.1 type reference for applying the basic encoding rules. For this purpose, type information has to be transmitted between the ASN.1-client and ASN.1-server. Therefore, one might think that the informationless approach, where such kind of information is not available, will not work in this area. In order to overcome this contradiction, it has to be mentioned that an ASN.1 value that is constructed in an informationless way should not be encoded directly. Instead, this ASN.1 value has to be assigned to one of the individual class (the most specific inheritance level) on the ASN.1-client side before an encoding method will be invoked. In case of a CMIS operation the corresponding ASN.1 Type (and its individual class) is called "CMIS_OperationArgument". This type individual class will simply contain a unique identifier that can be transmitted together with its value between client and server. This identifier enables the association of an ASN.1 value with the corresponding metada information in order to perform a correct value encoding on the ASN.1-server side. In practice, one can create an empty instance of an individual class (like CMIS_OperationArgument) and assign a reference to all contained elements. Thus an individual identifier has been assigned to a complex informationless constructed value without the necessity to copy one or more of the contained objects. That is why individual classes has not necessarily to be generated for each ASN.1 type definition. In order to conclude this subject, it is sufficient if the root element of a complex ASN.1 value representation is represented via a individual class. Therefore, no additional individual class has to be generated to support the encoding of a new GDMO based information model on the client side.

---

[0] "Specification of Basic Encoding Rules for Abstract Syntax Notation one (ASN.1)", ITU-T Recommendation X.209, Melbourne, 1988

[1] "ASN.1 encoding Rules: Specification of BER, CER, and DER", ITU-T Recommendation X.690 (1994) | ISO/IEC 8825-1:1995

Integration in an existing TMN - Environment



Up to now, the protocol between both functional components of the client/server relationship has not been described in detail. For encoding an ASN.1 value using BER it is required to transform the Java representation on the client side into an AIDA format on the server side. The same functionality in exactly the reverse direction is required for decoding an ASN.1 value. In order to provide this translation, we have defined the ASN.1 Interchange Format (AIF). In contrast to the basic encoding rules, no type reference is required for the AIF based encoding and decoding of the values between client and server. In addition, this interchange format can be integrated into an existing ASN.1 API with only a few additional lines of code.

It has to be mentioned, however, that several other kinds of protocols between client and server are feasible. The powerful Java mechanism of remote method invocation or object serialization might also be applied. Another approach will be to use the ASN.1 value notation.

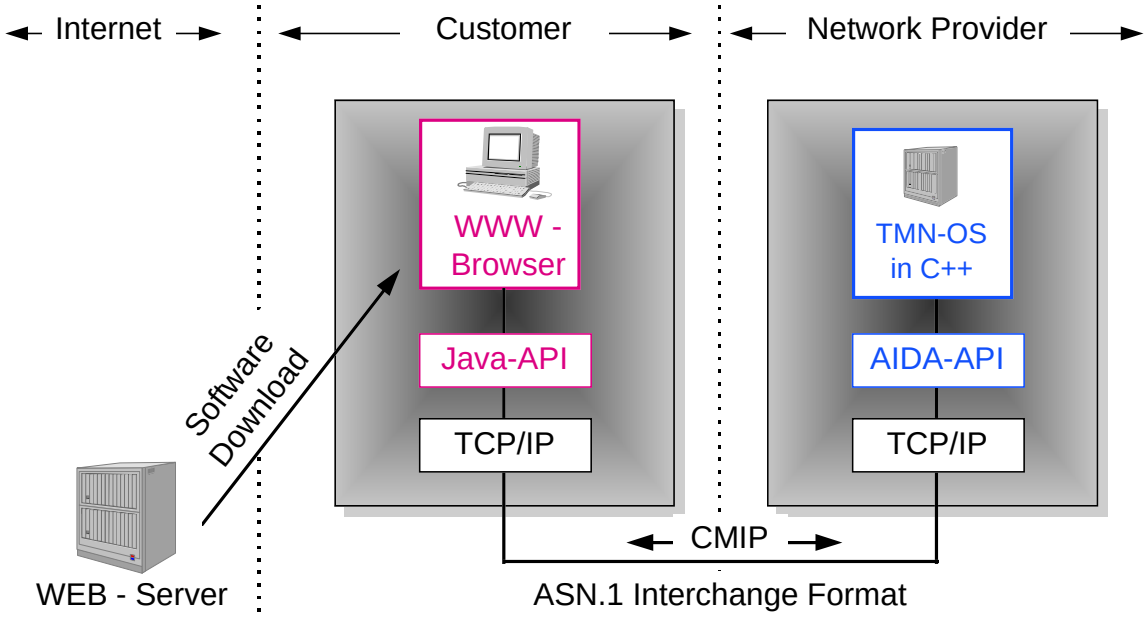
The possibility to split the ASN.1 API into two functional components, which will be interconnected via AIF, enables a totally new kind of architecture for management platforms. This can be used to design a small independent client part and a powerful server part for BER value encoding and OSI communication. The adaptation of the Common Management Information Service (CMIS)[9] into this architecture is quit easily due to the fact that the CMIS interface is completely based on a ASN.1 definition.

The platform client is programmed in pure Java and does not contain any native code. The processing of ASN.1 value representations is completely performed on the Java virtual machine. For the purpose of OSI communication, a small Java API for the Common Management Information Service (CMIS) has been specified. The main functionality of this CMIS API is to forward the requests from the client to the server. The CMIP API makes use of the AIF adaptation to ensure the correct exchange of ASN.1 values. It has to be pointed out, that even the AIF adaptation at the client side is written in Java and does not make use of native methods.

The server part is, in contrast to the Java client, implemented in a vendor specific manner. The easiest way is to use an already existing TMN platform for this purpose. That is why we have decided to use the O.N.M.S from Siemens. For integrating the AIF adaptation into this platform a small C++ application has to be provided which performs the conversion between AIDA and AIF. For our implementation we have used a generic architecture of the AIF adaptation at client and server side. Therefore, these implementations could be used for every ASN.1 type without modification or recompilation of the source files. For this purpose it is only necessary to reference at the server side a new database entry with metadata information of the related ASN.1 definition.

[9] "Common Management Information Protocol Specification for CCITT Applications", ITU-T, Recommendation X.711, Geneva 1991

Customer Network Management scenario



Due to the possibility of downloading Java bytecode at runtime Java will be widely applied in conjunction with the popular browser technology of the Internet. This offers the opportunity to use the web even in a telecommunication environment. Thus a browser oriented usage of the Java-ASN.1 API seems to be appropriate.
Today it is already possible for a customer to exchange management information across the Internet with a network provider. Due to the complicated integration of these services into an existing TMN environment this technique has not been widely adopted yet. By using Java in conjunction with the proposed Java ASN.1 API a network provider will be able to offer functionally more complex Customer Network Management (CNM) applications to the customer without the necessity of any investment in platforms on the customer side. It is obvious that from the customer point of view these applications are not novel, because the only difference is the kind of information which is exchanged between service provider and browser at the client side. The network provider, however, is able to integrate a CNM- application into the standard management platform due to the usage of CMIP for this purpose. The customer will be able to send TMN conform protocol data units to the network provider without the requirement of maintaining a full seven layer OSI stack itself. All software that the customer will need for this purpose can be downloaded from an Internet server. The exchange of management information will be done with aid of AIF which will be transmitted over TCP/IP. That is why the network provider is able, to easily offer powerful services directly to the customer via the Internet.

Conclusion

In this paper we have discussed the possibility to use Java realizing TMN management applications. We have identified the requirement for dealing with ASN.1 datatypes on a Java virtual machine. For this purpose, we have analyzed the specific benefits of concepts for ASN.1 APIs and have discussed the possibility to reuse these concepts in Java. In order to get the best technical solution for an Java-ASN.1 API, several alternatives for migration existing C++ based implementations into Java have been considered.
As a result of this investigation, we have specified a Java-ASN.1 API which is able to reuse existing software components. In addition, this API was specified in a flexible way, to allow the integration of ASN.1 type information (metadata) in several ways. In order to enable a lean applet implementation, this API has been realized in a client/server approach.
To prove the concept, we will evaluate the new Java-ASN.1 API by a prototype application in the area of customer network management. For this purpose, we will use specific advantages of Java language, as e.g. the simplified creation of a platform independent user interface.