TTCN-3 Tools Interoperability Between Java and C/C++ Platforms

Ricardo Rezzano[1], Ariel Sabiguero[1], and César Viho[2]

[1] Instituto de Computación, Facultad de Ingeniería, Universidad de la República
J. Herrera y Reissig 565, Montevideo, Uruguay
{rrezzano,asabigue}@fing.edu.uy
http://www.fing.edu.uy/inco

[2] IRISA
Campus de Beaulieu
35042 Rennes CEDEX, France
viho@irisa.fr,
http://www.irisa.fr/armor

Abstract. The TTCN-3 language defines two different language mappings for the implementation of its interfaces. State of the art of TTCN-3 compilers and tools specialize in either of the two languages, but ignores the other. The problem faced by the industry is that only some Test cases can be implemented using a single platform language. This situation meets the language specification, but it is not the only possible interpretation of the standards. An alternative is a compiler that implements both interfaces simultaneously, allowing seamless integration of platforms. This paper describes the definition, design, and implementation of a proof-of-concept dual API compiler based on an Open/Free compiler.

Keywords: TTCN-3, Java, C/C++, Interoperability, Open Compiler, Dual API, Cross platform

1 Introduction

The Testing and Test Control Notation version 3 (TTCN-3) language [1] addresses the definition of Abstract Test Suites (ATS). Low level implementation details are removed from the abstract specification, but must be provided in order to build the Executable Test Suites (ETS). These details are integrated into the ETS through the implementation of two software interfaces: TTCN-3 Runtime Interface (TRI) [2] and TTCN-3 Control Interface (TCI) [3]. TTCN-3 tools combine the ATS with those routines in order to produce the ETS.

The TRI is used to communicate the test system with the System Under Test (SUT) and to do platform dependent tasks. The interface to communicate with the SUT is used to send and receive messages that are the SUT stimulus and reaction, these are defined in the ATS behavior. The name of this interface is

TRI SUT Adapter (TRI-SA). The TRI is also used to provide platform specific services to the TTCN-3 executable, like timer definitions and execution to system level functions. These functions are accessed through the TRI Platform Adapter (TRI-PA).

Functions related to the control of the execution, distribution of tasks, logging, and so on, are defined through the TCI interface, each one of these is provided for its correspondent libraries, they are then TCI Control Management (TCI-CM), the TCI Component Handle (TCI-CH) and the TCI Trace & Logging (TCI-TL).

By implementing all these interfaces, TTCN-3 language keeps the abstraction of the language removing from the language itself the handling of platform specific details. This leads to an abstract and platform independent language. It also allows work distribution between TTCN-3 and platform languages specialists and then each one can focus and specialize in their specific goals. Other advantage derivate of this abstraction is the reuse of ATS between different vendor tools and platforms.

Ultimately, executables have to be built and run against implementations. Doing so requires to implement certain operations using platform languages. TTCN-3 standardizes two platform language mappings and interfaces: C/C++ and Java Languages. As a consequence of this, the industry has developed disjoint efforts to create tools based on these platforms. The result is that there exists two isolated families of compilers, tools, libraries, and implementations: one for C/C++ and other for Java. For this reason different communities are unable to reuse naturally the libraries or tools developed by the other when they try to implement test-cases for the same kind of System Under Test (SUT). Also in some cases Implementations Under Test (IUT) are naturally more adequate to be adapted using specific languages platform. As an example, network protocols are naturally developed using C/C++ and Web-Services are easier and cheaper to develop using Java. Selecting a single tool forces the developers to pay the cost to cross the platform to implement the SUT or use a non adequate Platform Language for the specific IUT.

The standard itself does not impose that tools must be either Java or C based, so this *de-facto* situation is not required by the standard itself. It is indeed a technological and practical situation. Existing tools translate TTCN-3 ATS into one of the two platform languages, producing what is called the Compiled ATS (C-ATS). Afterward this C-ATS is compiled and linked with the TRI and TCI implementations, producing the ETS. This technological decision introduces the incompatibility with the "other" platform. This work aims to the experimental extension of a TTCN-3 compiler to do it capable of integrate Libraries and Tools from both platforms natively. The extension is fully standard compliant, because it do not need any change in the actual standards to work, what it need is changes to the tools to enable the platform selection. It makes the tools, that test developers need, available, despite of the platform base tool, in a more transparent and easy way.

During this work a prototype implementation of the proposal was developed. This proof-of-concept show that we can have a TTCN-3 compiler and its corresponding tools as we propose interoperable beyond its base platforms. Using that test programming tools the developers transparently could use libraries and components from different platforms, selecting those that suites their needs best, without changes in the language programming itself.

The rest of this work is organized as follows. In Chapter 2 motivations and practical aspects required to understand this work are described. Other related experiences and projects that contributed this work are described too. Chapter 3 defines the solution proposed, the main ideas to found it, how to integrate this changes to TTCN-3 Tools, and the technology choices made. After that this chapter describes the component architecture and design of the TTCN-3 compiler used for this experiment and how the Dual solution is integrated to it, Chapter 4 describes details of the prototype implementation and the experience in a concrete implementation of the DNSTester example. In Chapter 5 different lessons learned during this experiment are discussed, key features are highlighted, tasks to do, and further applications. Finally Chapter 6 concludes this work highlighting its pros and cons and suggesting future lines of work.

2 Background

There exists almost a decade of design and application experience on the third version of the TTCN language, counting also the articles on the language design [4, 5]. Our field experience strongly influenced us on the C side of the TTCN-3 standards. Contributions to the T3DevKit [6] or Go4IT Project [7], amongst others, are on C world. As research institutions we face the challenge to work on the Java side too. When we thought about this we rejected the idea of reimplement the different solutions in Java. Then we worked to integrate the solutions to the Java platform, these were some of our interoperability experiences. Working in this direction we naturally evolve to the idea of factorizing this experience in a general solution for interoperability, to avoid rework in the future. The goal was to found a place in the TTCN-3 Tools where we can add some kind of "Dualization logic" in the sense of make available implementations in both platforms to it.

One of the biggest challenges in order to address this goal was to have a compiler available to experiment with the proposed solutions, this means not only its License of use but its source code and the capability to modify it. To do it our first task was develop a reduced version of the compiler that we named μTTCN-3, this version scope was adequate to be operative and cover the more used language parts. After spend some work in the development of the μTTCN-3, specifically we worked in the parser for the language, we joint our effort to the Go4IT Package 2 team that was also working in this direction. As a result of this work we had available at the end of the 2007 the first version of the TTCN-3 open/free compiler, the A0 release of the TTCN-3 compiler named at

that moment as picoTTCN-3. The Figure 1 describes the picoTTCN-3 compiler architecture that we used for this work.

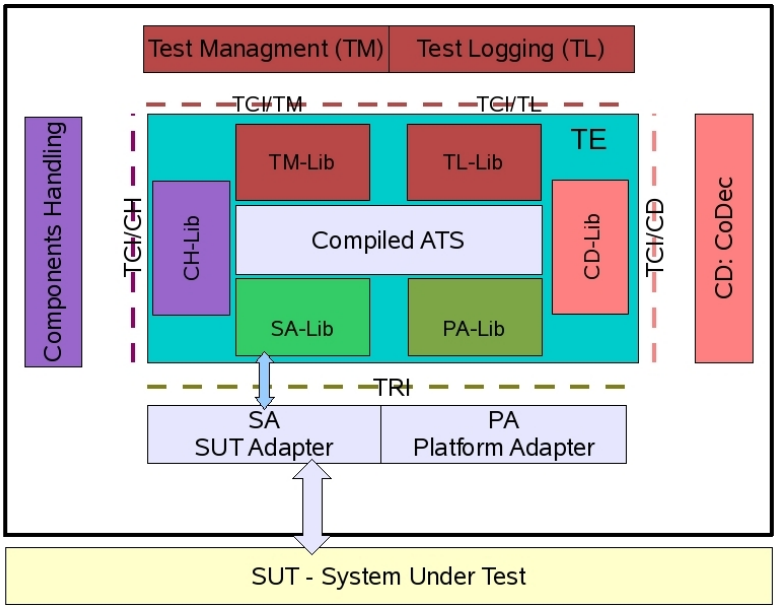

Fig. 1. Go4IT picoTTCN-3 Compiler - Initial Architecture

2.1 Dual Experiences

Similar requirements like those addressed for this work come directly from Go4IT projects, those devoted to platforms interoperability. In the context of these projects we can evaluate the effort to implement cross platform solutions or migrating solutions between platforms. There are two main experiences, one allow the use of the CoDec Generator and the IPv6 T3DevKit, ones of the C/C++ tools produced in the project, from Java compilers, the second the integration of the Java GUI Tools developed in the Go4IT Package 1 with the picoTTCN-3 Compiler developed in C/C++ in the Package 2.

We also investigate some other experience regarding interoperability in TTCN-3. Other teams faced the need of using different platforms before us. We present some of the published works as representative examples of platform interoperability requirements.

– **T3DevKit Wrapper for Java**
  This work, that took place at the InCo-UDELAR, addresses the reuse of the T3DevKit for Java based environments. It extends the T3DevKit -hopefully

in the future it will be integrated to its distribution- allowing it to generate TCI and TRI implementations usable from Java tools, specifically to the TTWorkBench tool.
- **Testing Tools Java Wrapper for Go4IT C compiler**
  This interoperability implementation was done in the context of the Go4IT project, in order to extend the use of the Testing Management Tools in Java to the picoTTCN-3 C compiler developed also in this project.
- **Modeling external activities in PERL**
  This paper [8] discusses the concepts of external behaviors, proposes extensions to TTCN-3 to realize them and demonstrates its application by a concrete example.
- **Mapping to the new platform: .net**
  The proposal [9] aims the extension of the standard to a new language mapping to the C# language. It would allow the usage of the .net platform and through it, access to other languages supported there.

2.2 JNI

Some experience was acquired in the use of Java Native Interface (JNI) [10] as an interoperability tool between Java and C/C++ platforms. We decided to investigate options for platform interoperability but finally we selected the JNI alternative. It results to be the more useful for our goals and a validated interoperability solution, between the code oriented integration tools. Some specifications was necessary for components platform and mapping, this specification shall be independent of the language and the platform components.

3 The Dual API: Solution Definition and Design

Our main goals for the solution design were generalize the platform integration and keep the solution as much transparent to the TTCN-3 users as possible. One of the main reasons of these goals was get a "pluggable" solution that can be extensible to the others Compilers and Tools in the industry.

3.1 Solution Definition

Incorporate the solution to the compiler seemed like the best option, to reach our objectives. To prototype this scenario we can use our experimental compiler and related tools, picoTTCN-3, that was developed in the Go4IT project context. That enabled us to work in the direction of integrate the solution to the language compiler itself, modifying the compiler in one of its steps or components. This alternative seems to be as the more transparent to users and developers, also as the more elegant in the design sense, living the solution concentrated in one point and easy to extend to another tools, so we decided to work in this direction. The new Dual Tool would automatically choose at runtime the SA and PA implementations of the different elements despite of their platforms. This is

what we found more interesting for our experimentation goals. In the other side, from the management and monitoring tools to the compiler, the dualization of the correspondent interfaces allow to the tools to work with compilers in any platform. For the specific goals of our experiment the implementation could have ports, timers, CoDecs, and/or external functions in different platforms, those are usually implemented in the platform languages for the developers. At the execution time the Dual Tool should select the corresponding platform to run each specific element and no special programming activity must be done for the test developers. To allow this behavior the solution only need some additional configuration available, related to the elements and its platform, this will be all developer extra work. We decided to store this configuration in a XML file that would specify the the Kind of element, e.g. Port, Timer, External Function or CoDec, the element Type Name and its platform and other related details needed. For compatibility reasons we decided to manage as default, if the element is not specified in the file configuration or if the file doesn't exists the elements, it will be considered in the tool based platform and managed as the tool normally would it. For example in the picoTTCN-3 compiler for default should be C/C++ ports implemented by the T3DevKit.

Integrating the solution to the TTCN-3 compiler Our first approach was introduce the Dual API solution in the TTCN-3 compiler translation step. Some considerations guide us to change this approach, between these the most important where the solution flexibility, dynamism and usability. If we implement the solution in the translation step, the developers, should recompile the ETS each time that some change platform configuration be needed, that is normally time expensive in the different TTCN-3 Tools. As an alternative we choose to place the Dual API solution in the TTCN-3 Run Time System (T3RTS). Then when the configuration change, dynamically the system can get the changes to its next execution or initialization. At the runtime the TE will make calls to different components through TTCN-3 Interfaces, at this moment it should know the base platform for this specific components and then call them through the correct interface implementation, they will be Java or C/C++. The initialization of this configuration should be done for each execution at the top level of the T3RTS, depending of the Tool design the Test System should be reseted at different levels.

Extending Language Abstraction to the implementations between platforms With this design we go one step forward in the abstraction for TTCN-3 implementations. When other platform should be used for an implementation element, the developer only will need to add the corresponding platform component and configuration and don't do any other task regarding that. The TTCN-3 developer should not pay attention in how to integrate the solutions for the other platform at technical low levels, and then the development of the corresponding adapters can be delegated completely to the corresponding team platform specialists.

Other goal was be compliant with the TTCN-3 standard, to do this we had to keep the TTCN-3 interfaces without changes, then the idea was to strictly follow the interfaces specifications. In order to achieve this objective we try to isolate the major part of the Dual API implementation in a specific module devoted to it and implement the interface with both platforms separately and complaint with the standard specifications. Only the minimal necessary information, related with TTCN-3 objects, was added to the T3RTS, that information represent the object platform configuration and its allow to make the decisions of use the corresponding platform.

3.2 Technology Selection

We select the JNI technology [10] as the most adequate to the code integration between Java and C/C++. As we discussed earlier the experience demonstrate that using JNI for interoperability between C and Java is one of the most effectives and tested approaches. Many solutions were developed using this technology, for different tools vendors, based in both platforms, to integrate work from different platforms. Other technologies were evaluated to implement the solution but most of them were rejected because they are oriented to services or components but not to code integration. Others oriented to code are partial and do not fill the goal of generality that we pursue.

For most of the experiences the idea behind the use of JNI was build some kind of wrapper that allow the execution of cross code, the wrapper act as an intermediary between both platforms that know how to talk with each one of them. Basically what JNI do is execute Java Code from C applications or components and vice versa execute C Code from Java platform applications or components. JNI allow to implement C/C++ methods from Java classes and Java methods from C/C++ modules or classes, trough the use of a Java Virtual Machine (JVM) environment. The JNI connector translates Java objects instances to C/C++ structured types and data-types in order to call the C/C++ functions, these are used as functions parameters and return values. In the reverse sense JNI convert C/C++ data-types to instances of Java objects, that are used to call Java native methods. Using JNI the JVM management is powerful and can be made trough a simple and flexible interface. JNI is considered the best tool to interoperate between C/C++ and Java and it is useful in our case for all the necessary tasks in the Dual API development.

3.3 Integration Architecture

The main component of the Dual API is an independent module to implement the cross platform execution, in our prototype case Java, this module resides yet in the C executable and it is linked with the other compiler parts at the link-edition time of the ETS, this module implement the call to the second platform using JNI technology. Also some very specific and small modifications were done to the T3RTS components, specifically in the libraries used to implement the TTCN-3 interfaces, these are SA, PA, CD, CH, TM, and TL Libs. These allow

the T3RTS to retrieve the platform elements information and to call the execution of code in the base platform languages that implement each one of them. This information will be used specifically, at the moment of calling TTCN-3 components and libraries that implement some interface methods.

Dual module The Dual module was designed to do all the independent activities needed for the implementation of the dual solution. This Dual module is responsible for storage in memory of the Language Platform Elements Configuration, after reading them from the defined configuration file. In the Dual module the internal RTS interfaces to call the cross platform interfaces were implemented and also the call to the cross platform interfaces using the standard interface definition can be found. The use of JNI is concentrated in this module, this imply a centralized JVM management and another activities related with JNI initialization and configuration.

Changes introduced to libraries The T3RTS module of the Go4IT compiler was designed to separate the RTS for the rest of the solution. And the T3RTS itself has isolated its components significantly, those facilitate us to be very specific where add the modifications for dualization. This definitions was preformed as specific as possible looking to add or modify the minimal amount of code as possible. In the Figure 2 there are described the changes to the picoTTCN-3 compiler modules based in its original architecture diagram.

4 Prototype Implementation

As was mentioned early in this paper we decided to concentrate the "dualization" logic in a separately module, its was named T3RTSDual. The T3RTSDual is responsible for the environment initialization and load platform configuration for the IUT. This procedure should be executed each time that the RTS is initialized, the moment for this initialization is tool dependent. For the picoTTCN-3 case we locate it in the T3RTSComponentType initialization. Some new configurations were needed at this level, for this implementation these were classpaths for JVM, JNI and Java implementation classes.

After the initialization some references will be available to the T3RTS, the most important are those for the environment and objects implemented in the cross platform.

The other relevant information to keep available is the platform of each object, this information is loaded and maintained also for this module. This information is retrieved at the initialization moment from the platform configuration file. At any moment that the T3RTS must get the platform of one port, timer, CoDec or external function there is a function that return it. At the moment that one of this elements is created in the T3RTS, one function will be used to set the element platform information.

In the other side each time that one interface will be called from the T3RTS some logic was added to manage the capability to execute in different platforms.

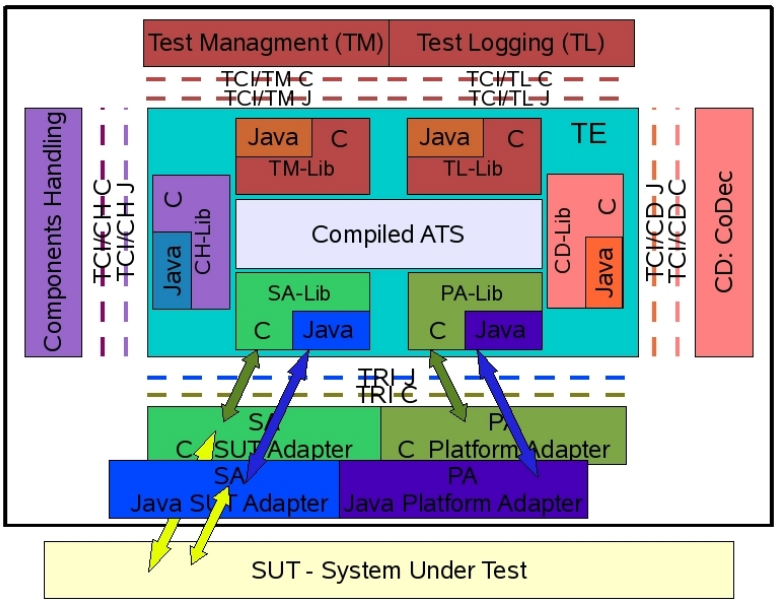

Fig. 2. Dual API - Design Integration Architecture for Go4IT Dual

For the example of the port objects case this logic was introduced to the following activities `Map, Unmap, Send (Enqueue), and Receive`.

One difficulty addressed at this point was that the interface names should be the same for both platforms, but for technical reasons the same name can not be used at link-edition time, else we have a name collision. The JNI technology help us to workaround this problem, we implemented the DualAPI calls for the second platform with the standard name ended with the platform, as a RTS internal call only, and delay the call to the correct interface name to the other platform call inside the JNI environment where there is not exist that collision problem. Other issue was the visibility of the Java environment in the T3RTS module. After several attempts to solve this issue by making references to the JVM environment in an global T3RTS class, and that in some parts of the TE do not succeed correct visibility of it, we chose to use the `T3RTSDual` module to load and give visibility to this variable, giving pending the final solution to this issue for next releases. Some general dualization activities were introduced at the `T3RTSCHRequired`. In the invocations to `TciExecuteTestCase` and `TciReset`, where are called the corresponding TRI functions, is needed to take in account the platforms of the Test Case objects in order to decide with which platforms this Test Case will work. To give the correct information to do that we should take into account the element platform configuration for all elements inside the Test Case and/or component. We need to add some logic to decide when an IUT

will work with which platforms and also to prepare the correct parameters to call those functions.

4.1 JNI Implementation details

Environment Management For the case of Java native implementations, tools and compiler in Java that call C/C++ functions, the JNI libraries use the current Java environment as its base environment. For the other case when Java functions should be invoked from C/C++ tools our JNI support libraries try to get the Java environment from a global reference and if this do not exists yet it is created. The main problems addressed at this point are JVM creation and class paths management for implementation of the function and components to use. Some alternatives was managed, environments variables and default values, to facilitate the use of the solution. Thread management, for some cases as the port listeners, advanced thread management was needed, for these cases thread "attach" attached was implemented.

Functions Implementation For this task the JNI library implement the correspondence between objects and data types, this is a generic implementation that, based in package and interface names create dynamically the necessary objects. In one sense to invoke from C functions interfaces implemented in Java methods, the caller module should know the classes that implement the interfaces. In the other sense to invoke from Java functions interfaces implemented with C methods, the caller Java module need to load a dynamic library with this interfaces implemented. This information was stored and retrieved from environment variables.

4.2 Library changes

As we establish in the section before each time that one interface will be called from the T3RTS some logic was added to the correspondent libraries to manage the situation, the details of changes done for this prototype is:

- Map and Unmap
- Send, Enqueue
- Receive, Enqueue
- Set the platform for each element

4.3 Integration with the Go4IT compiler

The Go4IT project is C/C++ platform and it use to build the solutions the Autotools package, these are a group of GNU utils to help in this activities. We build some scripts to adjust the actual Go4IT compiler implementation and keep for the moment the dual compilation as an optional feature of the solution. Avoiding in this way an unnecessary overload until this solution was finish and tested.

4.4 The SUT Experiment - Java DNSTester

We decided to reuse the DNSTester as our Test Case experiment, to test the Dual Compiler. The idea was keep the TTCN-3 code unchanged, implement the SUT for the DNSTester in Java and add the necessary configuration to indicate to the compiler the SUT platform. We proposed to test the solution for both platforms, to do that we should execute the compiled Test Case and check whether the DNSTester execute trough CodeGen or trough Java implementation.

The figure 3 show the diagram with the solution distributed between he platforms.

The experiment work in the desired way, when the configuration is absent for the DNSPort the Test Case work as normally calling the CoDec Generator interfaces, when the correspondent Java configuration was added for this port the RTE call the Java Port implemented for this test, to do this first call the internal cross platform functions and then from the Java environment call the functions of the standard interfaces.

Bellow there are some implementation details, first a diagram representing the DNSTester DUAL, second examples of the source codes for different platforms, third the XML configuration file and finally the trace results for each case.

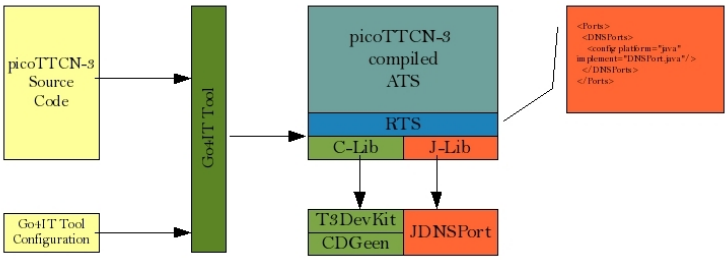

Fig. 3. Experiment - DNSTestr DUAL

4.5 DNSTester Source Code Map - Java

```
public TriStatus triMap(TriPortId compPortId, TriPortId tsiPortId) {
    System.out.println("***** triMap *****");
    System.out.println("    CompPort: " + compPortId);
    System.out.println("    TsiPort: " + tsiPortId);
            this.compPortId = compPortId;
            this.tsiPortId = tsiPortId;
            this.nameserver = "200.40.30.245";
```

```
            dNSServerAddress = InetAddress.getByName(nameserver);
            if (connected) {
                return new TriStatusImpl("Already Connected Port"); // automatically se
            }
            this.triCommunicationTE = new TriCommunicationTEImpl();
            this.start();
        return new TriStatusImpl(TriStatus.TRI_OK);
    }
    public TriStatus triRaise(TriComponentId componentId, TriPortId tsiPortId,
            TriAddress sutAddress, TriSignatureId signatureId,
            TriException exception) {
        System.out.println("***** triRaise *****");
        return new TriStatusImpl(TriStatus.TRI_ERROR);
    }
```

4.6 DNSTester Source Code Map - C

```
    public TriStatus triMap(TriPortId compPortId, TriPortId tsiPortId) {
        System.out.println("***** triMap *****");
        System.out.println("    CompPort: " + compPortId);
        System.out.println("    TsiPort: " + tsiPortId);
            this.compPortId = compPortId;
            this.tsiPortId = tsiPortId;
            this.nameserver = "200.40.30.245";
            dNSServerAddress = InetAddress.getByName(nameserver);
            if (connected) {
                return new TriStatusImpl("Already Connected Port"); // un poco oscuro, p
            }
            this.triCommunicationTE = new TriCommunicationTEImpl();
            this.start();
        return new TriStatusImpl(TriStatus.TRI_OK);
    }
    public TriStatus triRaise(TriComponentId componentId, TriPortId tsiPortId,
            TriAddress sutAddress, TriSignatureId signatureId,
            TriException exception) {
        System.out.println("***** triRaise *****");
        return new TriStatusImpl(TriStatus.TRI_ERROR);
    }
```

4.7 DNSTester XML configuration for Java port

```
<Ports>
  <DNSPorts>
    <config platform="java" implement="DNSPort.java"/>
  </DNSPorts>
</Ports>
```

5 Summary and Remarks

5.1 Lessons learned

During the first steps we can see that the effort to use cross platform adapters, libraries or tools was too big and that don't stimulate the solutions reuse or sharing.

5.2 Evaluate this solution design and discuss some alternatives.

The option done for locate the solution at the run time system against to include it in the translation step give much more flexibility to the solution and keep the solution smaller and easy to implement in another compilers or tools. Also this design left the solution more transparently for the teams involved in the design and development of SUTs than the other alternatives.

5.3 To do

Some tasks need to be done to finalize the picoTTCN-3 native interoperability implementation. In order to provide a full DUAL API TRI and TCI implementation we need complete the modifications in all the interfaces

- SA Library (done)
- PA Library (done partially)
- TM Library (to do)
- CD Library (to do)

Another task to do is the implementation of the Dual module as a c/c++ class, this task is partial accomplished and should be finished.

5.4 Future Applications

The idea behind this solution, is to provide simultaneously dual access to both standard platform component implementation that, can be reused for others compilers and tools, implementing natively the interoperability. To allow Java Compilers to use C code implementations and libraries by providing dual interface implementations and, inn the other hand to allow C Compilers to use Java code implementations and libraries by providing dual interface implementations. From the test and management tools perspective to allow them to manage compilers in the other platform.

Same idea can be used to extend the implementations to others platforms languages without the needed of implement an entire tool in this language and avoiding modifications to the standard, e.g. C#.

6 Conclusions

After the Dual API prototype construction and our experimentation with the DNSTester example we can confirm that this new interpretation of the standard allows the reuse of tools and libraries between different platforms with a minimal effort at the implementation time. This seems to be the most transparent solution from the platform interoperability point of view and also for the different teams involved in testing those that create Abstract Test Specifications and those that generate Executable Test Suites. Suppliers that implement this solution could easily reuse cross platform tools and libraries. Their solutions also can be used from another suppliers or open TTCN-3 Tools. In general, if this proposal is implemented widely the TTCN-3 user community will be able to take advantage of the better services and solutions provided by each platform, instead of having to choose beforehand the platform for a project as a whole.

6.1 Pros

As a result of this experiment we can conclude that automatic interoperability between Tools and APIs in different Language Platforms in TTCN-3 is already reached in specific prototype implementation. We validated the methodology approach of tools that are open to work with both platforms implementations and libraries. This gives the real possibility to choose the best base platform language to implement TTCN-3 adapters to each implementation, independently of the tools base platform language. With this approach we empower the capability of reuse existing libraries and tools for different compilers. Also revalorize the work already done in the different fields where TTCN-3 was used to develop Test Cases, because it can be reused in the other platform. Clearly, this fact increases TTCN-3 usability and promotion of tools and library development for the language in different fields. The new generated situation promote the joint efforts between different Tools suppliers and language platform specialists.

6.2 Cons

Dual Implementations are more complex. Developers should take into account different components from different platforms and should understand the relationship between each part of the solution and the correspondent platform. The Test Architect should master TTCN-3, C, and Java, they should decide in each case which is the best platform to use. Based on the SUT nature they should evaluate the pros and cons for use the cross platform functionality or make the implementation using native tools platform.

Acknowledgment

The authors would like to thank Hernán Martínez for his support, experience and invaluable help working with JNI for different tasks in the context of Go4IT project.

References

1. ETSI. ES 201 873-1 Part 1: TTCN-3 Core Language, Version: 3.2.1. http://www.ttcn3.org/StandardSuite.htm, 2007. [Online; accesed 9-April-2008].
2. ETSI. ES 201 873-5 Part 5: TTCN-3 Runtime Interface (TRI), Version: 3.2.1. http://www.ttcn3.org/StandardSuite.htm, 2007. [Online; accesed 9-April-2008].
3. ETSI. ES 201 873-5 Part 5: TTCN-3 Runtime Interface (TRI), Version: 3.2.1. http://www.ttcn3.org/StandardSuite.htm, 2007. [Online; accesed 9-April-2008].
4. Jens Grabowski and Dieter Hogrefe. Towards the third edition of ttcn. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *(TestCom 1999) Testing of Communicating Systems, Methods and Applications, ISBN 0-7923-8581-0*, pages 19–30. Kluwer Academic Publishers, 1999.
5. Jens Grabowski, Anthony Wiles, Colin Willcock, and Dieter Hogrefe. On the design of the new testing language ttcn-3. In Hasan Ural, Robert L. Probert, and Gregor v. Bochmann, editors, *(TestCom 2000) Testing of Communicating Systems, Tools and Techniques, ISBN 0-7923-7921-7*, pages 161–176. Kluwer Academic Publishers, 2000.
6. T3DevKit. http://t3devkit.gforge.inria.fr/, 2007. [Online; accesed 22-April-2007].
7. Go4IT Consortium. Go4IT project - Advanced tools and services for IPv6 testing. http://www.go4-it.eu, 2008.
8. Theofanis Vassiliou-Gioles , George Din and Ina Schieferdecker. Execution of External Applications using TTCN-3. http://www.springerlink.com/content/5cp0b7qpa5j6ut5a/, 2004.
9. U. Grude and F. Schröer and P. Enskonatus. TTCN-3 for .NET. TTCN-3 User Conference 2006 - 31 May to 2 June, Berlin, Germany (http://www.ttcn-3.org/TTCN3UC2006/FinalPresentations/P6.1_grude-presentation.pdf), 2006.
10. SUN. Java Native Interface. http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html, 2006.