TTCN-3 SIP/SDP Codec – Design document

1 Introduction

The purpose of this document is to present and describe the different problems, solutions and design choices encountered while developing SIP/SDP codec within STF370.

The purpose of this document is to present and describe issues and design choices made while developing a TTCN-3 SIP/SDP codec in the context of the development of an IMS interoperability test system within STF370 .
For further information the reader is referred to [IMS arch] for an overall view of the IMS interoperability test architecture which has served as the main source for design requirements.

This document has been written with the assumption that the reader is well versed in C++ and TTCN-3 [core] programming. Also good knowledge of the operation of TCI [TCI] standard is assumed.

2 Design Objective

The main purpose of the TTCN-3 SIP/SDP codec is to implement codec entity [core, TCI], e.g., in a TTCN-3 IMS interoperability test system described in [IMS arch], i.e., transferring SIP and SDP messages from their abstract syntax into transfer syntax and vice versa. Since the codec essentially only depends on TTCN-3 SIP and SDP types it is reusable in any test system that uses these types, i.e., the TTCN-3 LibSip library.

3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

DNS	Domain Name System (protocol)
IP	Internet Protocol
IMS	IP Multimedia Subsystem
ISDN	Integrated Service Digital Network
ISUP	ISDN User Part
OS	Operating System
RFC	(IETF) Request For Comments
SDP	Session Description Protocol
SIP	Session Initiation Protocol
TCI	TTCN-3 Control Interface
TTCN-3	Testing and Test Control Notation 3
TE	TTCN-3 Executable (as defined in [TRI] and [TCI]

4 Design Requirements

4.1 RFCs to be covered

The codec should be able to encode and decode all SIP messages supported by the TTCN-3 SIP library type structure. Thus it should cover the message formats described in the following RFCs:

• RFC 3261	SIP: Session Initiation Protocol
• RFC 3262	Reliability of Provisional Responses in the Session Initiation Protocol (SIP)
• RFC 3265	Session Initiation Protocol (SIP)-Specific Event Notification
• RFC 3313	Private Session Initiation Protocol (SIP) Extensions for Media Authorization
• RFC 3323	A Privacy Mechanism for the Session Initiation Protocol (SIP)
• RFC 3325	Private Extensions to the Session Initiation Protocol (SIP) for Asserted Identity within Trusted Networks
• RFC 3326	The Reason Header Field for the Session Initiation Protocol (SIP)
• RFC 3327	Session Initiation Protocol (SIP) Extension Header Field for Registering Non-Adjacent Contacts
• RFC 3329	Security Mechanism Agreement for the Session Initiation Protocol (SIP)
• RFC 3455	Private Header (P-Header) Extensions to the Session Initiation Protocol (SIP) for the 3rd-Generation Partnership Project (3GPP)
• RFC 3515	The Session Initiation Protocol (SIP) Refer Method
• RFC 3608	Session Initiation Protocol (SIP) Extension Header Field for Service Route Discovery During Registration
• RFC 3841	Caller Preferences for the Session Initiation Protocol (SIP)
• RFC 3891	The Session Initiation Protocol (SIP) "Replaces" Header
• RFC 3892	The Session Initiation Protocol (SIP) Referred-By Mechanism
• RFC 4028	Session Timers in the Session Initiation Protocol (SIP)
• RFC 4244	An Extension to the Session Initiation Protocol (SIP) for Request History Information

- RFC 5009 Private Header (P-Header) Extension to the Session Initiation Protocol (SIP) for Authorization of Early Media

Some SIP message constructs reuse some headers defined in the HTTP protocol. Thus the following RFCs should be partially supported:

- RFC 2616 Hypertext Transfer Protocol -- HTTP/1.1
- RFC 2617 HTTP Authentication: Basic and Digest Access Authentication

5 Further design considerations

The following design considerations have been captured based on an analysis of the various protocols that need to be supported by the codec and the type definitions provided by TTCN-3 libraries.

5.1 General

The codec should be extensible at low cost. SIP is still evolving and new headers and messages may be added to the protocol. Therefore it is essential to be able to add new types in the codec without having to fully redesign it.

The codec should be as generic as possible and the chosen solution should not be operating system or hardware specific. Ideally, the codec would be useable on both Windows and UNIX-like operating systems.

In order to ease test system debugging and logs analysis, the codec should provide a mean to display raw messages, i.e. protocol messages exactly as they are received. This goal will be achieved by using the “payload” field in message templates.

Concerning the different kinds of SIP payloads that will be handled by the codec, it is important to note that the only supported payload will be SDP. All other payload types (among them, XML) will be processed as character strings.

UTF-8 is the standard encoding type for SIP messages. However, SIP test suite has been developed using `charstring` instead of `universal charstring`. The codec should somehow deal with this issue. One possible solution would be to replace unicode characters by displayable characters and raise a warning.

5.2 Encoder

The TTCN-3 types in the SIP library provide an abstract representation of the SIP messages. Only the semantic information is represented, and all syntactic elements which do not carry any useful information are not represented. Among them:

- linear whitespace and newlines between the header fields
- delimiters , e.g`. : ; , ? & = @` a as well as `"` for quoted string.

It is the responsibility of the encoder to add these structural characters when necessary, in order to build valid SIP messages. On the contrary, these elements will have to be removed by the decoder before assigning TTCN-3 values.

5.3 Decoder

Extra white space is semantically meaningless and can appear in many places inside a SIP message. The codec will have to be able to deal correctly with these white spaces and ignore them when necessary.

Additionally, SIP specification allows some header fields to appear multiple times, but with different encoding, i.e., one header field per header, all header fields in same header, or any variation of the prior. The codec will have to carefully process these headers and regroup them in a single TTCN-3 list value when decoding SIP messages, as the TTCN-3 typing is designed to handle multiple header fields in a list of header field values. The sequence in which these header fields appear must absolutely be preserved as it has a semantic importance.

SIP specification also gives the opportunity to use short names for some header identifiers. The codec will have to deal with these synonyms, which is not really complex but needs to be taken into account especially when dealing with the previous point.

Current TTCN-3 type system makes use of “record of” and “set of” structure. Unfortunately there are two ways of representing an empty optional “set of” or “record of”: either the “record of” is omitted or it is present but contains zero elements. These two possibilities for representing the same data can have a real impact on matching process and therefore requires a design decision in decoding.

The decoder shall support the test case writer and ensure or fill in correct values for the SIP Content-Length header.

Last but not least, it is necessary to find a way to handle an unsuccessful decoding of messages. The codec should produce a short report for each failed attempt showing the raw message as well as a short description of the error and its location in the message.

6 Software Architecture

6.1 Rejected solutions

6.1.1 External SIP application

One possible solution could be to use an external SIP application to decode and encode SIP messages. For this approach it is necessary to develop a non-standardized interface with this application. In addition, it would make the codec and thus the complete test environment dependent on that external application. This situation is not desired especially when considering the questions of maintenance and extensibility. In addition, the codec would here be vulnerable to possible errors present in the external application, which would completely reduce the confidence one can have towards this test suite.

Another issue raised by this solution is the difficulty to debug the encoding/decoding of the messages. The codec provided by the existing implementations of the protocol (server, client, …) are designed to accept the message or drop it if it is not conformant, but they are not generally designed to help in debugging the messages received by another implementation (like giving a detailed account of the location of the error and its description).

6.1.2 Wireshark

A second alternative would be to use a network protocol analyzer such as Wireshark and use its dissector library to decode SIP messages. This approach brings sensibly the same problems as the previous one:

- Possible bugs in external tool
- Extensibility and maintenance dependent on external application

In addition, wireshark can only decode the messages. An encoder has to be implemented separately which could be considered to render the overall solution more complex to maintain.

6.1.3 Yacc parser

Another approach would be to use a parser generator such as Yacc. A parser typically receives a sequence of tokens from a lexer and tries to derive the unique sequence of grammar rules that can produce that token sequence. It would be then possible to create a parser in order to decode SIP messages.

However, parsers usually require the protocol grammar to be in a particular form (typically, Yacc requires LR(1) grammar). Unfortunately SIP grammar is not of type LR(1). In addition, even if it is LR(1), there is absolutely no guaranty that future adjunctions will not modify this property of the grammar

In conclusion, this solution cannot be selected, as it would require rewriting SIP grammar to make it compatible with our needs, and to repeat this operation every time the protocol is extended, which does not match requirements concerning maintainability of the codec.

6.2 Selected solution

6.2.1 T3devkit

6.2.1.1 Overview

T3DevKit is a project developed by IRISA which aims to provide tools and libraries to help developing TTCN-3 codecs and adapters. It is composed of two main elements: T3DevLib, a library of C++ classes that handle generic codec & adapter functionalities, and T3CDGen, a tool that can generate a codec implementation template from TTCN-3 source files. Protocol specific adaptations and codec behaviour are then customised by adding pieces of code called “codets”. The global integration of T3DevKit in a TTCN-3 project is shown in Figure 1.

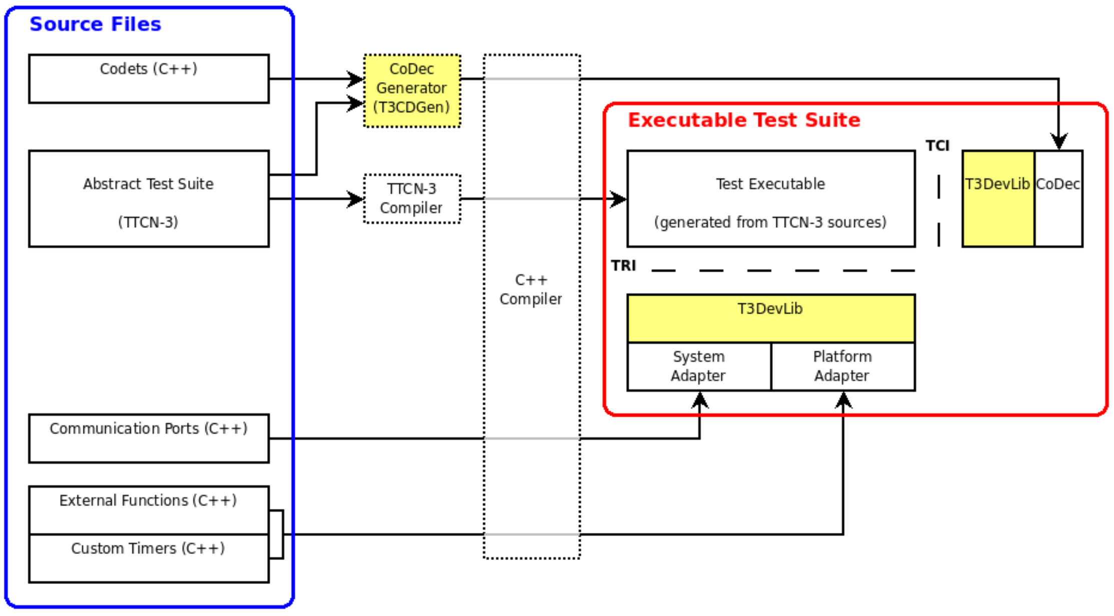


Figure 1 T3DevKit integration (©IRISA http://t3devkit.gforge.inria.fr/doc/userref/)

6.2.1.2 Advantages and drawbacks

T3DevKit provides several advantages compared to the previously described solutions. First of all, T3DevKit features a codec generator (T3CDGen) which handles the creation of codec structure and removes the need to manually implement lots of generic functions.

In addition, T3DevKit is tool-independent and does not rely on external SIP (or other protocol) application. This is a very important point, especially for the maintenance of the codec, as being dependent on an external application or library means maintaining the interface between the codec and these external elements.

Concerning the maintenance of the codec, only the specific processing concerning new types or modified types has to be added or updated. Everything else remains unchanged and there is no need to revalidate old codec sections (except the ones which interact with the new code).

T3DevKit also provides a stable interface to codet developers, and in case of changes in the TCI API, no code modification are required in the manually generated code to update the codec: the TCI related functions are all located in T3DevLib.

On the drawback side, following are the known issues extracted from the user documentation:

- The library does not support the following types:
    - float
    - universal charstring
    - address
    - object id
    - anytype
- TTCN-3 primitive subtypes (integers, octetstrings...) are not supported by the CoDec generator, they need to be declared manually in a .h file provided by the user.
- Type and field attributes (except the optional keyword) are ignored by the CoDec generator.
- Remote procedure calls as well as broadcast and multicast communications are not yet supported.
- Although ready to be handled internally, modules names are not fully supported. Name clashes between identifiers from different modules should be avoided when writing Abstract Test Suites.
- Nested type definitions are not supported.
- Imports statements are ignored by the generator, all the TTCN-3 type definitions are imported by default.
- Separation of TRI and TCI specific code is hard to achieve
- TTCN-3 parser of T3CDgen has limited support of latest TTCN-3 language features (should only be used with modules containing TTCN-3 types)

These drawbacks, which are mainly due to the fact that the project is still under development and are will be addressed in the future, do not have any real impact as far as SIP codec is concerned.

6.2.1.3 Design strategy

T3DevKit is designed to generate automatically a codec from the type definitions from the TTCN-3 source files. The default behaviour is to encode and decode all the fields of the message successively and concatenate the result. This default behaviour generally needs to be customised so as to:

- handle syntactic elements that are not represented in the TTCN-3 types (e.g. white space and delimiters,…)
- identify the position of each field when decoding a raw binary message

The customisation is possible by implemented additional functions named “codets” that will be integrated in the final codec and that will be called by the generated codec. For instance “SDP_Message::Predecode()” is a function called just before the codec decodes a raw message of type SDP_Message. A codet is implemented as a C++ member function. Any kind of processing can be done inside the body of the function and it is possible to interact with the generated codec with the well-defined API of T3Devlib. It is then possible to read or write data in the binary buffer and make some prediction about the length or the presence of a field.

6.2.1.4 Encoder

The encoder mainly appends the syntactic element that do not appear explicitly in the TTCN-3 message (but that can be derived from the structure of the message). This is done by writing the delimiter in question in the buffer before and/or after the adequate field.

Encoding of the “Conent-Length” field of the SIP messages requires more processing. The length of the message body is not known in advance, the value provided by the TTCN-3 code may be incorrect or not be available. Instead it is possible to fill it with blank spaces and remembering the position of this value in the buffer. Then in the PreEncode() and PostEncode() codets of the message body, the current position in the buffer is saved and then used to compute and encode the value of the Content-Length field.

6.2.1.5 Decoder

Implementing the decoder mostly consists of:

- validating the format of the message
- skipping the syntactic elements that are not represented in the TTCN-3 structure
- make predictions for the codec about the length of each fields (so that when the generated codec decodes a variable-length field (typically a charstring) it does not reads all the bytes until the end of the buffer, but stops at the real end of the buffer.

Since the format of the SIP and SDP message is mostly described by a BNF representation, it was decided to validate the input message and identify each field using regular expressions.

A utility class named “Regex” was developed. It is implemented based on the portable C++ BOOST regex library and implements perl regular expressions. Moreover it is integrated partially with the T3DevKit API by providing functions to match the regex on a part of the encoding buffer, to report easily the length of a field, to move the position in the buffer at the beginning or the end of a matched group in the regex and to report detailed error messages in case of mismatch.

7 Validation procedure

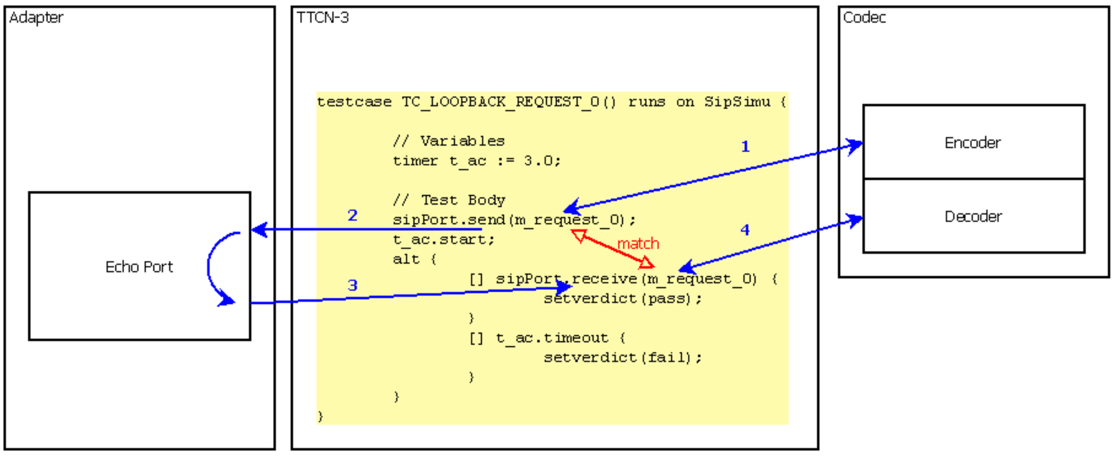
In order to validate the correctness of the TTCN-3 codec implementation and ensure its reliability, two complementary approaches have been used.

7.1 Loopback tests

The idea of this approach is to verify that a TTCN-3 representation of a message can be encoded and then decoded correctly by the codec. Additionally, the initial and final messages are check to be identical. By doing this, it is possible to ensure that both the encoder and the decoder work in the same manner. Moreover, a large variety of messages are generated and tested, so that they cover the complete type system.

This method should quickly detect errors located in one part or the other of the codec. However, if the same error is present both in coding and decoding functions, then it might stay undetected. In addition it is important to note that it is not possible to test all messages combination; therefore these tests will not permit to ensure 100% reliability of the codec.

7.1.1 Test System Architecture



```
testcase TC_LOOPBACK_REQUEST_0() runs on SipSimu {

        // Variables
        timer t_ac := 3.0;

        // Test Body
        sipPort.send(m_request_0);
        t_ac.start;
        alt {
                [] sipPort.receive(m_request_0) {
                        setverdict(pass);
                }
                [] t_ac.timeout {
                        setverdict(fail);
                }
        }
}
```

Figure 2 Loopback architecture

In loop back tests all test cases share the same structure. In fact the only difference in each testcase is the top-level template used for the “send” and “receive” operations.

```
testcase TC_LOOPBACK_REQUEST_0() runs on SipSimu {

      // Variables
      timer t_ac := 3.0;

      // Test Body
      sipPort.send(m_request_0);
```

```
        t_ac.start;
        alt {
              [] sipPort.receive(m_request_0) {
                    setverdict(pass);
              }
              [] t_ac.timeout {
                    setverdict(fail);
              }
        }
}
```

7.1.2 Template generation

TTCN-3 message templates are generated automatically by a script based on a TTCN-3 type module. The main difficulty is to generate a reasonable amount of templates: it is very easy to generate millions of template due to combinatory explosion. To avoid this problem, it is important to carefully define the rules that will be used for template generation. Ideally, these rules lead to the generation of a minimum number of templates by keeping only the most interesting ones, from a testing point of view.

7.1.2.1 Generation rules

The following generic rules have been selected for this project: Port primitives are the starting point of the process. Complete templates are generated for each primitive, using a recursive strategy and by applying the following rules:

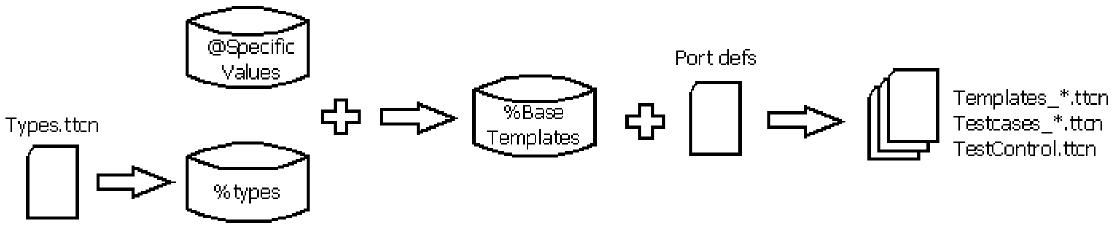
- While processing a `record`, generate two templates, one containing values for mandatory fields only, and one containing values for mandatory fields and optional fields. Additionally, generate a parameterised number of templates containing values for mandatory fields and randomly selected optional fields. By doing this, extreme case and some intermediate case templates are generated. Note: with this method, each field is tested at least once.
- A `set` is processed like a `record`.
- While processing a `union`, generate one template per alternative.
- While processing a `record of`, generate different templates with different list sizes. The number of templates and the sizes of the list should be parameters.
- A `set of` is processed like a `record of`.
- While processing an `integer` field, generate different templates with different values (minimum value, maximum value, and optionally random values).
- While processing a string field (`octetstring`, `bitstring`, `charstring` ...), generate different templates with different values (short string, long string, and optionally random length strings).
- While processing a `boolean` field, generate two templates (`true` and `false`).
- While processing `enumerated` field, generate all possible templates, based value constraints.

- If a complex type as already been derived once in any of the already generated templates, it should not be derived again (it is very likely that the same codec function will be used), and should trigger the generation of only one template.

In addition to these rules, the template generator should offer the possibility to use specific values for a particular field. This way, it is possible to test more intensively some important fields.

7.1.2.2 Template generator design

The generator developed within STF370 is a Perl script which receives type and port definitions as input and produces three files: one containing all the generated message templates and sub-templates, a second file containing the generic test cases and a test control file.



Before using the generator, some values need to be adjusted. First of all, the generic testcase pattern can be modified. It is stored in the global variable `$testcaseTemplate`. Default values are stored in the global hashmap `%defaultValues` and initialized through the function `initializeDefaultValues()`. This hashmap stores two kinds of values:

- default values used for basic types (`charstring`, `integer`, …);
- values for specific fields in a complex type (typically `record`). In this case, the syntax for hash entries is `Type<space>Fieldname`.

The following table shows some example entries:

Hash entry	**Hash value**	**Comment**
`Charstring`	`['"a"', '"abcde"', '"abcdefghij"']`	Three possible values for `charstrings`
`Integer`	`[1, 2, 3]`	Three possible values for `integers`
`Boolean`	`["true", "false"]`	Boolean can be `true` or `false`
`StatusLine sipVersion`	`['"SIP/2.0"']`	Field `sipVersion` in record `StatusLine` will be filled with value “`SIP/2.0`”
`DeltaSec`	`['"1"', '"123456"', '"123456789"']`	Three possible values for `DeltaSec` fields. In this case `Deltasec` is an alias of `charstring`. This entry will override default `charstring`

		values.
`SDP_contact addr_or_phone`	`['"test_email@etsi.org"', '"+33 4 - 9294 4200"']`	Field `addr_or_phone` in `SDP_contact` record will be filled with these two specific values.

Figure 3 Examples of default and specific values

The generator is typically used as follow. The output files are created in the current directory.

```
$ cat compiledTypes.txt | SipCodecTestGenerator.pl
```

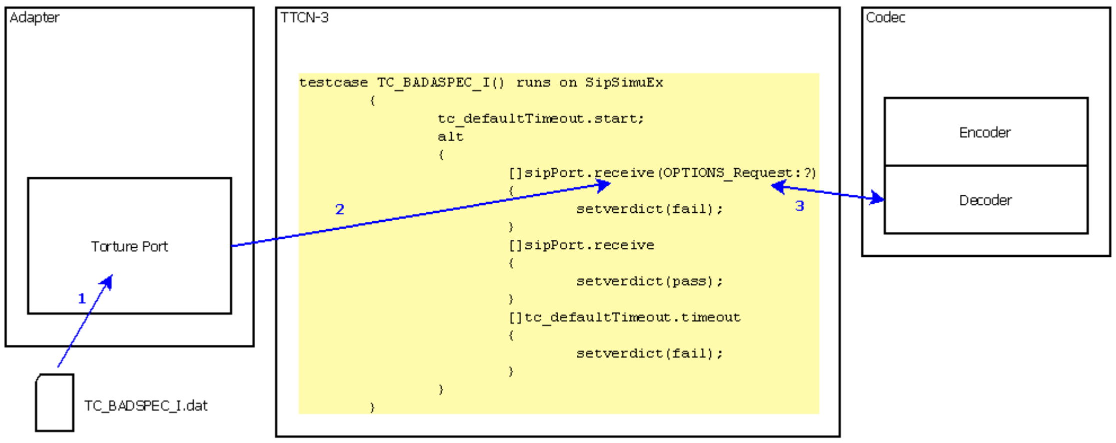
Among its known limitations, this Perl script does not feature a complete TTCN-3 parser; only types and port definitions can be analyzed and any other TTCN-3 instruction causes a failure. It is also important to notice that so far, the Perl script does not support comments either.

7.2 Torture tests

In the case of an encoder input data is controlled and cannot go beyond the TTCN-3 typing possibilities. The decoder however can receive a huge variety of encoded messages. Therefore it is essential to ensure that the decoder will be robust enough to deal with these messages and to decode them. To achieve this goal, this second validation approach makes use of torture messages defined in RFC 4475. Each of these messages presents some characteristics which would cause trouble to weak decoders.

7.2.1 Test System Architecture

These tests rely on the injection by the SUT adapter of encoded messages based on the testcase name and on timing after `TriMap()` call.



```
testcase TC_BADASPEC_I() runs on SipSimuEx
        {
                tc_defaultTimeout.start;
                alt
                {
                        []sipPort.receive(OPTIONS_Request:?)
                        {
                                setverdict(fail);
                        }
                        []sipPort.receive
                        {
                                setverdict(pass);
                        }
                        []tc_defaultTimeout.timeout
                        {
                                setverdict(fail);
                        }
                }
        }
```

Figure 4 Torture architecture

```
bool SipTorturePort::Map (const PortId& connected_port_id)
```

```
{
        string filename ("data/");
        filename += GetTestcaseId().GetObjectName();
        filename += ".dat";

        cout << "Reading testcase data from " << filename << endl;

        ifstream in (filename.c_str(), ios_base::in |
ios_base::binary);

        if (!in) {
                cerr << "Cannot open " << filename << endl;
                return false;
        }

        in.seekg (0, ios_base::end);
        streampos size = in.tellg();
        in.seekg (0, ios_base::beg);

        char* buff = new char[size];
        in.read (buff, size);

        EnqueueMsg (connected_port_id, MappedBitstring (buff,
size*8));

        delete buff;

        return in.good();
}
```

All test cases look more or less the same since templates are only specified at the message type level, e.g., any INVITE_Request. The verdict is assigned depending on decoding result. In the following test cases example a SIP message is actually not valid and should therefore only set the verdict pass if it could not be decoded.

```
      testcase TC_BADASPEC_I() runs on SipSimuEx
      {
            tc_defaultTimeout.start;
            alt
            {
                  []sipPort.receive(OPTIONS_Request:?)
                  {
                        setverdict(fail);
                  }
                  []sipPort.receive
                  {
                        setverdict(pass);
                  }
                  []tc_defaultTimeout.timeout
                  {
                        setverdict(fail);
                  }
            }
      }
```

```
OPTIONS sip:user@example.org SIP/2.0N)
Via: SIP/2.0/UDP host4.example.com:5060;branch=z9hG4bKkdju43234
```

```
Max-Forwards: 70
From: "Bell, Alexander" <sip:a.g.bell@example.com>;tag=433423
To: "Watson, Thomas" < sip:t.watson@example.org >
Call-ID: badaspec.sdf0234n2nds0a099u23h3hnnw009cdkne3
Accept: application/sdp
CSeq: 3923239 OPTIONS
l: 0
```

8 Installation instructions

8.1 T3DevKit installation

8.1.1 Standard installation

1. Install Cygwin (http://www.cygwin.com/), including the following packages:

    - gcc
    - gcc-core
    - gcc-g++
    - gdb
    - subversion
    - flex
    - bison
    - boost
    - boost-devel
    - libboost
    - make
    - diffutils
    - autoconf
    - automake

NOTE: do **not** install gcc4 (there is an incompatibility with telelogic's runtimes)

2. Download the sources from the svn repository

    - authenticated access:

```
      svn co  --username login
https://scm.gforge.inria.fr/svn/t3devkit/t3devkit/branches/stf370/
t3devkit
```

or

```
       svn co
svn+ssh://username@scm.gforge.inria.fr/svn/t3devkit/t3devkit/branches
/stf370/ t3devkit
```

- anonymous access:

```
         svn co
svn://scm.gforge.inria.fr/svn/t3devkit/t3devkit/branches/stf370/
t3devkit
```

3. Generate the configuration scripts and makefiles templates

```
       cd t3devkit && ./autogen.sh
```

4. Make your TTCN-3 tool accessible (there should not be any space in their path)

```
       mkdir /opt
       ln -s /cygdrive/c/Program\ Files/Elvior  /opt/
       ln -s /cygdrive/c/Program\ Files/Telelogic  /opt/
```

5. Configure and install the toolkit as explained in the user manual

```
       ./configure --with-telelogic=/opt/Telelogic/Tester_3.1
       make
       make install
```

Note: the configure looks for boosts headers in `/usr/include/boost` by default, in the case it does not find them, then you may need to make a link to the adequate directory:

```
       ln -s boost-1_33_1/boost /usr/include/
```

6. Compile and execute a sample test suite to verify the installation

```
       cd examples/DNSTester
       make
       make exec
```

7. Troubleshooting

If you are using gcc version 4 then the executable test suite will very likely return immediately with the error code 5. This is a known incompatibility between Telelogic's runtime and gcc4 on cygwin and it should not happen since you have read the manual from the beginning. This may happen anyway if you have both versions installed on the system. In this you have two workaround alternatives:

- uninstall the gcc4 packages
- ensure that the gcc and g++ map to gcc version 3 (try to execute `gcc --version` or `g++ --version`). This can be done by placing a shell script named gcc/g++ in /usr/local/bin that executes the right compiler (gcc-3/g++-3)

8. Have fun!

8.1.2 Addendum: installation with Message Magic

----

a. Perform steps 1 to 4 of the previous section and include the following packages in the installation of cygwin:

   - gcc-mingw
   - gcc-mingw-core
   - gcc-mingw-g++

b. Get the latest boost binaries compiled for mingw32 (it is available on the H: drive) and extract it into `c:\cygwin\opt`

```
      H:\STF370\WP2 - IMS case study\codec\t3devkit\boost-mingw-gcc345-1.38.0.tar.gz
```

   make a symbolic link to make it easily accessible:

```
      ln -s boost-mingw-gcc345-1.38.0 /opt/boost-mingw
```

c. Configure the toolkit as explained in the user manual

```
      ./configure --target=mingw32
--with-mmagic=/opt/Elvior/MessageMagic5
--with-target-cppflags=-I/opt/boost-mingw/include
--with-target-ldflags=-L/opt/boost-mingw/lib
```

d. Compile host applications first (they require different options)

```
      (cd t3cdgen  && make CC=gcc CXX=g++ LDFLAGS= CPPFLAGS=)
      (cd t3devlib && make t3devkit-config.exe CXX=g++ LDFLAGS=
CPPFLAGS=)
```

e. Compile the rest of the toolkit and install it

```
      make
      make install
```

f. Compile and execute a sample CoDec to verify the installation

```
      cd examples/HelloWorld
      make CPPFLAGS='-Ic++ -I/opt/boost-mingw/include'
T3DK_LDFLAGS='-L/opt/boost-mingw/lib'
```