System tests for Enterprise JavaBeans and Java based Applications with TTCN-3

Guy Collins Ndem[1], Ina Schieferdecker[1,2], Hajo Eichler[1], Alain Vouffo-Feudjio[1]

Fraunhofer FOKUS, MOTION
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

[2]Technical University Berlin, Faculty IV,
Straße des 17. Juni 135
10623 Berlin, Germany

{guy-collins.ndem, schieferdecker, eichler, vouffo}@fokus.fraunhofer.de

http://www.fokus.fraunhofer.de/motion

Abstract. Nowadays the use of complex distributed systems increases rapidly and the need for solid testing, too. Enterprise JavaBeans (EJB) is one of the most used standards for developing distributed systems. The applications based on EJB technology are secure, concurrent, transactional, etc. TTCN is in the telecommunication domain a widely established and used test technology. In its new version, TTCN-3 (Testing and Test Control Notation), it has a wider scope and applicability. It cannot be used only for testing the conformance and interoperability of communication protocols but also for testing the correct behaviour of distributed systems and application. This paper describes the language mapping from EJB/Java to TTCN-3 and an Eclipse based solution for testing EJB/Java based applications with TTCN-3.

1 Introduction

Testing is a generally accepted approach to validate systems and system components in their development and target environment. Assured quality of system and system components is particularly important as the time-to-market becomes ever shorter and the requirements on system functionality, reliability, availability, integrity and performance further increase. A systematic approach to testing distributed systems is essential, so that the requirements of the market can be fulfilled.

One of Java's most important features is platform independence. Enterprise JavaBeans (EJB [11][12]) is not just platform-independent; it is also implementation-independent. The EJB has revolutionized the way we develop mission-critical enterprise software. It combines server-side components with distributed object technologies, asynchronous messaging, and Web services to simplify application development. It automatically supports many of the requirements of business systems such as security, resource pooling, persistency, concurrency, scalability, portability, or transactional integrity.

TTCN-3 [1] is an implementation-independent test specification and implementation technology, specifically developed to support the testing of static and dynamic, local and distributed, sequential and parallel reactive systems. It is already links into object, component and Web service technologies with the IDL to TTCN-3 [6], the XML to TTCN-3 mapping [8], and the upcoming C/C++ to TTCN-3 [9] mapping. However, a mapping to enable the testing of EJB/Java-based applications is missing. This is the target of this paper.

In this paper, we describe a "*Three-Component*" system test architecture approach for testing dynamically EJB based applications with TTCN-3. The *Loader component* is able to load SUTs from a given location. The *Transformer component* will analyze the loaded SUTs and according to their functionalities, a TTCN-3 document object model (TDOM [10]) containing each functionality and its associated test case will be generated. This component is also able to generate TTCN-3 codes from the generated TDOM. Finally the *TestAdapter component* is a bridge between TTCN-3 and SUT(s) during the execution of test cases. TDOM enables testing of systems without the necessity to see the associated TTCN-3 codes.

The paper is organized as follows. Section 2 respectively Section 3 addresses TTCN-3 respectively EJB. Section 4 presents basic concepts of the combined use of Java and TTCN-3. The deal with our implementation is discussed in Section 5. Session 6 provides a concrete example and Section 7 summarizes the paper.

2 Testing and Test Control Notation (TTCN-3)

The TTCN-3 language was created due to the imperative necessity to have a universally understood (specification and implementation) language syntax able to describe test behaviours and test procedures. Its development was imposed by industry and science to obtain a single test notation for all black-box and grey-box testing needs. In contrast to earlier test technologies, TTCN-3 encourages the use of a common methodology and style which leads to a simpler maintenance of test suites and test systems. With the help of TTCN-3, the tester specifies the test suites at an abstract level and focuses on the test logic to check a test purpose itself rather than on the test system adaptation and execution details. A standardized language provides a lot of advantages to both test suite providers and users.

TTCN-3 enables systematic, specification-based testing for various kinds of tests including e.g. functional, scalability, load, interoperability, robustness, regression, system and integration testing. It allows an easy and efficient description of complex distributed test behaviours in terms of sequences, alternatives, and loops of stimuli and responses. The test system can use a number of test components to perform test procedures in parallel. TTCN-3 is characterized by a well-defined syntax and operational semantics, which allow a precise and unambiguous test execution. The task of describing dynamic and concurrent configurations is easy to perform. The communication can be realized either synchronously or asynchronously. In order to validate the data transmitted between the entities composing the test system, TTCN-3 supports the definition of templates with powerful matching mechanism. To validate the described behaviours, a verdict handling mechanism is provided.

The types and values can be either described directly in TTCN-3 or can be imported from other languages (e.g. ASN.1, XML schema, or IDL). Moreover in TTCN-3, the parameterization of templates, functions, test cases, modules, etc. is allowed. The selection of the test cases to be executed can be either controlled by the user or can be described within the execution control construct. The external configuration of a test suite through module parameters is possible.

3 Enterprise JavaBeans (EJB)

Enterprise JavaBeans provided by Sun Microsystems was first introduced as a draft specification in late 1997. It has since then established itself as one of the most important Java enterprise technologies. EJB is a standard server-side component model for distributed business applications. It provides the server-side with the Component Transaction Monitors (CTM) representing two technologies: traditional transaction-processing (TP) monitors (such as CICS, TUXEDO, and Encina), and distributed object services (such as CORBA, DCOM, and native Java RMI). Combining the both technologies, CTMs provide a robust, component-based environment that simplifies distributed development while automatically managing the most complex aspects of enterprise computing, such as object brokering, transaction management, security, persistence, and concurrency. In order to implement entity and session enterprise beans, one needs to define the component interfaces, a bean class and a primary key:

The Remote (Home) interface defines the bean's business methods which can be accessed from applications outside the EJB container. The *Remote (Home) interface* is the most important element when testing EJB based application with TTCN-3. *The Local (Home) interface* defines business methods that can be use by other beans in the same EJB container. *The Endpoint interface* defines business methods that can be accessed from outside the EJB container via SOAP. *The Message interface* defines the methods by which messaging systems, such as JMS, can deliver messages to the bean. *The Bean class (Entity, Session)* must have methods matching the signatures of the methods defined in the *remote, local, endpoint interfaces,* and must have methods corresponding to some of the methods in both *remote and local home interfaces. The Primary key* is a class that provides a pointer into the database. Only entity beans need a primary key.

4 Java to TTCN-3 Mapping

This section defines the mapping rules for Java to TTCN-3 to enable testing of Java/EJB based applications. The following mapping rules are an initial step to a Java-to-TTCN-3 mapping. The rules do not cover the whole Java Core Language, but all parts needed for EJBs.

4.1 Lexical Convention

The lexical conventions of Java define comments, identifiers, keywords, and literals conventions which are described below.

Comments. Comment definitions in TTCN-3 and Java are the same and therefore, no convention for the mapping is necessary. Both Java and TTCN-3 are using the pair "/*" and "/*" for comment blocks or "//" for end line comments.

Identifiers. Identifiers in Java and TTCN-3 consist of alphabetic, numeric and underscore characters where the first character must be an alphabetic character and all characters are significant. Both Java and TTCN-3 use case sensitive identifiers. However Java supports overloading of identifiers according to the context in which they are defined. There is no overloading of identifiers in TTCN-3 so that no identifier name can be used more than once in a scope hierarchy. Hence, the identifiers in TTCN-3 need to reflect the overloading situation in Java.

Keywords. It has to assured that no TTCN-3 keyword is used as an identifier in the Java definition if a seamless mapping has to be guaranteed. For that, identifiers are renamed by appending a special prefix or suffix in case of a conflict with keywords in TTCN-3.

Literals. The definition of literals differs slightly between Java and TTCN-3 so that some conversions to be made. The Table 1 gives the mapping for each literal type.

Literal Group	Literal	Java Convention	TTCN-3 Convention
Numeric	`long`	no "0" as first digit	no "0" as first digit
	`integer`	no "0" as first digit	no "0" as first digit
	`short`	no "0" as first digit	no "0" as first digit
	`byte`	no "0" as first digit	no "0" as first digit
	`octal`	"0" as first digit	'FF96'O
	`hexadecimal`	"0X" or "0x" as first digit	'AB01D'H
Floating	`float`	1; 1.5f; 1.5F; 2e-5f; 2e-5F	1.0; 1.5; 2E-5
	`double`	1d; 1D; 1.5; 2e-5	
Others	`boolean`	true, false	true, false
	`char`	'c'	"c"
	`String`	"text"	"test"

Table 1. Literal mapping

4.2 Mapping Java Names to TTCN-3 Names

In general, each Java name is mapped to an equivalent name in TTCN-3 (Java and TTCN-3 are both case sensitive). However, there are some exceptions when the Java name is not a legal identifier in TTCN-3.

Mapping packages to modules

We map Java package names to TTCN-3 modules. Each Java package becomes a separate TTCN-3 module. Packages within packages will be mapped as new modules. Therefore, a Java package `a.b.c` would turn to into a TTCN-3 module `a_b_c` and `a_b.c` would be mapped to `a__b.c.`

Mapping Java names that clash with TTCN-3 keywords

For Java names that collide with TTCN-3 keywords, the Java names are mapped to TTCN-3 by adding a leading `"j"` word. So the Java name `oneway` is mapped to the TTCN-3 identifier `j__oneway`.

Mapping Java names with leading underscores

For Java names that have leading underscores, the leading underscore is replaced with `"j_"`. So `_fred` is mapped to `j_fred`. An existing underscore in an identifer is use as a separator and it is replaced by a double underscore. So `a_b` is mapped to `a__b`. An existing leading `"j_"` is mapped to `U006A` (Unicode for "`j`").

Mapping Java names with illegal TTCN-3 identifier characters

Given the current lack of support for Unicode in TTCN-3, we define a simple name mangling scheme to support the mapping of Java identifiers to TTCN-3 identifiers. For Java identifiers that contain illegal TTCN-3 identifier characters such as `'$'` or `"U"` replaces Unicode characters outside of ASCII, any such illegal characters followed by the 4 hexadecimal characters (in upper case) representing the Unicode value. So, the Java name `a$b` is mapped to `aU0024b` and `x\u03bCy` is mapped to `xU03BCy`.

Names for inner classes

In Java, the name of the inner class is composite name formed by concatenating the name for the outer class, the dollar character `"$"`, and the name of the inner class. When we are mapping names for Java inner classes, we can refer to the mapping for illegal identifiers. The corrections for illegal TTCN-3 identifiers described above are then applied. For example, an inner class `Fred` inside a class `Bert` will be mapped to a TTCN-3 name of `BertU0024Fred`.

Overloaded methods names

If a Java method's names are not overloaded, then the same method's names are used in TTCN-3 as were used in Java. Given the absence of overloaded methods in current TTCN-3, we define a simple name mangling for overloaded methods. Note that a method may be uniquely defined in a base interface (and therefore its name will not be mangled in that interface) and then be overloaded in a derived interface (in which case the name will be mangled in the derived interface). For overloaded Java methods, the mangled TTCN-3 name is formed by taking the Java method name and then appending each of the qualified TTCN-3 types of the arguments separated by two underscores, followed by "_signature". For example, the two overloaded Java methods: `hello()` is mapped to `hello_signature()` and `hello(int x,` `a.b.c y, long z)` is mapped to `hello__int__c__long_signature(in JavaInt x, in a_b_c y, in JavaLong z).`

Methods names that collides with other names
In some cases, applying these rules for name mappings can generate TTCN-3 with collisions between method names and constant or field names. This is because Java constants and fields can have the same names as methods, but TTCN-3 constants and fields cannot. The following rules are used to avoid such name collisions in TTCN-3: Method, constant or field names are mapped unchanged (subject to other mangling rules). For example, if a Java class has both a constant `fb` and a method `fb()`, the TTCN-3 method is called `fb_signature` (if it is mapped) and the TTCN-3 constant is called `fb` (whether or not the method `fb` is mapped).

Container names that clash with their members
In some cases, applying these rules for name mappings would generate TTCN-3 with collisions between a container name and members of the container. This is because a Java member can have the same name as its container, but TTCN-3 members cannot. The following rules are used to avoid such name collisions in TTCN-3: Container names are mapped unchanged (subject to other mangling rules). Java method, constant, or field names whose mapped name collides with the mapped name of their Java container are mapped according the general mapping rules. For example, if a remote Java interface `Foo` has a method `foo`, the TTCN-3 interface is called `Foo` and the TTCN-3 operation is called `foo_signature`.

Names that would cause TTCN-3 collisions
If the name mappings defined in this specification would produce TTCN-3 method, constant, field, or attribute names that are not unique within their declared scope, this is treated as an error. For example, if a Java remote interface has methods `foo()`, `foo(int x)`, and `foo__int()`, the corresponding TTCN-3 names would be `foo_signature`, `foo__long_signature`, and `fooU002Dlong_signature` `(normally foo__int_signature)`, which is legal TTCN-3.

4.3 Mapping Java Types to TTCN-3 Types

Java provides type declarations for values and constants and basic data types, constructor types and complex types. Their mapping to TTCN-3 will be shown in the following subsections. A construct for naming data types and defining new types by using the keyword class is provided by Java. This can be done under TTCN-3 via the keyword type, too.

Mapping for Primitive Types
Mapping Java primitive data types to TTCN-3 data types is straight forward. It is necessary to map each Java primitive type to a new appropriate one TTCN-3 in order to ensure their reconstruction in Java during the execution of tests. This notion introduces the JavaObject@TTCN-3 concept, which consists to transport the Java

Objects in TTCN-3 and to manipulate them as well as in Java. Table 2 shows mapping of those primitive types and their derived types in TTCN-3.

Java Type	TTCN-3	TTCN-3(derived)
`void`	`–`	-
`Boolean, boolean`	`boolean`	-
`Character, char`	`char`	-
`Byte, byte`	`JavaByte`	`type integer JavaByte …;`
`Short, short`	`JavaShort`	`type integer JavaShort …;`
`Integer, int`	`JavaInt`	`type integer JavaInt …;`
`Long, long`	`JavaLong`	`type integer JavaLong …;`
`Float, float`	`float`	`type float JavaDouble …;`
`Double, double`	`JavaDouble`	`type float JavaDouble …;`

Table 2. Java primitive Type to TTCN-3

Mapping for java.lang.String and java.lang.Object
As in many programming language, a String is a sequence of char. Therefore, `java.lang.String` is mapped to `universal charstring` in TTCN-3 and `java.lang.Object` is to a `record` see Table 3 for more information.

Java Type	TTCN-3	TTCN-3(derived)
`java.lang.String`	`universal charstring`	-
`java.lang.Object`	`JavaObject`	`type record JavaObject {` `universal charstring name` `}`

Table 3. Mapping for java.lang.String and java.lang.Object

Mapping Java Classes and inner Classes
A Java class represents objects and their properties. Therefore, a class is mapped according its complexity to a record. When we are testing EJB, a class or inner class is only mapped if it's instantiated. The fields of class are carried to TTCN-3 according their accessibility i.e. a final field is transported only if it is public and a private field is carried only if the delegated get and set methods are implemented

Java	TTCN-3

`package alpha.beta`
`class Jet{`
`private String name;`
`public double height;`
`private Jumbo jbo;`
`…`
`private class Jumbo{`
`  public float value;`
`  }`
`}`
	`type record`
	`alpha_beta_JetU0024Jumbo {`
	`   float value;`
	`}`
	`type record alpha_beta_Jet{`
	`universal charstring name,`
	`JavaDouble height,`
	`alpha_beta_JetU0024Jumbo jbo`
	`}`

Mapping for Arrays and Lists
Java array can be mapped directly to the TTCN-3 array type because they provide the same functionality. However the dimension of the array is unknown during the transformation from Java to TTCN-3 and therefore we mapped a Java array to a TTCN-3 record of. The List types are Java elements which are (sub)classes having (sub)interfaces of `java.util.Iterator` or `java.util.Map` such as *Vector, List, ArraList, Hashtable,* etc. Lists are mapped to record of in TTCN-3 (see below for more details).

Java	TTCN-3

`Book[ ] ;`	`type record Book{…}`
	`type record of Book Book_ARRAY;`

`Book[ ][ ];`	`type record of Book innerArray;`
	`type record of innerArray Book_ARRAY;`

`java.util.Vector ;`	`type record of JavaObject`
	`java_util_Vector;`

`java.util.HashTable;`	`type record of JavaObject`
	`java_util_Hashtable;`

Mapping for Java Exceptions
In Java, exceptions are used in conjunction with operations to handle exceptional conditions during an operation call. Thus, a special struct-like **exception** type is provided which has to be associated with each operation that can trigger this exception. Java allows subclassing of exception types and the Java types system is used to distinguish different flavors of exceptions at run time. It is very common for a Java Interface to say it raises a fairly generic exception (such as java.io.IOException) but for implementations to throw more specific subtypes (such as java.io.InterruptedIOException) and for clients to use instanceof operator to check for specific subtypes. TTCN-3 also supports the use of exceptions with procedure calls by binding it to signature definitions. However, it provides no special **exception** type. Hence, exceptions are defined by using type **record**.
A definition of an **exception** is shown in the following example. The use of exception binding in signature definitions and exception catching is shown in the context of operation declaration.

Java	TTCN-3

`Exception`	`type record StackTraceElement {`
	`  universal charstring className,`
	`  universal charstring methodName,`
	`  JavaInt lineNumber };`
	`type record of  StackTraceElement`
	`StackTrace;`

	`type record Exception {`
	`     universal charstring reason,`
	`     StackTrace stackTrace };`

Mapping for Java Package

A Java package is mapped to a TTCN-3 Module

Java	TTCN-3
`package alpha.bravo;`	`module alpha_bravo { … }`
`package alpha.bravo.fly;`	`module alpha_bravo_fly { … }`

Mapping for Java Interfaces

Interfaces describe objects with all their access methods by using operations and parameters. Additionally, interfaces can contain local type definitions like exceptions and constants which can be used by its operations and attributes. A mapping for interfaces should provide a similar scoping and grouping mechanism as well as an appropriate handling under TTCN-3 as in Java. Because of lacking an object model in TTCN interfaces have to be flattened and all interface definitions are stored in one group. Hence, import of single interface definitions from other package via the importing group statement is possible. Therefore, a Java interface is mapped to group, port, template and component.

Java	TTCN-3

`Interface identifier`
`{ body }`
	`group identifierInterface {`
	`   ... body definitions ...`
	`  type port identifier`
	`  signature { ... }`
	`  type template identifierObject;`
	`}`

Mapping for Java Methods

Operations (methods) are the main part of interface definitions in Java and are used, for instance, in the Java scheme as procedures (or functions) which can be called by clients. Methods under Java consist of invocation semantics, return results, identifier, parameter list, optional raise expression. The matching of all this parts to TTCN-3 will be described now.

Java	TTCN-3

`class Foo{`
`boolean isMax(int a,`
`int b)  throw`
`java_rmi_exception;`
`}`
	`type record java_rmi_exception{`
	`  universal charstring reason,`
	`  StackTrace stackTrace`
	`};`
	`Foo__isMax_signature(in JavaInt a, in`
	`JavaInt b)`
	`return boolean`
	`exception(java_rmi_exception);`

Mapping for Non-conforming Classes and interfaces
Non-conforming classes or interfaces are the data types which have not been mentioned above. Those types are mapped to JavaObject in TTCN-3 and therefore `java.io.File` is mapped to `type JavaObject java_io_File`.

5 EJB System tests architecture

This section describes our implemented approach for testing EJB based applications with TTCN-3. Fig.1 shows the system test environment, which is described in the following:

– SUTs: EJB applications are Java based and therefore they can be provided as *jarFiles, zipFiles, or classFolders.*
– SUTLoader is one of the components described above. The SUTLoader able load more than one SUT once.
– SUTParser is the transformer component. It uses the mapping rules to generate dynamic test cases.
– TDomUnparser is a part of the SUTParser. It's able to transform and save a generated TDom to TTCN-3 codes.
– TestAdapter is a bridge between TTCN-3 and SUTs [3][4].

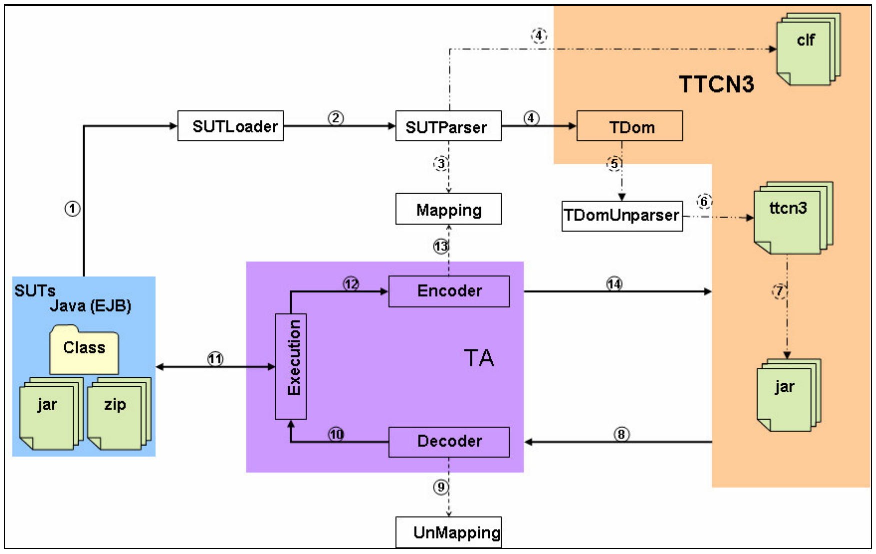

Fig. 1. System test architecture

5.1 SUTLoader (SL)

The SUTLoader is based on Java Class loading technology and is able load more than one SUT at once. This singleton application is the only loader component used during the transformation from Java-to-TTCN-3 because it also provides the functionalities of the Classic Java Classloader (ClassLoader and URLClassLoader). The SUTs loading mechanism is describe as follows:

– The default SUTs location is checked while the loader is initializing and its contents is loaded. The loader provides methods and functionalities to loads more SUTs after the initialization.
– The SUTs are sorted in four different lists: *"All"* (to test Java), *"Only interfaces"* (to test concrete Java's functionalities), *"Only EJB interfaces"* (to test All EJB including the base functionalities), *"Only EJB Remote interfaces"* (to test EJB based applications).
– According the test mode, SUTs are compressed in a SUTHastable containing more SUTPackages and passed to the transformer.
– Each SUTPackage contains the classes of the same Java package.

5.2 SUTParser (SP)

The SP can receive SUTs as a Java Hashtable, a SUTHashtable, a SUTPackage, or a Java Class. The base SUT entity is the SUTPackage, because the top-level unit in TTCN-3 is the module. A Java Hashtable is automatically converted to SUTHastable and each SUTPackage of a SUTHastable will be used to define a new TTCN-3 module. A Java class will be converted to a SUTPackage with one class. Fig.2 shows the internal structure of the Parser.

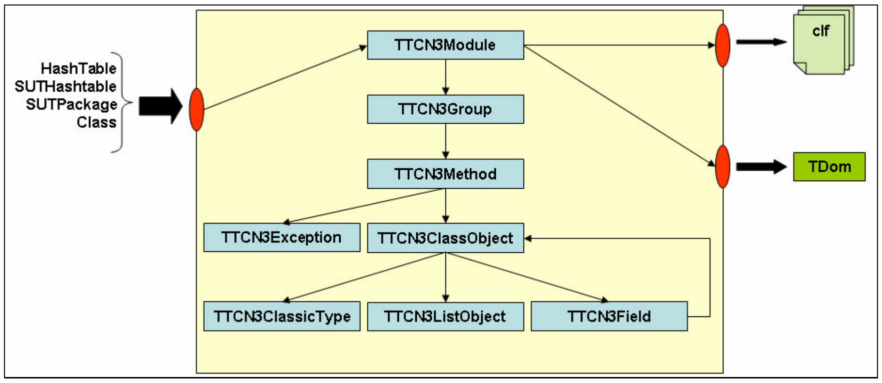

Fig. 2. Structure of the SUTParser

- TTCN3Module is the top-level unit of the parser, it receives a SUT module as SUTPackage, and each of its elements defines a new TTCN3Group.
- TTCN3Group is the representation of a Java class or interface, each methods of a class is used to define a TTCN3Method.
- TTCN3Method is mapped to a signature in TTCN-3; each can have parameters, a return value, and can throw exceptions. Each parameter type or return type defines a TTCN3ClassObject, and each exception type is used defines a TTCN3Exception
- TTCN3ClassObject can be simple (primitive) or complex. According to its complexity, a TTCN3ClassObject can be a *classic* or a *list* type.

The SUTParser provides also elements such as, TTCN3Field, TTCN3ListObject, TTCN3Exception, and TTCN3ClassicType.

5.3 Test Adapter (TA)

One of the advantages of TTCN-3 is the availability of both standardized open runtime and control interfaces (TRI and TCI) that enable an exchange of tool components between different test tool providers. The execution of every TTCN-3 ATS against a SUT requires the use of a TTCN-3 compiler and runtime environment but also the implementation of a SA according to the standardized TTCN-3 runtime interface [TRI] definition, i.e. particular methods like triMap, triSend etc. have to be implemented. We applied a Java-based TTCN-3 test tool environment. An additional major issue for the adaptation of the SUT and the test system is the implementation of a coder/decoder. It is possible to semi-automate this task by generating the codec. In this case extra information required by the CD on particular message identifiers need to be provided via extended type information using again the TTCN-3 “with” statements. The Implemented EJB/Java TA consists in three different components describe bellow:

- The *Decoder* provides a reconstruction mechanism, which is able to transform TTCN-3 values to original JavaObjects.

- The *Encoder* provides a construction mechanism, which is able to transform JavaObjects to expected TTCN-3 values.
- The *Execution* communicates directly with the SUTs. It is able to request execution of tasks and the results are forwarded to the TTCN-3 test manager [TTman] via the Encoder, the [TTman] will analyze the results and will set verdict according the expected behaviour.

6 An example

The following example – a book search example – shows the content of a generated TDOM so that we see how the transformer generates dynamically test cases for testing a given EJB based application.

The EJB remote interface `BookSession` contains the functionality to be tested:

```
package de.fokus.ejbbook.session.interfaces;
  public interface BookSession extends
   javax.ejb.EJBObject {
  //function to test
     public de.fokus.ejbbook.view.BookView[]
                getAllBooks(java.lang.String author,
                  java.lang.String title,
                 java.lang.String isbn) throws
              java.rmi.RemoteException;
}
```

Like in the other TTCN-3 language mappings, the Java mapping uses a module `JavaAUX` for the auxiliary type definitions:

```
module JavaAUX {
  type integer JavaByte (-2E7..2E7-1);
  type integer JavaShort (-2E15..2E15-1);
  type integer JavaInt (-2E31..2E31-1);
  type integer JavaLong (-2E63..2E63-1);
  …
  type record JavaObject {
     universal charstring name
   }
  type record StackTraceElement {
       universal charstring className,
       universal charstring methodName,
       JavaInt lineNumber
  }
  type record of StackTraceElement StackTrace;
  type charstring address;
  type charstring DEFAULT_EXCEPTION;

  . . . more declarations . . .
}
```

A TTCN-3 module specifically for the functionality to be tested is generated. The module name `de_fokus_ejbbook_session_interfaces` is derived from the Java package name:

```
module de_fokus_ejbbook_session_interfaces {
 import from JavaAUX all;//default declarations
 … //more module parameters declarations . . .
```

The exception is mapped to a reason together with the whole stack trace so as to enable the analysis of the exception trace also during testing:

```
//mapping for the exception
type record java_rmi_RemoteException {
   universal charstring reason,
   StackTrace stackValue}
```

The data being used to characterize a book are defined in record structures:

```
//mapping for BookPK in BookValue
 type record de_fokus_ejbbook_entity_interfaces_BookPK {
   universal charstring iSBN}

//mapping for BookValue in BookView
 type record de_fokus_ejbbook_entity_interfaces_BookValue{
   universal charstring author,
   JavaInt editionNumber,
   universal charstring imageFile,
   universal charstring ISBN,
   universal charstring title,
   universal charstring titleName,
   JavaDouble price,
   de_fokus_ejbbook_entity_interfaces_BookPK primaryKey}
```

This type is the root type for the `BookView` function that is the target of the test:

```
//mapping for BookView as roottype for BookView[]
type record de_fokus_ejbbook_view_BookView {
   de_fokus_ejbbook_entity_interfaces_BookValue bookValue}
```

A list of books is stored in `record of` structures:

```
//mapping for BookView Array
type record of de_fokus_ejbbook_view_BookView
   de_fokus_ejbbook_view_BookView__1__ARRAY;
```

A `BookSession_group` group is generated to cover all definitions relating to the `BookSession` interface such as the signature for retrieving books, the port for the communication with the SUT and external functions for the creation and deletions of beans:

```
//mapping for Interface/class as group
group BookSession_group {
//mapping for methode within group as signature
```

```
 signature
  BookSession__getAllBooks__String__String__String_signature
     (in universal charstring Par0,
      in universal charstring Par1,
        in universal charstring Par2) return
     de_fokus_ejbbook_view_BookView__1__ARRAY exception (
     DEFAULT_EXCEPTION,       java_rmi_RemoteException);
 }

 … // template declarations . . .
//definition for the port declaration
type port BookSession_porttype procedure {
 out BookSession__getAllBooks__String__String__String_signature;
}
//definition for the component
type component de_fokus_ejbbook_session_interfaces__BookSession
   {port BookSession_porttype BookSession_proc_port}

//some external functions
external function createCMPBean(in charstring ejbName) return address;
external function deleteCMPBean(in address id);
```

Finally, a test case for `getAllBooks` is generated that invokes the procedure and awaits a correct reply. The test data for the correct reply use default values. Whenever an incorrect response is received such as an incorrect reply, an exception or no response at all, a `fail` verdict will be assigned:

```
//generating the test case for the functionality
testcase BookSession__getAllBooks__String__String__String_testcase()
  runs on de_fokus_ejbbook_session_interfaces__BookSession
  system de_fokus_ejbbook_session_interfaces__BookSession {
   map (self: BookSession_proc_port,
        system: BookSession_proc_port);
 var address id :=
createCMPBean("de_fokus_ejbbook_session_interfaces__BookSession");
 var charstring exceptionMessage;
 var java_rmi_RemoteException java_rmi_RemoteExceptionMessage;
BookSession_proc_port.call(
   BookSession__getAllBooks__String__String__String_signature:
   BookSession__getAllBooks__String__String__ String_signature_template,
       5.0) to id {
 [] BookSession_proc_port.getreply (
    BookSession__getAllBooks__String__String__String_signature:
    BookSession__getAllBooks__String__String__String_signature_template
       value de_fokus_ejbbook_view_BookView__1__ARRAY_template) {
          setverdict (pass);
          log ("SUCCESSFUL...");
       }
 [] BookSession_proc_port.getreply {
          setverdict (fail);
         log ("UNEXPECTED VALUE WAS RETURNED...");
       }
 [] BookSession_proc_port.catch (
     BookSession__getAllBooks__String__String__String_signature,
     java_rmi_RemoteException :? ) -> value
     java_rmi_RemoteExceptionMessage {
         setverdict (fail);
         log (valueof (java_rmi_RemoteExceptionMessage.reason));
         }
[] BookSession_proc_port.catch (
     BookSession__getAllBooks__String__String__String_signature,
     DEFAULT_EXCEPTION: ?) -> value exceptionMessage {
         setverdict (fail);
         log (exceptionMessage);
        }
[] BookSession_proc_port.catch (timeout) {
           setverdict (fail);
          log ("TIMEOUT OCCURRED...");
        }
    }
}
```

This concludes the example. Because of the absence of overloading, inheritance and packaging in TTCN-3, the generated names in TTCN-3 tend to become long during the flattening. However, flattening the names is the only way to prevent name clashes within the generated TTCN-3. The mapping rules and the tools described beforehand have been successfully used to test a more complex EJB book store case study dynamically.

7 Summary

Our work was primarily focused on the definition of a set of Java-to-TTCN-3 mapping rules, in order to support an adequate testing of EJB based systems. The defined mapping rules focus on the translation of type and structural information contained in Java class and interface definitions.
Secondly, we implemented a *three-component* based system namely a SUTLoader for loading the SUTs, a SUTParser to apply the transformation from Java to TTCN-3 using TDOM, and a generic Test Adapter to ensure a bi-directional translation between Java and TTCN-3.
Future work will be to complete Java-to-TTCN-3 mapping rules, in order to test all Java based applications.

Acknowledgments

We like to thank our colleagues Edzard Höfig, Yuri Glickmann, Jacob Wieland, Dimitrios Apostolidis, Chaparadza Ranganai, Titus Ngwangwen, Laurette Lacmene, and Maik Busch for their advices and support.

References

[1] ETSI ES 201 873-1 V2.2.1 (2003-02): TTCN-3 Core Language.

[2] ETSI ES 201 873-4 V2.2.1 (2003-02): TTCN-3 Operational Semantics.

[3] ETSI ES 201 873-5 V1.1.1 (2003-02): TTCN-3 Runtime Interface (TRI).

[4] ETSI ES 201 873-6 V1.1.1 (2003-07): TTCN-3 Control Interface (TCI).

[5] ETSI DTS/MTS-00080 V1.2.1 (2003-07): The IDL to TTCN-3 Mapping

[6] M. Ebner, A. Yin, and M. Li: Definition and Utilisation of OMG IDL to TTCN-3 Mappings, In TESTING OF COMMUNICATING SYSTEMS XIV --- Application to Internet Technologies and Services, ed. I. Schieferdecker, H. König and A. Wolisz. number 14, pp. 443--458, IFIP, Kluwer Academic Publishers, ISBN 0-7923-7695-1, Berlin, Germany, March 2002

[7] OMG: Java™ to IDL Language Mapping Specification v1.3 (2003-09)

[8] I. Schieferdecker, B. Stepien: Automated Testing of XML/SOAP based Web Services, 13. Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), Leipzig, 26.-28. Febr. 2003, Informatik Aktuell Springer 2003.

[9] M. Karki, A. Nyberg: Initial input for using C/C++ with TTCN-3, ETSI MTS#40, Berlin, March 2005.

[10] Trancón Y Widemann, B., Lepper, M., and Wieland, J. 2003. Automatic Construction of XML-Based Tools Seen as Meta-Programming. *Automated Software Engg.* 10, 1 (Jan. 2003), 23-38.

[11] R. Monson-Haefel: Enterprise JavaBeans, 4th Edition, O'Reilly, 2004.

[12] Enterprise JavaBeans Technology. http://java.sun.com/produts/ejb

[13] Testing Technologies IST GmbH: TTworkbench (TTCN-3 tool set) http://www.testingtech.de