N° d'ordre: 3585

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Ariel SABIGUERO YAWELAK

Équipe d'accueil : DIONYSOS - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

*From Abstract Test Suites (ATS) to Executable Test Suites (ETS):
A Contribution to Conformance and Interoperability Testing.*

*Des suites de tests abstraits aux suites de tests exécutables:
Une contribution au test de conformité et d'interoperabilité*

soutenue le 26 octobre 2007 devant la commission d'examen

M. :	Héctor	CANCELA	Président
MM. :	Ana Rosa	CAVALLI	Rapporteurs
	Ina	SCHIEFERDECKER	
M. :	Gerardo	RUBINO	Examinateurs
	César	VIHO	
	Nora	SZASZ	

Acknowledgments

I would like to thank Héctor Cancela, who honored me being the Président of the jury, and being a strong and great influence during my grade and postgrade career. I thank Ana Rosa Cavalli and Ina Schieferdecker for accepting being Rapporteurs, for all their remarks and comments that helped enhancing the final document. I would also like to thank Gerardo Rubino, Director of DIONYSOS -formerly ARMOR- team, for taking part of the jury, and for inserting me in the research team. I thank Nora Szasz, Director of this thesis work, not only for accepting such burden, but for giving me decisive support and encouraged me to make this real. I also thank, deeply, César Viho, Director of this thesis too. I thank César for his technical support, scientific guidance, thorough reviews, comments and his human character, not only for supporting me, but for giving me a friend far from home, making this co-tuttele less colder.

I thank my family Baby, Olga and Maria Eugenia for their love, support and patience. It has been a great effort for our small family to put up with the physical distance that only love can bridge. I could not have make it without you.

Thanks all you guys, testing guys, who worked with me during these six stages à la France. You made this possible too, and I am honored of having worked with you. Thanks for sharing expertise and knowledge, giving me your ideas and accepting mine during the different projects we shared. Thanks Annie Floch for your amitié, influence, experience and kindness. Thanks Anthony Baire (beeeer) for all the friendship, experiences and tough work we shared. Thanks Antoine Boutet for your friendship, craziness and enthusiasm. Thanks Frédéric Roudaut for the work and moments shared. Thanks to other testing thesards that shared your work with me during this period: Alexandra Desmoulin, Francine Ngani and Kamal Singh.

I thank the members of ARMOR and DIONYSOS teams for receiving me and helping in so many things. They made the work atmosphere enjoyable and pleasant. Thank you guys who shared these last years with me, Fabienne Cuyollaa, Guilles Guette, Alexandre Guitton, Yézékael Hayel, Louis-Marie Le Ny, Joanna Moulierac, Fatma Othman and Martín Varela.

I would also like to thanks my friends back home in Uruguay for your patience, help and understanding.

Contents

Introduction

This chapter gives a glimpse on the topics in this thesis, motivating the relevance in the practice. It suggests the importance of not only building correct systems, but being able to ascertain their correctness. A global overview of test case implementation and execution requirements is given. At the end of this chapter a synopsis of the contents of this work is given.

Motivation

Computers are everywhere, and nobody discusses that we will keep producing them more and more. We are rapidly extending their field of application to anything imaginable. From wearable music players to planes, from mascots to weapons. Not only sensors, but actuators are digital and software based nowadays. We are rapidly replacing legacy mechanical control devices with cheaper software driven ones. Music players are digital now and other household appliances like refrigerators, watering devices and central heating are digitally controlled.

All these devices must be networked, that is the new trend. Some of them are still connected through proprietary networks, but the tendency is to adopt standard and large scale networks, or at least, the capability of being connected to. Generally mobile applications adhere to cellular network protocols while static ones are devised to be connected to fixed networks. Convergence in the communications field is merging networks on the new Internet.

Explosion of software driven connected devices implies that the correct functioning has a deep social impact. Tight time-to-market cycles imposed by current commercial practices often do not consider the results that faulty systems might produce. Moreover, commercial software development practices from the PC industry, that consider acceptable to release products and fix them via patches after errors are encountered, have impacted the software development process unfavorably. These practices have the most noticeable impact on critical systems, specially when the safety of those is considered. As a property, safety measures the impact in death, injuries, loss of equipment or environmental damage due to an error or failure on a system. We evidence these factors when we find in the news events like space probes failing to communicate or human health consequences produced by faulty medical devices. New series of problems are going on the undertow. We started listening that wireless technology to transfer phone book information to cars built in phones can freeze the on-board car display due to a corrupted name. Or that 60% of one brand of luxury German cars had to return to be serviced due to software errors during 2006. Next year, that brand moved to the second place in the German ranking of luxury cars.

The relevance of ensuring correctness will be evidenced with some examples, as it is done in most testing related thesis. Aero-spatial disasters take the headlines, but this selection might show other evidences of the social impact of smaller, but yet relevant systems that fail:

- On 27 February 2007, the Chinese stock market dropped 9%[1]. This apparently inspired heavy selling on the New York Stock Exchange, with a volume about twice normal. At one time, the calculation of the Dow Jones Industrial average was running about 70 minutes behind. Recognizing some sort of computer problem, Dow Jones switched to a backup computer, which over a period of about three minutes updated the indexes. During those three minutes, the index dropped an average of 240 points. This evidently led to some further panic selling. The market fell 546 points, closing only 416 points down. The cause of the software problem is under investigation.

- In January 2003 two important monitoring systems were disabled in a nuclear power plant[2]. The FirstEnergy's systems were affected by the Slammer worm. The plant Process Computer and the Safety Parameter Display System were taken down and redundant analog backup systems took control. The plant was off-line at that time due to maintenance, but consequences could have had a great impact.

- In August 2007 IRISA's network collapsed, bringing down not only computer but telephone networks. The problem was a wrong manipulation of a network cable affecting an IP-phone. The way cables were connected produced a loop in the network which remained undetected by the loop detection features of the network switch. This wrong cabling overloaded the central switch CPU, preventing the rest of the services to be accessible. The services were down for almost 24hs. Fortunately it happened during holidays.

These few examples intend to show how close is the impact of software errors in our digital life. Several Internet sites and magazines keep updated lists of errors and their consequences. Every day they have more material to write about, and flaws are getting closer. Our life has become more digital than what we imagine.

Ascertaining correctness

Ascertaining correctness might be as old as our capacity to invent things: try them before using them in practice. It was not until recently that it become a field of research. Despite the fact that there are results, there is no agreement on the way it has to be done

[1]`http://catless.ncl.ac.uk/Risks/24.58.html#subj3.1`
[2]`http://www.crn.com/it-channel/18839752`

when facing complex implementations. Let's call *validation* to the process of checking a system to verify that it behaves as expected. Validation covers from a luthier listening to his new instrument before delivering it, to a numerical model of an airfoil run on a computer to determine if the design holds required properties.

The expected behavior is named *specification*, which is a collection of all the properties and/or descriptions of the expected behavior to be found in devices. Unfortunately it is very difficult to find precise and unambiguous specifications. Most specifications are based on human languages, which are content-dependent and ambiguous. Maybe one of the most notorious examples is the Internet Engeneering's Task Force (IETF) Request For Comments (RFC) collection, which specifies most of Internet protocols. A system that claims to have been manufactured according to a certain specification, is called an *implementation*. With these terms we can define validation as the process of verifying that an implementation correctly meets all specification requirements.

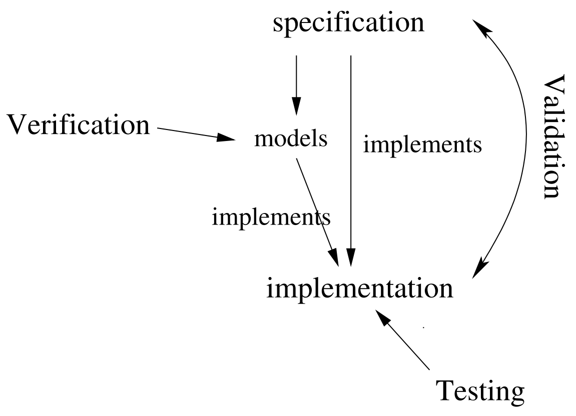


Figure 1: Validation techniques

Different methodological approaches address this subject, some with a deeper mathematical approach, while others hold empirical practices. These approaches are complementary and are named *verification* and *testing* respectively. As seen in Figure 1, the subject of testing is the implementation itself, while verification addresses the mathematical model. A big issue is that most specifications are not ready to be addressed by validation, and a mathematical model, hopefully similar to the original specification, has to be derived. Then the verification subject is the mathematical model, but neither the specification nor the implementation. When things are done properly, verification results are relevant to the implementation and the specification. Testing itself addresses the implementation, and has indeed practical and direct consequences.

From ATS to ETS

In the context of digital communications, and particularly computer networks, there are basically two approaches to testing implementations and ensuring that they will work effectively together: Conformance and Interoperability testing. Conformance testing determines whether a single implementation under test (IUT) conforms or not to its specification (generally a standard, RFC, etc.). Interoperability testing determines the ability of two or more implementations to work together, interacting in a real operational environment. In the context of both conformance and interoperability testing, the final goal is to provide ETS (Executable Test Suites) which are executed against IUT. A lot of work has been done to provide languages for specifying ATS (Abstract Test Suites) and environments for deriving ETS, most notably, the Testing and Test Control Notation version 3 (TTCN-3) language. The problem here is that languages to be used for specifying ATS need to be as abstract as possible, to allow easy specification of scenarios to test. Abstraction helps also in portability and reusability of test definitions, allowing the test expert to concentrate on the main aspects of the test definition. On the contrary, environments used to execute ETS against IUT are designed to be as near as possible to the concrete `niches` of these implementations. So, one can observe that there is a gap between ATS and ETS. Indeed, the work to derive ETS from ATS is still complex, intricate and often error-prone. This makes testers trying to write directly ATS close to the ETS and using the same low-level programming languages as those used for developing the implementations to be tested. As drawbacks, the obtained ETS may not correspond to the test purposes (or scenarios) previously defined in a more abstract level. To be executed on another implementations or in another environment, the so obtained tests have to be completely rewritten.

Growth in the complexity of the systems requires more and better testing. It becomes necessary to find a way to bridge the gap between ATS and ETS allowing the specification of ATS in an abstract high-level language and deriving ETS either automatically, or at least easily. The new version of TTCN tries to provide a solution but still lot of work has to be done to provide necessary tools and environments.

Even though it is implicit in the previous definitions, it is worth mentioning that bridging the gap from ATS to ETS is mostly relevant on testing. Validation, regardless its importance, is a theoretical discipline, and has little requirements for executable level details and know-how. This thesis deals with testing.

It is the common case that the complexity of the ATS to ETS transformation is not understood at first glance. There are more details on protocol execution than is apparent to the eye. No matter how abstract you are on your ATS, all details must be present on your ETS. The problems addressed go further than just a one-time ETS development, but ETS-ATS lifecycle management. Testing is a vivid and moving activity. Protocols evolve, develop new functionality and time-to-market forces always push the limits. It is our responsibility to propose solutions that can follow the requirements evolution, to evolve ATS and ETS in a controlled and reliable manner.

This document is organized as follows. Chapter 1 introduces in more detail the context of the work. Different types and requirements for testing are introduced, together with methodological practices that show a standard test definition process starting from the specifications to the verdict issuing. Main existing problems are introduced. TTCN-3 language and framework is presented as the environment where we will validate our findings. IPv6 is presented too, the suite of protocols that we will address with our tests throughout this work.

Afterward, in Chapter 2 we will present the methodologies designed to ease ATS to ETS derivation and test suite lifecycle management. Our approach to learn and use TTCN-3 language is presented. Different methodological proposals to solve the problem of CoDec generation, how it eases reusability and maintenance of TTCN-3 ATS are introduced too. We also present results and recommendations on how to split the complexity of test case design across the different TTCN-3 API.

Finally, in Chapter 3 we describe different solutions for automating interoperability testing. Proposed solutions were implemented and applied and practical results are presented too. Methodological gains are exposed, with a stepwise explanation of steps required to achieve test execution automation and test platform virtualization.

The work concludes in Chapter 4, where it is summed up and future lines of work are presented.

Chapter 1

Context of the work and state-of-the-art

Any sufficiently advanced technology is indistinguishable from magic
Arthur C. Clarke

This section describes the background of the present work. We ramble on testing concepts and testing disciplines addressed. Different problems that motivate this thesis work are presented.

As the work addresses the gap between abstract specifications and executable test suites, both sides are described. For the application of the results methodological approaches and technologies were used. TTCN-3 language was used as an abstract specification notation, due to its uniqueness as a standard, all purpose testing language. The field of application is IPv6 testing, the proposed replacement of current Internet Protocol version four. As a network layer replacement, it is required to have at least equal IPv4 characteristics, thus ascertaining IPv6 maturity is of industrial requirement. We will introduce the relevant characteristics of both TTCN-3 and IPv6

1.1 Testing

Let's start from what we understand testing is about. The word testing has a Latin origin, *testum*, and comes from the Middle Ages. The testum was an earthen pot, a cupel, used for evaluating precious metals. When impure silver or gold were heated in the porous cup, impurities in the metal were absorbed in the porous material, obtaining a sample of relatively pure silver or gold. The metal has been tested. By the sixteenth century, the word test started to be used figuratively too. To "put something to test" was to make a trial of it, to determine its quality, genuineness, as a precious metal was tested in the testum.

The word "test" is widespread, as we would like to think, the need for it. Test is found as a word in Spanish, accepted by the Real Academia Española and in French too, as in many other languages. The Merriam-Webster associates to the word "test" meanings like: "a critical examination, observation, or evaluation", "the procedure of submitting a statement to such conditions or operations as will lead to its proof or disproof or to its acceptance or rejection" or "basis for evaluation".

Now, closer to our field, we can quote from Glenford Myers that: "testing is the process of executing a program with the intent of finding errors" [Gle04]. The Institute of Electric and Electronic Engineers, IEEE defines test as: "An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [iee90].

1.1.1 The different testing needs

There are different aspects or properties of systems that might be of interest. When we test, we want to add some value to the systems we are testing. Adding value through testing consists of finding and removing errors, thus, raising the quality or reliability. Depending on the expected usage of the system, different properties or aspects might be the subject of our tests. We will continue borrowing definitions from IEEE [iee90].

Let's think of testing an electronic summing device. Just a simple example to illustrate different things that we would like to know about the quality of the implementation. One of the main goals of testing is to ascertain the implementation correctness from its functional point of view. We would like to know that given two numbers, the output provided by the calculator corresponds to the mathematical result of the given addition. We will not discuss right now how to select the numbers to use for testing the behavior, or if we can try to use them all. **Functional testing** is the kind of testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. It is conducted to evaluate the compliance of a system or component with specified functional requirements. Let's assume that we have a functionally correct calculator, tested. We might want to know how long it will last. **Reliability testing** addresses the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

Other aspects that we might need to test might include the performance. For some application, it might not be acceptable that operations take longer than a specific time. In control systems, where time constraints exist, a functionally correct device whose results arrive out of time would not be acceptable. **Performance testing** is the testing conducted to evaluate the compliance of a system or component with specified performance requirements.

Depending how we are going to use the summing device, we might want to know how it works under non standard conditions. Maybe we would like to operate it in extreme climate conditions, beyond its operational range. After overheating it in a car or freeze it in a mountain: will it keep on adding numbers correctly? **Stress testing** is the discipline of testing that is conducted to evaluate a system or component at or beyond the limits of its specified requirements.

We might also want to know how often our calculator will power on and work. **Availability** is the degree of which a system or component is operational and accessible when required for use, and it is often expressed as a probability. Sometimes we are required to provide and design components with availability requirements, and in such cases, we must test that characteristic too.

These examples of testing requirements are not exhaustive, but we just want to make the reader aware that testing is a broad discipline and that there exists several aspects to be tested, even in a simple device.

1.1.2 Different test approaches

Quoting Dijkstra “Program testing can be used to show the presence of bugs, but never to show their absence!”. It is not possible to test a general system to find all its errors. It is often impossible or impractical. Different strategies are taken to design test cases. Two of the most relevant ones are *black-box* and *white-box* testing.

1.1.2.1 Black-box testing

Also known as *data driven* or *input/output driven* it is an important testing strategy. The concept behind this strategy is to use no information regarding the internals of the implementation to test it. No internal behavior or structure knowledge should be used. Testing should concentrate of finding input/output interactions in which the implementation does not behave according to its specifications. Applying this approach, test data are derived only from the specifications.

1.1.2.2 White-box testing

This strategy is also known as *logic driven*, and it allows to design the test cases after examining the internal structure of the implementation. Test data is derived from the examination of the internal logic and structures.

1.1.3 Test case design

Knowing that we cannot test a system completely we face the problem of making the test activity meaningful anyway. A test of any system will be incomplete and the design trade-off is to determine the meaningful subset of all possible test cases that has the highest probability of detecting the highest number of errors. Two main and radically different approaches are followed in the definition of test cases: manual and automatic.

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly. Others, such as advocates of agile development, recommend automating 100% of all tests.

1.1.3.1 Manual design

It might be the most common strategy applied in the practice. Experts determine which are the most relevant test cases and design test cases accordingly. The confidence on the quality of the tests is pretty relative, and might be difficult to provide any metric about it. Coverage is difficult to justify and, in general, metrics regarding manual test cases are obscure and not very promising. On the other hand, the definition process is simple to understand and to modify. The right experts produce high quality test suites.

1.1.3.2 Automated derivation

There are different techniques for automatic derivation. Most of them have solid and strong mathematical bases and have been developed and evolved by well known experts in the area. Today the field has an important synergy with Model Based Testing. In the Object Management Group's (OMG) model-driven architecture, the model is built before or in parallel to the development process of the implementation under test. Starting from the model, test cases are generated automatically. Different problems are faced by automatic generation, like the explosion on the number of possible tests, and their meaningfulness. Cutting the number of tests generated without losing error detection capabilities is a big challenge in the area.

1.1.3.3 Automated vs. manual

There is a very important discussion amongst experts regarding which approach is better. Some of the aspects considered by both groups are collected here. Automated generation supporters consider that manually generated test cases lack of a good coverage and only consider very few test cases. On the contrary, manually generated supporters consider that automatic generated test cases are so complex and expensive to create that are only usable on small and simple laboratory experiments, but remain unusable for real, state of the art, applications and/or systems. A fact that supports this assertion is that it is very difficult for automatic generated test cases to be able to match manual ones on error detection on real application scenarios.

Another problem comes from the definition of the systems that are being tested. Many of the system definitions are done in some ambiguous way, perhaps using natural languages or incomplete system specifications. It is very difficult to define and derive a test specification in this context automatically. Using manual definition it is possible to avoid solving these problems during test case definition, and postpone it to test case implementation stage.

1.2 Conformance and Interoperability

Based on the previous discussion, we can define more precisely the subject of the work on this thesis. The approach followed is the black-box testing one. We will concentrate on functional testing, particularly on the disciplines of conformance and interoperability testing.

Different methodologies exist for conformance and interoperability test case definitions and generation [RC91, FJJV97, CR05], which include formally (ioco, ioconf, mioco, etc. [Tre99, vdBRT04]) and manually generated test case definitions. This work does not deal with test case design. We will take as input both manually generated test suites or automatically derived ones. This thesis is on executing test cases, starting from their given definitions. We take the test case definitions in the form of Abstract Test Suites. We study, solve, and propose solutions to the problems found in the process of turning them into executable test suites.

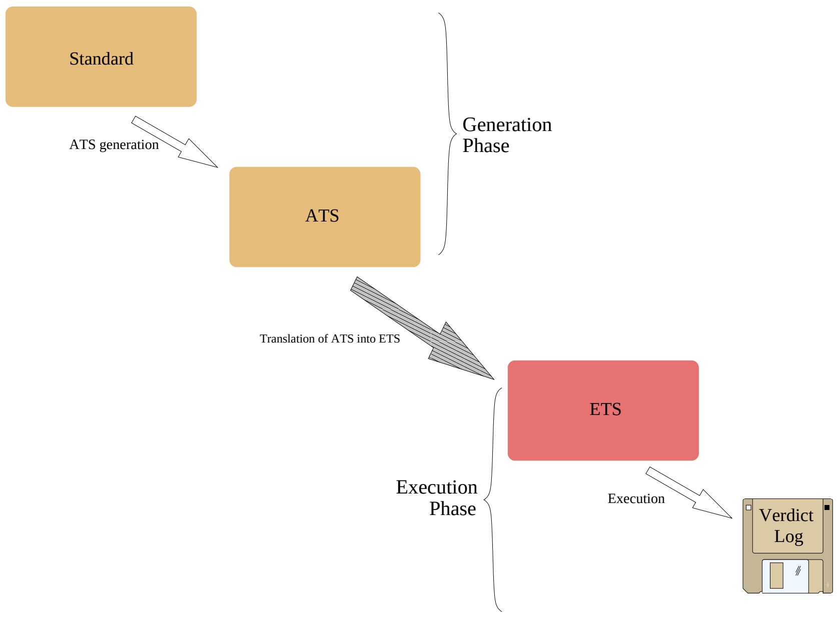


Figure 1.1: Test generation phases

Classically the realization of tests is divided into two phases: generation and execution phases. The generation phase starts from the specification that defines the implementation that we plan to test. Starting from that specification, the generation phase produces the Abstract Test Suite (ATS). The execution phase takes the Executable Test Suite (ETS) and runs it on the IUT, producing traces and execution logs. Afterwards, logs, traces and other data gathered during the ETS execution are analyzed and the corresponding verdict is issued. The binding of the two phases is not standardized.

1.2.1 Steps from ATS to ETS

Different approaches exist for turning an ATS into an ETS. The initial approach is to write directly the ATS using ETS language. We collapse both steps into a single one, without having any high level specification of the test suites. The source code of the ETS becomes the specification of the test. ETS specification languages do not have all required elements for unambiguously defining test cases. Either we restrict the expressiveness of the test cases to what is available in the programming language or we generate particular test libraries for each language. In this approach there is no standard way of defining tests in an abstract way. Moreover, it is difficult for a test expert to understand and work on test specifications if he is not an expert in the programming language too. Test specifications are not portable between platforms and languages. Abstraction is required.

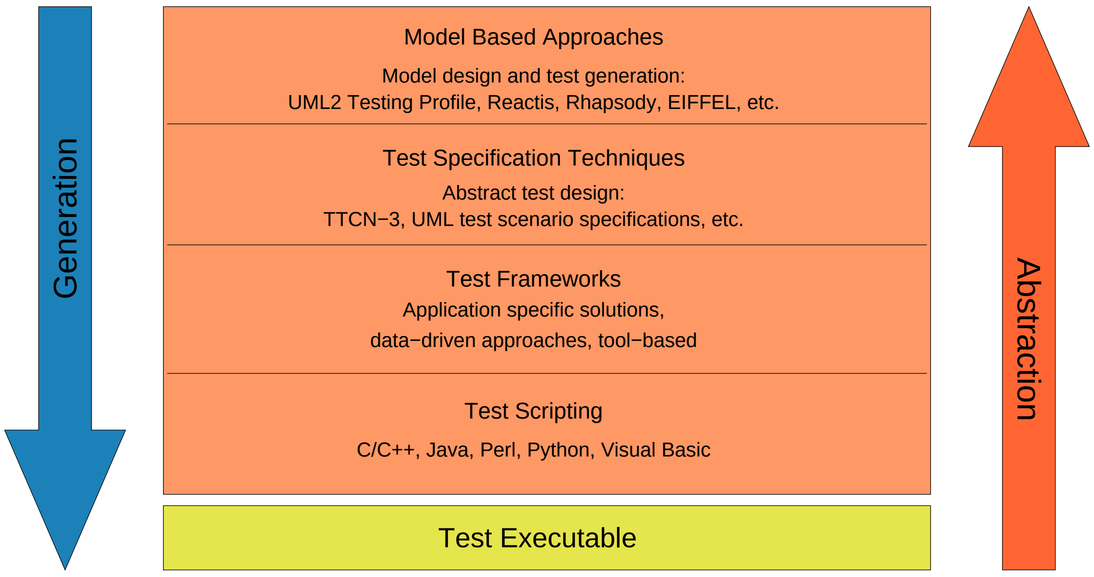

Generation
Model Based Approaches
Model design and test generation:
UML2 Testing Profile, Reactis, Rhapsody, EIFFEL, etc.
Test Specification Techniques
Abstract test design:
TTCN−3, UML test scenario specifications, etc.
Test Frameworks
Application specific solutions,
data−driven approaches, tool−based
Test Scripting
C/C++, Java, Perl, Python, Visual Basic
Test Executable
Abstraction

Figure 1.2: Abstraction vs generation

Figure 1.2 shows graphically how different levels of abstraction are stacked. What we have been describing is named Test Scripting, the lowest level of abstraction possible over the executable. From that specification, compilation or interpretation suffices for executable test generation. Languages used are regular programming ones.

The next level of abstraction is provided by Test Frameworks. They are domain-application specific tools, which are mostly data-driven and poorly configurable. There is no access to the internal implementation of the tool, maybe they support some scripting language where automation or extensions could be defined, but they are constrained to particular niches. Even though they are useful, they are not general purpose testing solutions, thus, they do not provide enough freedom to the test developer. Even though they can automate several tasks properly, a very tight provider dependency is generated, and most of the time it is required to wait for a tool update to extend testing capabilities (if there is a new release).

The next level of abstraction is based on general purpose Abstract Test Specification languages or tools. Most notably we can mention the TTCN-3 and Unified Modeling Language (UML) Test Scenario Specifications. TTCN-3 is an evolution of the TTCN language which escaped from the ISO 9646 methodology and evolved into a general purpose test specification language. UML 2.0 Testing Profiles are new UML extensions for test specifications, but they are part of a broader and more abstract vision of system design, which is the complete UML. It can not be separated from the complete framework.

At today's highest abstract level, Model Based approaches start from a model of the system, which is used for deriving the abstract test specifications. The model of the system can be used only for formal abstract test case derivation, as it is the case for Eiffel, or can be used for both test derivation and system definition, which is the case of UML. U2TP can be used to generate TTCN-3 abstract specifications, which can be later turned into ETS written in C or Java and later compiled into an executable.

It can be seen that as we add layers of abstraction, the gap between the test executable and the specification grows.

1.2.2 The subject of this thesis

This thesis deals with the general problems faced when turning an abstract test specification into an executable one. The specific problems and details might vary from approach to approach, but their underlying nature is the same. Whenever we choose an abstraction, detailed things must be left aside. But the abstract specification must be augmented until the executable one is produced.

How do we provide additional test case dependent, IUT dependent and test suite dependent details that must not be part of the abstract specification? How do we describe generally a communication process without specifying the intricacies of the protocol assembly and disassembly? How do we achieve reusability and all the other software engineering principles required by current development practices? How can we solve executable problems in such an abstract way that allows us to factorize the behavior and provide an abstract vision of the concrete problems? These are the kind of questions we address in this work and provide answers to.

To be able to apply our answers and validate or refuse our findings, concrete abstraction and application technologies were selected. This fact does not interfere with the general and theoretical value of the problems addressed. It supports its validity. In this thesis we propose two sets of solutions. The first set proposes tools and solutions for translating ATS into ETS, and are presented in Chapter 2. The second set of solutions aims at facilitating execution of interoperability testing, which are described in Chapter 3.

The ATS specification language selected is TTCN-3 and the protocol suite is IPv6. The reasons behind these choices will become clearer after they are presented and motivated in sections 1.3 and 1.4. TTCN-3 is both new and unique, making it a challenging ATS specification language, methodology and framework to research on. IPv6 is new, fashionable, sexy and tested very roughly, without the levels of abstraction desired by the academy, but with the empirical methodological proof offered by IPv4.

We will show solutions to make TTCN-3 language abstract test specifications easier to turn into executable ones. We will also present how to abstract executable problems of IPv6 testing.

1.3 Testing and Test Control Notation version 3

The Testing and Test Control Notation version 3 (TTCN-3) is a language for defining test specifications for a broad range of telecommunication and computer systems. It is internationally standardized and actively promoted by the European Telecommunications Standards Institute (ETSI).

TTCN-3 is computationally complete, safe typed, procedural language with deep roots in the telecommunication domain. The field of application has been growing over time and success stories have been published on almost any field of testing, ranging from web services to railway systems.

This section introduces the language from its origins. Core language and runtime characteristics are presented. Key aspects for this thesis are highlighted.

1.3.1 When TT was neither Test nor Testing in TTCN

The Tree and Tabular Combined Notation (TTCN) was born back in 1984 by the International Organization for Standardization / International Electrotechnical Commission (ISO/IEC) Joint Technical Committee (JTC) 1/Sub-Committee (SC) 21 and in the Comité Consultatif International Téléphonique et Télégraphique (CCITT) Study Group (SG) VII. It was part of the OSI conformance testing methodology and framework. In 1992 it was standardized as ISO/IEC 9646-3 [ttc92] and CCITT Rec. X292 [cci92], as one of the set of seven texts of the respective ISO/IEC 9646 and CCITT X.290 series.

Several European standard organizations (ITU-T, ATM Forum, amongst others) applied the TTCN language for describing abstract test suites, mainly for conformance of communication protocols. Several reasons explain the acceptance. Matching mechanisms provided unambiguous means so that conformance of received messages can be automated and evaluated against the test purpose. The notation was easy and natural to use according to existing standards and maturity of programming languages. The uniqueness of a verdict system embedded in the language itself facilitated conformance judgment.

The first version of the language did not include built-in functionality to describe concurrent behavior within the tester to deal with general concurrency efficiently. Other concepts of structured languages aiming reusability and encapsulation were not clearly present. Constructs like modules and packages were a new requirement too. The new version of the language, TTCN-2 addressed them. It also included enhancements in ASN.1 type handling. The new definition was also standardized by ISO/IEC 9646-3 1998 [ttc98] and ITU-T [itu98] in 1998.

Regardless the enhancements introduced to the new version, the whole methodology and language design were much influenced by OSI protocols and conformance testing in mind. Other kind of testing started being required by the industry and studied by the academy. Remarkably, Internet Protocol requirements of interoperability testing evidenced weaknesses of the language. Internet Protocol, and other protocols that did not show good isolation between layers proved to be difficult to address with TTCN-2 language. Requirements like those presented in robustness testing, regression testing, system testing and integration testing were not considered. Mobile protocol testing, service testing module testing were not addressable either. A major redesign of the approach was required. Two new Specialist Task Forces (STF) were created at the European Telecommunications Standards Institute (ETSI) to address this new evolution of the language: STF 133 and STF 156. Work began on 1998 and was completed by October 2000.

Discussion was tough and characteristics included in the new version of the language were much discussed too. The abbreviation (TTCN) was maintained, but with a different source, showing the change of the underlying technology. In spite of the fact of the redesign of the language, TTCN-2 features were retained as much as possible to keep the investment of companies and organizations. The notation drastically changed, and the new look-and-feel is the one of a modern programming language. Tree and Tabular notation were removed from the core language. They were converted into graphical representations that can be translated into TTCN-3 language. Currently, Tabular notation and Message Sequence Charts (MSC) are standardized translations to and from the core language. Well defined syntax was obtained and language's operational semantics was defined. Other advances in computer science available during those days were not introduced in the language, most notably, object orientation capabilities. There is some resemblance of object orientation in the dotted notation used for some signaling operations, but object orientation was left aside.

1.3.2 The new TTCN-3 language

During October 2000 TTCN-3 was approved and standardized [ttc00] by ETSI. In 2001 it was also standardized by the ITU-T as Z.140 [itu01a] and Z.141 [itu01b].

The language was designed with a more powerful textual syntax, that can define the complete semantics of the test cases, as it is a language designed specifically for testing. Most of the concepts and constructs of the language are similar to others in imperative, procedural programming languages. A basic set of generic constructs is extended with additional concepts, specific for testing. Extended concepts are suitable not only for verdict handling, but message assembly, reception and matching. Moreover, concepts like distributed test system architecture, concurrent execution of test components and dynamic configuration of the tester are supported too. Communication paradigms like message based or procedural based communication are implemented. Timer constructs are included in the language. The language as a whole is better suited for meeting emerging test needs.

There is a general agreement in the fact that the main contributions of TTCN-3 as a language are the following:

- data and signature templates with wildcard based matching mechanisms,
- type and value parameterization,
- different presentation formats, including standardized ones, and the possibility for providers to develop their own presentation formats,
- dynamic concurrent testing configurations,
- operations for synchronous and asynchronous communications,
- ability to specify encoding information and other attributes (including user extensibility),
- assignment and handling of test verdicts,
- test suite parameterization and test case selection mechanisms,
- combined use of TTCN-3 and ASN.1 (and potential use with other languages such as IDL).

How these functionalities are achieved and the component design of the TTCN-3 language is the subject of the following sections.

1.3.3 TTCN-3 architecture

The objective of a testing language is to provide a comfortable and abstract environment for test specification. Despite the fact that abstract operations can be defined, it is required to execute low level operations on the service or protocol being tested. Depending on the level of abstraction of the system under test, to generate an executable test system, it would be required to either handle complex message structures in XML notation or maybe to directly handle bits over a serial link. It is not possible to execute an abstract test specification, since execution details have to be provided.

The way TTCN-3 handles test system executable complexity is based on the *divide and conquer* approach. A set of standardized Application Programmer Interfaces (API) that are used to complement the TTCN-3 language abstract definition of the test case and generate a complete executable test system. The TTCN-3 language is not used to handle low level details of test case implementations. These operations are relayed to languages, adequate for low level handling. The standardized and selected languages are ANSI/C++ and Java, even though other non-standard extensions have been presented like a .Net one [Ulr06]. We will refer to Java and C++ as platform languages in the context of TTCN-3. Despite other initiatives, only Java and C++ languages are standardized by ETSI.

Every TTCN-3 test system should be understood (and thought of) as a set of interacting components. Each component performs a different part of the required functionality. Component responsibilities include: manage test execution, execute compiled TTCN-3 code, communicate with the system under test, administer types, values and test components and handle timer operations.

A runtime schema showing the main identified components, the standard API and their interactions is taken from [ETS05c] and shown in Figure 1.3.

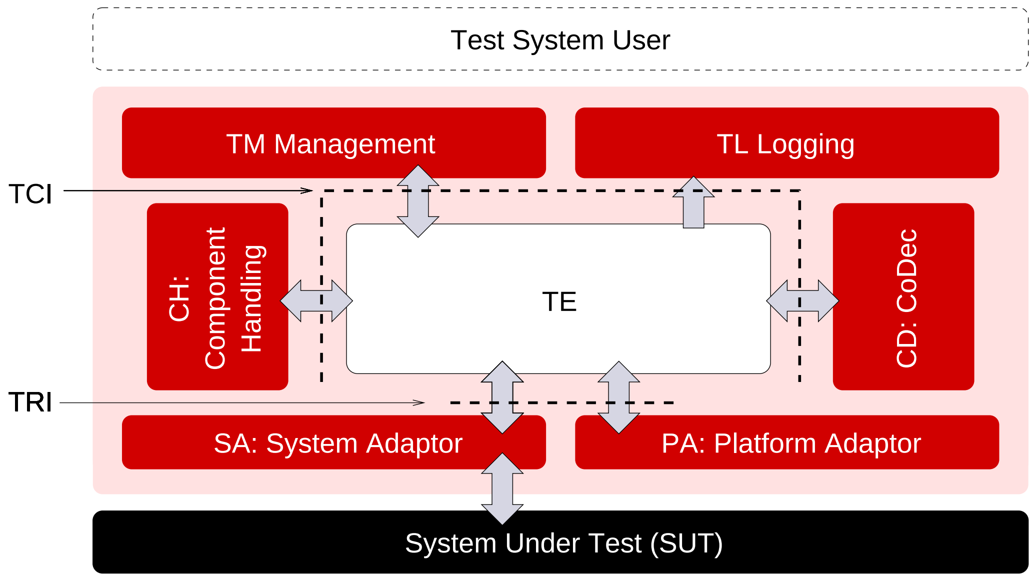


Figure 1.3: Conceptual architecture of TTCN-3

The concept of Implementation Under Test (IUT) is known as System Under Test (SUT) in TTCN-3 standards. The central element of the figure, the TTCN-3 Executable (TE), executes TTCN-3 modules. Control of the test case, components, values and queues are amongst TE structural elements, as defined by modules or TTCN-3 language definition itself. Support for test case distribution is part of the language, thus, TE may be executed in a centralized or distributed manner. Distribution of Parallel Test Components (PTC) can be done over a single test device or across several ones.

Despite the abstraction of TTCN-3 language, low level operations have to be implemented in some way. TE implementation is an abstract level description of a module and other entities defined by the standard. The entities and their interactions are used to provide specialized implementation details required for test case execution. Other entities of a TTCN-3 test system make these abstract concepts concrete. The language and test system architecture were designed so as to separate concerns amongst entities. As an example, the abstract concept of sending a certain message has the abstract keyword `send()` as part of the TTCN-3 language, but it must be complemented with entities that tell how to encode the message and send it over a concrete physical medium. The API for complementing the abstract test specification is split into TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI).

The TTCN-3 Runtime Interface defines the interaction between the TE, SUT Adaptor (SA) and Platform Adaptor (PA). It is defined in [ETS05b]. It provides the means for the TE to send test data to the SUT, receive responses or handle timers amongst other tasks. TRI is split into two bidirectional sub-interfaces: triCommunication and triPlatform interfaces. The triCommunication interface addresses the communication with the SUT, which is low-level implemented in the SA. The triPlatform allows the customization of a particular ETS to a specific execution platform.

The triCommunication Interface collects the operations required to implement ETS-SUT communication. API calls can be grouped in Test System Interface (TSI) initialization, SUT connection establishment, message based communication and procedure based communications.

The triPlatform Interface adapts the TTCN-3 executable to a particular execution platform. It provides primitives to handle timers, access to external functions and maintenance operations on the component.

The TTCN-3 Control Interface (TCI) defines the interaction between the TE and the Test Management and Control (TMC), and is defined in [ETS05c]. The TMC entity includes functionality related to management of test execution, components, coding and decoding of test data exchanged with the SUT. The Test Management (TM), Coding and Decoding (CD) and Component Handling (CH) entities conform the TMC. The TM is responsible for the overall management of a test system. After initialization of a test system, actual execution starts within the TM entity. It is responsible for the proper invocation of TTCN-3 modules, including configuration of test dependent parameters of the actual execution through module parameters. It is usual that this entity has a tight correlation with the user interface of the test system. The CD is the entity that provides means to transform TTCN-3 representation of messages into transmittable bitstrings and vice versa. The TE determines which CoDecs can be used, it hands the TTCN-3 data to the appropriate coding routine to obtain a bit oriented, transmittable representation of the message. When data is received from the SUT, it is decoded in the CD entity and converted into TTCN-3 values. The CH entity handles parallel test case execution. As the TE can be distributed among several test devices, the CH provides services to synchronize test system entities.

Each testing node of a test system includes TE, SA, PA, CD and TL entities. The CH and TM controls the execution of the different TEs on each node. The TE which starts a test case is a distinguished one, who is responsible of computing the test case verdict. This is the only distinction amongst different TEs.

The remaining entity of the TCI is the Test Logging (TL). It performs the task of gathering log information from the other entities for test event logging and presentation to the test system user. It provides information that allows the expert to know what is going on inside the test system. Creation of test components, information sent to or received from the SUT, template matching and timer operations are logged, amongst other.

1.3.4 On TTCN-3 language design

The TTCN-3 language addresses the challenge of being a general purpose testing language. It aims at providing the testing community testing tool provider independence, and full standardization of test case behavior. The broad spectrum of testing activities, ranging from unit testing, functional and non-functional testing, classical conformance or model driven test generation makes the language a complex one. The design approach was to include as many keywords in the language as required to meet all testing needs, and keep as much compatibility as possible with previous versions of the language. The result is a large language, consisting of 140 keywords, as shown in Table 1.4.

If we compare it with other general purpose languages we can see the difference in the approach. As an example, Java 5.0 has 50 language keywords[1]. Java also counts with 37 operators, some of which are keywords in TTCN-3, specially logical and bitwise operations.

Another example are the C/C++ languages. Depending on the version and compiler considered, C reserves about 30-35 keywords and C++ has 30 more. How is it possible to develop complex systems using either C++ or Java? Expressiveness and power is removed from the core language and relayed to libraries, keeping the language minimal. In the case of C and C++, those libraries are system dependent and this the problem of lack of standardized libraries for all platforms is one of the biggest challenges/obstacles for C portability. In Java the set of classes is distributed with the virtual machine, allowing the Java platform to provide homogeneous services across the different platforms where it can be deployed. Right from the design stage, these languages consider and provide ways of distributing pre-compiled code: object files, class files, libraries, etc. TTCN-3 standard does not address these issues.

TTCN-3 language on the contrary provides an extense core language, but no standard set of libraries to the developer. Test specification projects must start either from scratch or from tool provider proprietary extensions. Using proprietary extensions threatens the tool provider independence objective. Developing tool independent libraries, starting from scratch, puts additional complexity to the test case generation process. The amount of work to be done for producing a tool vendor independent testing library is a considerable burden.

1.3.5 Catching up TTCN-3

TTCN-3 language has a step learning curve, even after the book "Introduction to TTCN-3" [WDT+05] was published. Despite the inherent complexity of learning a complex, extense and specialized language, there is an additional learning cost in understanding the complete architecture and interaction of entities. As discussed before, language

[1] `http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html`

action	error	match	return
activate	except	message	running
address	exception	mixed	runs
alive	execute	mod	select
all	extends	modifies	self
alt	extension	module	send
altstep	external	modulepar	sender
and	fail	mtc	set
and4b	false	noblock	setverdict
any	float	none	signature
anytype	for	not	start
bitstring	from	not4b	stop
boolean	function	nowait	subset
case	getverdict	null	superset
call	getcall	octetstring	system
catch	getreply	of	template
char	goto	omit	testcase
charstring	group	on	timeout
check	hexstring	optional	timer
clear	if	or	to
complement	ifpresent	or4b	trigger
component	import	out	true
connect	in	override	type
const	inconc	param	union
control	infinity	pass	universal
create	inout	pattern	unmap
deactivate	integer	port	value
default	interleave	procedure	valueof
disconnect	kill	raise	var
display	killed	read	variant
do	label	receive	verdicttype
done	language	record	while
else	length	rem	with
encode	log	repeat	xor
enumerated	map	reply	xor4b

Figure 1.4: TTCN-3 terminals as of version 3.2.1

design is not minimal, and did not follow other tendencies of the late ’90s. Lack of on-line tutorials, mailing lists and forums leave the newcomer alone with a difficult challenge. No collection of *good practices* or recommendations exists. Moreover, no clear agreement amongst experts exists right now.

Publicly available ETSI language standards are the main and comprehensive source of information about the language, but it provides only raw and too technical information for the beginner. Indeed, it is not the objective of a language standard document to be a tutorial, but in practice TTCN-3 standard ends up to be a language reference too.

As a language, TTCN-3 can be considered a Pascal-like language, mainly imperative and structured. The main complexity does no arise from understanding the language, but the complete TTCN-3 architecture. Effective usage of runtime entities, reusable and maintainable code is difficult to learn to write. The general case is that TTCN-3 code is considered the important part of the ATS, and constitutes the place where the test expert is supposed to be concentrated. Other entities are considered just helpers and not important, but we believe that disregarding their importance is a source of test case development complexity.

Some authors consider that producing an executable test suite out of an abstract test suite written in TTCN-3 language is just a matter of compilation [Gra94]. Back in 2004, at the beginning of this thesis work, it was very difficult to understand why it was required to research on the subject “From Abstract Test Suites to Executable Test Suites” based on these strong statements. It seemed that there was nothing more to be done there. It was required the strong conviction of the thesis director to explain the relevance of the subject once more. Indeed, it is true that a part of the ETS can be built just compiling the TTCN-3 code, but what is not clearly said is that TTCN-3 compilation is not enough. Developing test specifications without considering the other entities, might lead to highly difficult to develop and maintain CD/PA/SA entities. Comprehension of these dependencies might be the biggest challenge when learning TTCN-3.

1.3.6 TTCN-3 in the Internet Community

In the Internet community in general, TTCN-3 is not widely adopted. As a newcomer into the IP testing arena, it is required to provide more or do better so as to justify the switch from one way of testing to another. Moreover, TTCN-3 language is even unfavorably criticized. This is mainly due to the confusion with its predecessor TTCN-2, which was considered a rigid language and difficult for generating tests for new protocols. This bad reputation applies for testing the new protocols developed for the new version of the Internet Protocol, called IPv6.

Possibly difficulties with TTCN-3 predecessors arise from their relationship to ISO 9649 and ISO’s OSI layered protocol philosophy. The communicating systems addressed by ISO’s 9646 methodology should be layered, with clear bounds, unlike TCP/IP protocol suite.

TCP/IP was born as an experimental suite of communication protocols, that evolved over decades by fixing problems, keeping backward compatibility, without major revisions and without including several advances of the field. IPv6 is not a redesign of the TCP/IP suite, but a replacement of the network layer. Upper and lower layers stay the same, despite some extensions. Thus, TCP/IPv6 suite is as badly layered as the original TCP/IP one. This introduces a lot of problems to modeling, testing and implementing it. The strongest complaints from the Internet Community to TTCN-2 language was that it was not well suited for this scenario.

Indeed, most of the existing test suites are developed using IPv6 dedicated languages and tools. The most famous one is the v6eval toolbox (`http://www.tahi.org`) developed by the Japanese TAHI project. In this context, it is difficult to convince people to use TTCN-3 without showing real executable test suites. TTCN-3 must show how it can solve these known problems, and do it well if it wants to be considered.

1.4 Internet Protocol version 6

IPv6 is short for "Internet Protocol version 6", sometimes also called Next Generation Internet Protocol or IPng. Even though Internet is seen as a new technology, its protocols and building blocks were developed during the '70s and '80s. What we know as Internet and all corporate and private intranets use IPv4.

IPv6 fixes several problems in IPv4, remarkably the availability of addresses. It also addresses enhancements in the message header format and options handling. The new IPv6 protocol suite is a network layer replacement, that preserves lower (data link) and upper (transport) layers as much as possible, with minor adaptation requirements.

In the following subsections we will present IPv6 protocol, why it is important to test it and the existing testing initiatives on the field.

1.4.1 Beginnings of IPv6

The organization behind the standardization and development of the Internet is the Internet Engineering Task Force (IETF). It is an open international community of individuals concerned with the evolution of the Internet architecture and its smooth operation. IETF's effort of scaling IPv4 protocol started in the early '90s and the core set of IPv6 protocols became an IETF Draft Standard on August 10, 1998.

It is clear now that IPv4 needs to be replaced after connecting the Internet for decades. There are several different reasons that motivate the evolution of the Internet protocol, from simply evolution to unforeseen requirements and services at the time it was designed. IPv6 addresses different aspects, but the main changes can be summarized as discussed in the following subsections:

1.4.1.1 Address space growth and autoconfiguration

The address size grew from 32 to 128 bits, solving for some years from now the address exhaustion problem. We will run out of IPv4 addresses in a few years from now and the logical step is IPv6, mainly a refinement of IPv4. The bigger address space also enables new communication mechanisms like anycast and gives better support for multicast than IPv4. Autoconfiguration features are also new in IPv6, allowing devices to get addresses and became ready to communicate without any configuration operation performed by administrators.

1.4.1.2 Simplified header and optional extensions

The protocol header now has a fixed size, which simplifies routing operations by lowering processing costs. Some unused fields were removed and others became optional. In IPv4 the options were part of the basic header, with complex rules of assembly. In IPv6, they are all handled as Extension Headers, inserted between the header and the beginning of the payload. IPv6 also allows the definition of future Extension headers, unknown for the moment. If this option was available in IPv4, then the extra 96 bits could be an extension of the standard IPv4 packet. As extensions are available in IPv6 it may be possible that future enhancements only require the definition of new extension headers.

1.4.1.3 Authentication and privacy

Support for authentication, data integrity and confidentiality extensions have been included since the beginning. The IETF wants that all IPv6 devices support these extensions, but the industry does not follow this directive tightly. Small devices usually omit this requirement.

1.4.1.4 Flow labeling

The concept of "traffic flows" is also new in the network layer. The sender can label his traffic, requiring special handling. As an example, it could be used to differentiate traffics with different quality of service restrictions.

During 1999 different groups converged into the IPv6 Forum[2], a world-wide consortium with the mission of promoting IPv6 by improving market and user awareness. Several lines of action were taken by the Forum:

- set up an open, international forum on IPv6, based on voluntary basis,
- share and disseminate knowledge and experience among members and non-members,
- create different chapters and task forces around the world,
- promote worldwide solutions to solve IPv6 deployment problems,
- organize several IPv6 summits worldwide, educating thousands of engineers every year,
- promote a globally unique certification program, the IPv6 Ready Logo.

The following subsection will discuss the importance of the IPv6 Ready Logo initiative in particular and testing IPv6 in general.

[2]http://www.ipv6forum.org

1.4.2 Relevance of IPv6 testing

The Internet is a big network. Statistics published during mid 2007 by `http://www.internetworldstats.com` show that Internet usage is still growing at a very fast rate and reaches more than 1.100 million people. This network runs IPv4, not IPv6. Banks offer services to their customers using IPv4. Companies connect their branches using Virtual Private Networks (VPN) over IPv4. All the e-business we know is based on IPv4 and we will only move to IPv6 after we can ensure continuity of services, availability and reliability into the new Internet. Not only business, but entertainment. People send their e-mails, publish their family photo albums and share their videos on the net. Convergence is happening over IP.

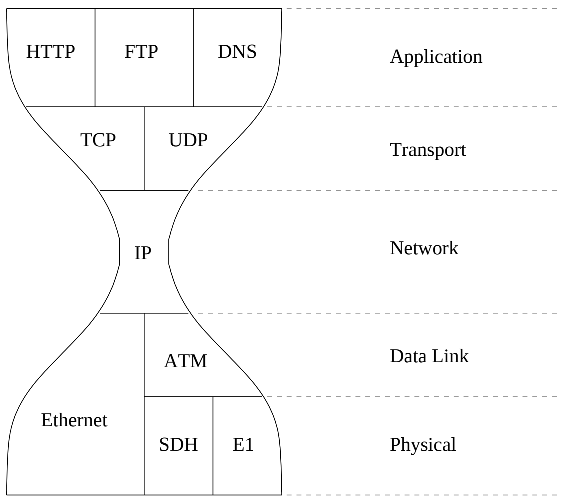


Figure 1.5: IP hourglass model

If we consider Steve Deering's IP hourglass model, shown in Figure 1.5 we might have a clue of the importance of the Internet Protocol. The thin waist of the figure is just what we need to replace. It is a critical change, and no one can deny it. Few people wants to risk something that works, since it can affect business continuity and may lead into losing money: "If it works, don't fix it".

One of the several means to transmit confidence on IPv6 implementations and its maturity is the IPv6 Ready [3] certification program. The program is created and sponsored by the IPv6 Forum. A globally unique certification program addresses the requirement of a global network replacement challenge by promoting confidence in the maturity of the implementations. Unlike IPv4 that started with a small and closed network, the scope of IPv6 covers a huge number of implementations on a global scale. The ability of interoperate (interoperability) has been a critical feature in the Internet community. It is essential that a single symbol identifies products that have been validated for interoperability: the Logo.

[3]http://www.ipv6ready.org

The organizations behind the IPv6 Ready since the beginning are the University of New Hampshire Interoperability Laboratory `http://www.iol.unh.edu`, TAHI Test Event `http://www.tahi.org`, ETSI IPv6 Plugtest `http://www.etsi.org/plugtests`, IRISA `http://www.irisa.fr/tipi` and Connectathon `http://www.conectathon.org`.

1.4.3 IPv6 testing

Different objectives were addressed from the beginning by the different organizations that converged in the IPv6 Ready program. Among the different decisions, the objective was to test the IPv6 protocol, no more and no less. Initially, a first phase was defined to rapidly motivate the vendors and organizations that will deploy IPv6, while the second phase was being defined. The sticker showing compliance with first phase requirements is a silver colored one. The second phase addresses complete requirements testing, and its sticker is a golden one. It ensures equipment and service interoperability and conformance according to the corresponding RFCs. At least, the objective for the test specification is that certified devices should be ready for production networking. The concept of IPv6 Core protocols was coined and become the initial milestone for Phase 2. There is a minor glitch regarding IPsec, that might lead to the third Phase of the Logo: The IPv6 Forum considers IPsec as mandatory in IPv6 implementations, but the silver and golden logos can be obtained without implementing IPsec.

The Internet Community traditionally based standardization decisions on interoperability. Interoperability addresses the ultimate requirement of having implementations working together and still providing their expected services. This pragmatical approach is generally criticized from the more formal side of testing, where conformance testing is the de-facto tool. It is said that it is an inferior way of assessing correctness. Solid mathematical grounds were developed over decades, but even with some simplifications, it is still not feasible to use formal methods to generate complete testcases for complex protocols as IPv6. Interoperability testing recognizes the need of verifying that even non-perfect implementations can still interwork.

The Ready Logo succeeded to include conformance testing as part of the logo. Even though test cases are not formally derived, but defined by experts, for the first time in the Internet Community conformance testing is required to get the certification.

Chapter 2

TTCN-3 based framework to assist ETS derivation

We have to remember that what we observe is not nature herself, but nature exposed to our method of questioning.
Werner Heisenberg

This chapter describes methodologies and solutions to simplify ETS derivation. The ATS specification language used to apply and validate our findings is TTCN-3. At the beginning, in Section 2.1 we present the different adventures we took on the TTCN-3 world. It presents our hands-on initial approach to the language, methodology and tools. The contents of this Section are based on two publications: *Some Lessons from an experiment using TTCN-3 for the RIPng testing* [SFRV05] presented during TestCom 2005 and *Using TTCN-3 in the Internet Community: an experiment with the RIPng protocol* [SBFV05] presented during the TTCN-3 User Conference 2005.

From the bruises acquired, we learned the importance of mastering TTCN-3 architecture and interaction of runtime components. One of our initial approaches was to overcome certain limitations of TTCN-3 logging capabilities, integrating traffic capturing and other features to the System Adaptor. The section 2.2 is based on the work that was presented as *Embedding traffic capturing and analysis extensions into TTCN-3 System Adaptor* [SBV06] during the MMB Workshop 2006.

We found out that the relationship between the TTCN-3 ATS and the CoDecs has a deep impact in the complexity of the implementation of the whole test system. Different approaches to deal with this fact were addressed, which are described in Section 2.3. The implemented solution for this problem is a tool that addresses the automatic CoDec generation for the C++ platform. This solution is currently distributed as Open/Free software [t3d07]. The solution was presented during the TTCN-3 User Conference 2006 as *Towards and IP-oriented testing framework - The IPv6 Testing Toolkit* [SBD+06]. The presentation was invited to take part of a STTT Special Issue on The Evolution of TTCN-3. No reference to the actual journal can be provided as it is not published by the time of this writing. Apart from the previous solution, a proposal for a platform language independent solution is presented in Section 2.5.

Finally, in Section 2.4 we present later experiments using the CoDec Generator for more complex protocol testing and discuss alternatives on test case design and development. The contents of this Section are based on the paper *The new Internet Protocol security IPSec testing with TTCN-3* [SCV07], presented in the TTCN-3 User Conference 2007.

2.1 Hands-on experience with TTCN-3: RIPng

TTCN-3 has been designed to provide a well suited language for any kind of testing activity [UKW99, SVG03, SVG02, VGSB+99], from abstract test suites specification to executable test suites [GD03, Tör99]. As it is a new language, there is not enough maturity regarding its usage and environments that are supposed to ease TTCN-3 usage. The European community, through the European Telecommunications Standards Institute (ETSI), promotes the use of the TTCN-3 language for testing purposes [GH99, GWWH00].

An important objective behind this hands-on challenge was to gain experience using the TTCN-3 language and tools while addressing a pending IPv6 test conformance problem. The Routing Internet Protocol for IPv6 (RIPng [MM97]) has the advantage of being relatively simple (at least compared to other IPv6 related routing protocols), and still being an important and widely deployed protocol in small to medium organizations. This work also aimed at proving to the Internet Community that TTCN-3 can be used for testing, covering all steps from abstract test suites (ATS) to executable test suites (ETS). It was also important to identify main issues when testing with TTCN-3 and providing solutions that may help simplifying future test generation.

The methodology behind this work was restricted in scope as the goal was to be able to obtain ETS to be executed against real implementations during the IPv6 interoperability event organized by the ETSI/Plugtests Service in October 2004. After October, we re-designed some details and produced a new ETS that was executed also against real implementations, with a tight schedule for the TAHI IPv6 Interoperability event in January 2005. We were forced to follow a straightforward approach due to time constraints: some decisions were based on time-to-executable-test parameters. On the other hand, one may note that this kind of requirements also corresponds to the real Internet Community and industry requirements of having ETS available and ready to be used as soon as the need of testing is identified. If we were given more time, it might have been possible to try different modeling alternatives and reach more elegant solutions (unfortunately, it was not the case).

Amongst all available TTCN-3 tools, the choice was made for a tool that allowed us to have access to the source code if necessary. Indeed, due to the youngness of TTCN-3 and our current knowledge in using this new language, it was important to use a tool which allowed libraries source code modification if needed. Work on portability of the ATS across different TTCN-3 tools, where access to tool internals is not required is addressed in Sections 2.3 and 2.5.

As a result of this work, a RIPng conformance ATS/ETS based on TTCN-3 is now available. These tests have been run against real implementations during an IPv6 ETSI-Plugtests Interoperability event in October 2004 and during the Japanese IPv6 TAHI Interoperability event in January 2005. Test results were considered of interest by participants. Doing this work and following the approach indicated above, we faced several issues that any new TTCN-3 user may have to deal with. Amongst other results, the main problems found are highlighted, and our solutions are introduced. Initial ideas that may help in easing test development using TTCN-3 are proposed.

The rest of this Section is organized as follows. Subsection 2.1.1 explains with more details the context of the work and provides a brief description of the RIPng protocol. Main TTCN-3 components that have to be developed are described. Section 2.1.2 outlines different steps to obtain TTCN-3 based test suites for the RIPng protocol. Problems encountered during test development phase and their solutions are also presented. Section 2.1.3 presents some results and lessons learned from this experiments in using TTCN-3 for RIPng testing. Some ideas that might help in easing other similar effort are presented. Conclusions of this work can be found in Section 5, where future work is suggested.

2.1.1 Background of the experience

We have been involved for years in developing IPv6 conformance tests suites. Personally I was a novice by that time, but my supervisor, César Viho, took part of the IPv6 Ready Logo from the beginning. César is the European technical comisaire, and I was glad to be received in his laboratory and contact the great group of experts working there.

The *de facto* tool used by the Internet Community is `v6eval`, developed by TAHI project (`http://www.tahi.org/`). Following IPv6 Ready Logo conformance testing recommendations, we worked to produce test suites for several IPv6 routing protocols, in particular RIPng, the experience described here.

One important reason behind the present work for us was to find provider-independent tools and languages for defining test suites. TTCN-3 is presented as a modern standardized abstract language, test oriented and provider independent. Tool providers implement their solutions according to the standards, but independently. It is widely accepted that multi-provider scenarios lead to more complete and general languages and tools than single provider ones. The lack of free/open reference TTCN-3 implementations also presents some limitations to a the Internet Community. The Internet Community has been working with open/free tools and operating systems for testing purposes.

Our primary motivation was to experiment with the ability of TTCN-3 for our testing purposes with real and concrete IPv6 protocol. On the other hand, we wanted to show to the IPv6 community that TTCN-3 can be used for this purpose. One way to prove that is to have executable test suites built with TTCN-3 language and tools, which can be used during interoperability sessions.

2.1.1.1 RIPng brief overview

RIPng[MM97] is the logical step of the well known IPv4 family of RIP protocols into IPv6 world. RIPng stands for *Routing Information Protocol - Next Generation*. RIP belongs to the class of algorithms known as "distance vector algorithms". Distance-vector algorithms are based on the exchange of only a small amount of information. Each network node that participates in the routing protocol must be a router as IPv6 protocol provides other mechanisms for router discovery, and it is assumed to keep information about all destinations within the system.

Limitations of RIP include network diameter restrictions, counting to infinity to resolve loop situations. Other drawbacks arise from the lack of metrics based on traffic or link cost parameters. Some of the limitations are not *per se* limitations, but they are a consequence of the design of the protocol. RIP is not intended to be used as Internet's single routing protocol, but as an Autonomous System (AS) internal protocol. RIPng is an UDP-based protocol and listens on the port 521. It is a message oriented protocol (implemented messages are 1-request and 2-response), based on distributed intelligence, without any distinguished node. The figure 2.1 shows a typical RIPng deployment scenario, where 6 interconnected routers exchange routing information as request-response messages.

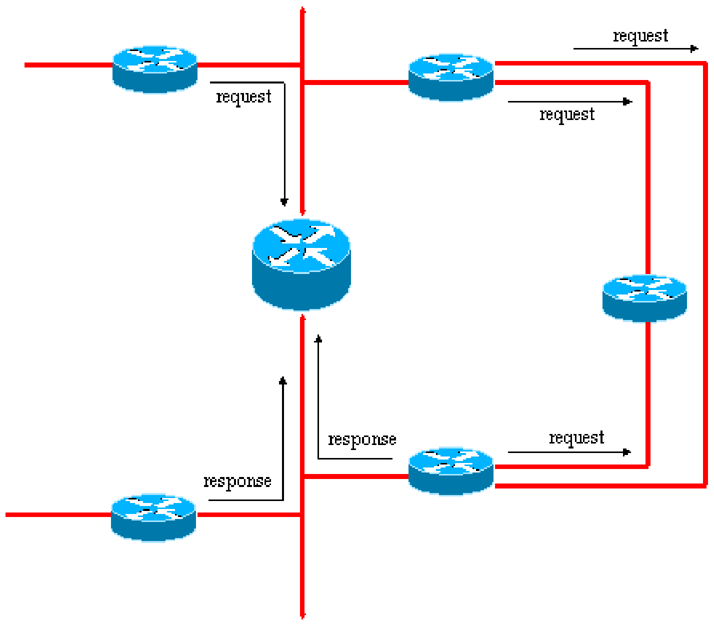


Figure 2.1: Autonomous System RIPng messaging

IPv6 protocol defines and implements three different types of communication destinations, which are: unicast, anycast and multicast. These enhancements at network/transport layers provide better support for protocols using their services. RIPng uses both unicast and multicast mechanisms for inter router communication, according to the kind of message exchanged. The multicast address ff02::9 is reserved as the all-rip-routers group, which is used except in some non-multicast channels, where explicit network addresses have to be used.

Authentication mechanisms have better grounds on IPv6 protocol stack and thus, are removed from RIPng protocol itself.

2.1.1.2 TTCN-3 main components

TTCN-3 is a pretty new language (current TTCN-3 Core Language[ETS03] was published on 02-2003) with only a first generation of compliers and tools supporting it. TTCN-3 was designed to be able to incorporate testing capabilities not present on other programming languages, and was also cleared from OSI peculiarities (that previous versions suffered). TTCN-3 is designed to be flexible enough to be applied to any kind of reactive system tests.

An alternative representation of the layout of a TTCN-3 test system general structure is shown in figure 2.2. This figure is taken from earlier TTCN-3 standard versions. It does not emphasize the communication architecture among entities, as it is done in figure 1.3, but allows us to depict better different encoding subsystems. As usual, this

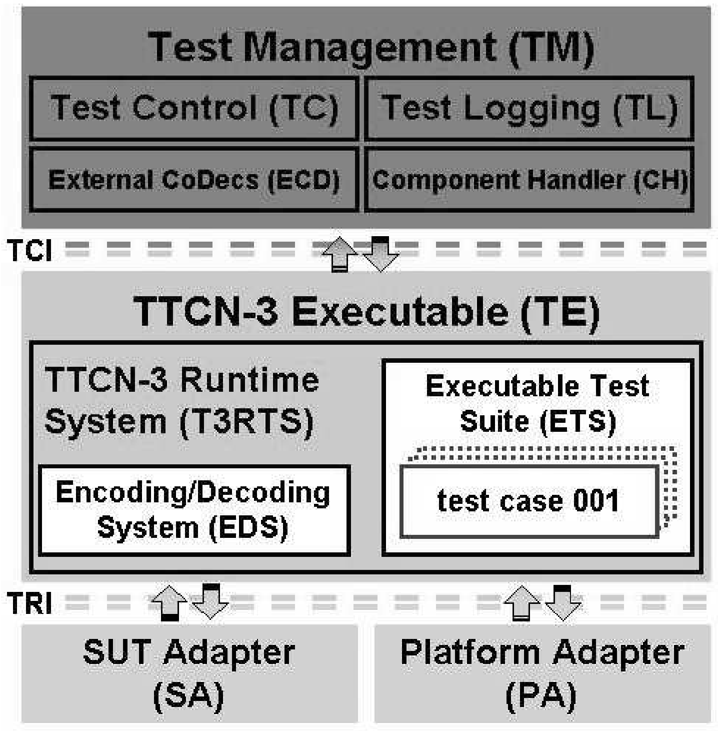


Figure 2.2: Alternative TTCN-3 Test System Architecture representation

test system is supposed to be executed against a system under test (SUT). Each block in the figure represents an entity implementing a particular aspect required by a test system. The test system user interacts with the Test Management (TM) and uses the general test execution management functionality. The TM entity is responsible for the global test management. The TTCN-3 Executable (TE) implements the functionality defined as TTCN-3 modules, which can be structured into sub-modules and import definitions from other modules. Modules have a definition part (which defines test components, communication ports, data types, constants, test data templates, etc.) and a control part (which is responsible for calling test cases and controlling their execution). Other test layout dependent parameters are defined at the SUT Adapter (SA) and the Platform Adapter (PA). A TTCN-3 test system has two main internal interfaces, the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI). TCI specifies the interface between Test Management (TM) and TTCN-3 Executable (TE) entities. TRI interface specifies the interfaces between TE, SUT Adapter (SA) and Platform Adapter (PA) entities. Note in figure 2.2 the presence of Encoding/Decoding System (EDS) as part of the Runtime System and the External CoDecs, behind the TCI interface. Both entities solve the problem of encoding messages, one in a tool-dependent way (the EDS) and the other, through a standardized API (the TCI-CD).

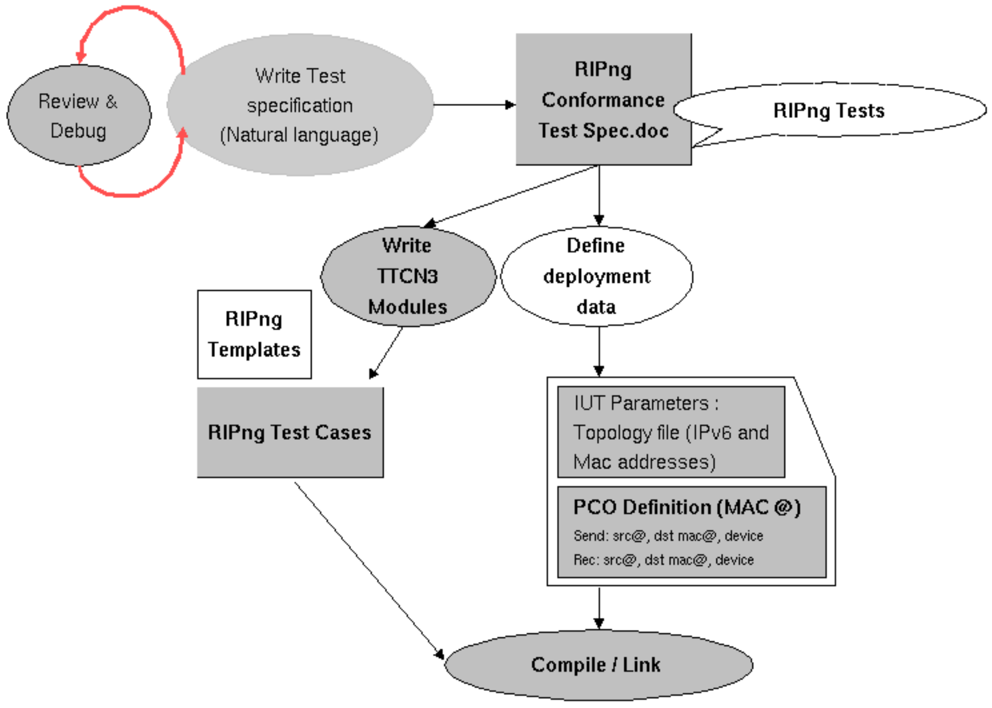

Review & Debug
Write Test specification (Natural language)
RIPng Conformance Test Spec.doc
RIPng Tests
Write TTCN3 Modules
Define deployment data
RIPng Templates
RIPng Test Cases
IUT Parameters : Topology file (IPv6 and Mac addresses)
PCO Definition (MAC @)
Send: src@, dst mac@, device
Rec: src@, dst mac@, device
Compile / Link

Figure 2.3: TTCN-3 based initial approach of test specification

Figure 2.3 shows the modules and main methodological tasks that have to be developed to produce test suites. The blocks named `RIPng Test Cases` and `RIPng Templates` correspond to the tasks required to define the TTCN-3 Executable block on figure 2.2. The blocks named `SUT Parameters` and `PCO Definition` correspond to parameters required by the SA to interface with the SUT.

2.1.2 The field experience

We have a broad experience on the IPv6 field, while these experiments were our first practical approach to TTCN-3. Nevertheless, both our experience and the methodology used in the IPv6 community matches the principles suggested in [WLY03]. The hands-on experience with TTCN-3 described tries to answer whether the language and methodology are ready for addressing the strong needs of the IPv6 test community. It is worth mentioning that the Internet community, for more than 20 years now, is a very pragmatic environment, who does not care about the way the tools are designed, but focus on the way they can quickly answer to their needs. Our goal was to develop tests for RIPng in a short time with existing new tools.

Conformance testing, based on a black-box approach did not allow us to use any particular knowledge of the IUT in order to test it. We had to exchange signals with the System Under Test (SUT): in this case, signals are RIPng messages. Designed test cases consisted of exchanging routing information with the SUT and later sending IP probes to selected destinations so as to determine the way routing information is not only learned and shared by the SUT, but also, applied on its own routing decisions. This is the general philosophy in the Ready Logo. All the things that are required must be implemented by all the implementations. Thus, when test cases are designed, the test expert must be aware that all requirements placed on the test are then placed on the IUT. Tests have to be done using the minimum set of capabilities demanded to all implementations, despite the fact that they are an embedded circuit, an IP camera, a router or a computer.

To be able to specify TTCN-3 test cases we had to obtain a tool and define the needed modules according to our test purposes. It was also required to provide the SUT Adapter (SA) with proper definitions so that the mapping between TTCN-3 components communication ports and test system interface ports is done. After this, the ETS was generated.

2.1.2.1 Approach for TTCN-3 test specification

Routing Table Entries (RTE) are the key elements exchanged within RIPng messages. Each router is supposed to have some sort of routing table with at least the following information: the IPv6 prefix of the destination, a metric, the IPv6 address of the next router along the path to that destination, a flag and various timers associated with the route. This suggests that basic routing operations being tested ought to be related to RTE maintenance like: RTE creation, RTE update, RTE deletion, RTE request.

The simplest test topology would consist of two routers and the SUT, each connected to a different physical interface of the SUT. The problem with this topology is that it does not allow us to perform the required message exchanges. From the test purposes settled we decided to build a more complex network layout, shown on figure 2.4. The small box in the center represents the role that the SUT plays in the topology, while the rest of it, marked as `Tester` represents what has to be developed to perform the tests. For specific test purposes we selected -projected- the relevant routers that would allow inspection of the desired property and specified the particular ATS only considering it. This methodology simplified test design because we had a single well known network, and it allowed us to concentrate on details of each test purpose by projection of relevant smaller parts of the network.

It was required right from the first test definitions to be able to emulate more than one router in order to explore even simple protocol behavior and properties. This fact made us define and handle several Points of Control and Observation PCO. The distribution of PCO over single or multiple test execution threads or processes promoted the discussion between parallel *vs.* single party testing, or in other words, a Master Test Component (MTC) with Parallel Test Components (PTC) *vs.* single MTC. Protocol complexity was not an issue at this point, as the protocol itself is simple: both solutions

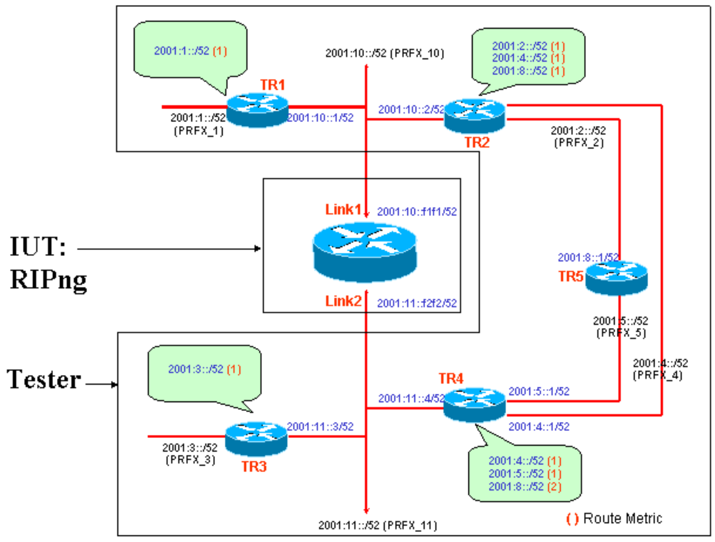


Figure 2.4: RIPng testing topology

were adequate for test requirements. From our previous experiences and the lack of time for enough testing of the TTCN-3 parallel possibilities and API, we decided for a solution with a single MTC that handled all required PCO. The decision of using a single node to emulate the whole network topology allowed us to avoid all parallel synchronization problems. This decision also considered easy deployment and testing re-usage: it is simpler to deploy a single device than a configuration with 6 nodes. We believe that naive deployment of PTC corresponding to each emulated router would have produced test suites with different characteristics. Complexity of test setup would have increased considerably as separate process on different machines had to be configured.

Another important decision was the tool selection, which was done considering all the existing tools known to us (testing_tech, Telelogic, Danet, OpenTTCN, etc.). At the time of the selection all available tools were equally eligible as they all implemented TTCN-3 required components. Moreover, none of them provided already built IPv6 libraries that might have helped with the building blocks for RIPng tests. The decision was based on our experience testing with C++ tools and licensing conditions that allowed us not only to use the tool for academic purposes, but also to have access to the source code when needed. Other aspects considered were Integrated Development Environments (IDE) and tools provided that helped with simple and repetitive tasks. From all those testing tools available we chose Danet's testing tool (`http://www.danet.de`).

2.1.2.2 PCOs management

Points of Control and Observation (PCO) play a very important role on what can be observed out of a system. Proper selection of PCO placement would allow better and detailed protocol inspection. As RIPng is a UDP based protocol the first test design tried to place PCO at UDP level, as shown in the figure 2.5. In TTCN-3, PCO are referred as ports.

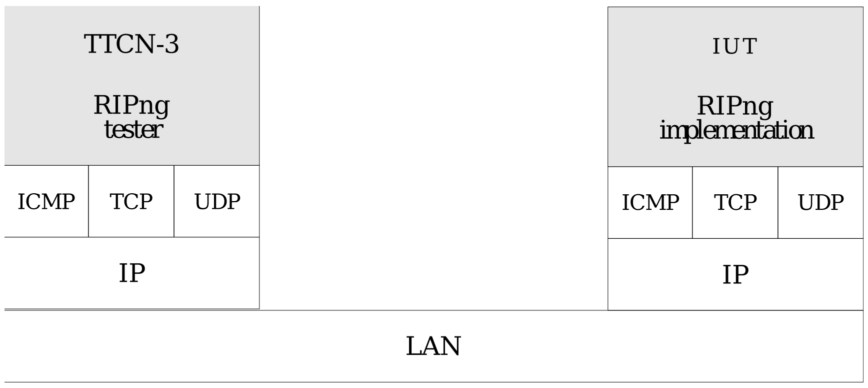


Figure 2.5: RIPng testing architecture, UDP level PCO

It was not possible to code a single tester that was capable of emulating several routers using the off-the-shelf TTCN-3 tool. The selected tool only implemented two types of ports: serial and socket. Serial did not apply for Ethernet communication, and socket was implemented using underlying operating system protocol stack services at socket level, thus it is not possible to simulate traffic to and from different routers: it would be necessary to define different IPv6 addresses and Ethernet MAC addresses. TTCN-3 definition is independent of this low level details. Thus, it does not allow dynamic definition of MAC/IPv6 addresses associated to ports on every implementation.

Another observation is that we did not only need UDP services: ICMP echos are sent through the SUT so as to check the routing decisions at a certain moment.

The figure 2.6 shows all the parts -grayed- of the protocol stack that had to be addressed with the test.

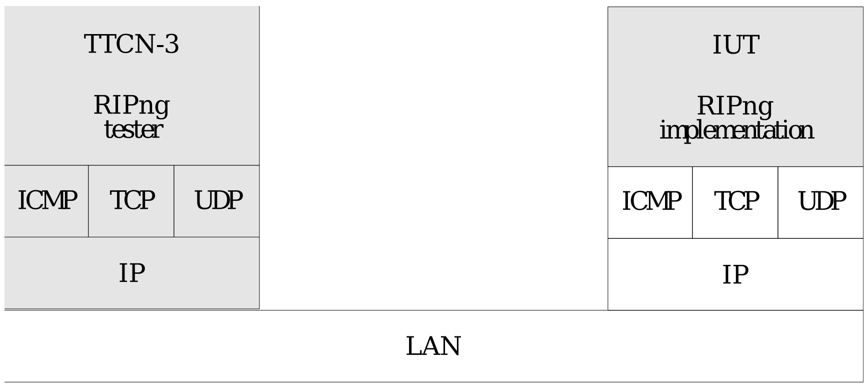


Figure 2.6: RIPng testing architecture, link layer/IP level PCO

IPv6 Ready logo conformance methodology tries to make simple the deployment of the solution, at expenses of a possible more complex test suite development. Instead of addressing parallel deployments, very specified behavior is generated from a single host, making the IUT believe that it is exchanging messages with as much other systems as required.

The main difficulty was that when more than one router had to be emulated using a single MTC, the IPv6 native stack on the host had to be disabled and all the steps of the communication had to be emulated from TTCN-3 modules. This is the way it is done in the Ready Logo by the TAHI tool. It would have been also the same situation with several PTC running on the same machine. Several issues arose during the development phase. Neither the TTCN-3 tool nor language were not designed to handle multiple host emulation using a single Network Interface Card (NIC). This problem is highly specific to IPv6 and is not easily found in a general purpose testing language and tool. Due to time constraints we worked out the problems by changing some aspects of the tool implementation by recoding parts of TTCN-3 primitives. We changed TRI provided implementation so as to handle link layer PCO, which were not implemented in Danet's tool. The main modification consisted in adding a new type of port that handled Ethernet communication, but at the physical interface level. With the modifications introduced we were able to emulate as many hosts -form data link layer up- as required from a single real host. The availability of the source code made our work more simple.

This kind of handling increased the complexity of the ATS as not only RIPng protocol communications had to be implemented. Required UDP assembly and disassembly of packets also was needed, including checksum and packet length calculation. IPv6 layer assembly and disassembly of packets was also mandatory. At the end also data link layer parameter handling had to be introduced to transmit packets with the corresponding MAC address of the router emulated. Moreover, the reception of the packets and their corresponding processing had to be handled.

Other link maintenance aspects of IPv6 Neighbor Discovery[NNS98] (ND) algorithm had to be addressed. IPv6 relies several host autoconfiguration tasks to the ND. Thus, for correct node emulation, ND signaling is necessary.

TTCN-3 template definition was not versatile enough to allow efficient matching of incoming data. Wildcard only matching mechanisms were not enough, and by that time, we did not address the development of external CoDecs. We only used internal ones, generically provided by Danet.

Based on those hypothesis, there was not much that we could do. The solution found was to create as many PCO as couples of communicating addresses required. We were able to match unique, low level information. Due to the way IPv6 handles addresses, each emulated node was associated to several addresses (unicast and multicast). To fulfill this multiple addressing scenario, several PCO were introduced. The complexity generated by this fact was significant, both at ATS coding and at tool modification level. ATS legibility was also an important issue as classification of messages received became complicated. Basically, every pair of origin/destination of addresses had to be matched.

2.1.2.3 Coding/decoding, libraries and low level data handling

The communication between RIPng nodes is message-oriented. Message definition has a low level of abstraction and coding/decoding is done dependent on the position of bits within the frame. Figure 2.7 presents RIPng packet as defined in the RFC 2080[MM97] with its corresponding IPv6 header prepended, without any IPv6 options.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |  Next Header  |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                         Source Address                        ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                      Destination Address                      ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source  Port         |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Length            |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  command      |  version      |       must be zero            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                Route Table Entry 1                            ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                Route Table Entry 2                            ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.7: RIPng packet format

Several codification issues needed to be solved in order to define a TTCN-3 module that abstracts the RIPng packet. First of all, some fields always took fixed values, like the protocol version, which is `'0110'B` for all IPv6 tests. Other fields must be changed during test execution, and were modeled as parameters of templates, like prefixes and prefix length values. Some fields were parameters of the component, like source and destination addresses (different from one tested router to other). Finally, others needed to be calculated each time a packet was about to be transmitted, like payload length and checksum values. As shown, there were requirements on packet by packet basis, destination, IUT and test suite dependent.

We found that there was no easy mechanism, like the ones defined on the RFC 2373[HD98], for IPv6 address text representation. No library or support for compact IPv6 address notation was provided. When defining parameters for a component, its IPv6 address `2001:2::1` had to be coded. In our environment, XML files were used (see figure 2.8).

```
<RUT_LINK2_GLOBAL_ADDRESS1  moduleId="IPv6RouterInterface\">
    <OctetStringValue valueKind="4\">
        20010002000000000000000000000001
    </OctetStringValue>
</RUT_LINK2_GLOBAL_ADDRESS1>
```

Figure 2.8: Markup defining an IPv6 address

To ease TTCN-3 based IPv6 test generation, a test environment should provide standardized methods for network address handling and representation.

TTCN-3 data type definitions were coded to provide abstract description of IPv6 packets. Templates are built based on data type definitions. Figure 2.9 shows an example of a template defined.

We modeled the protocol version field as a `bitstring` field of length four. We expected that integer values (like 6 for the protocol number) would be simply assigned, but they have to be converted to bitstrings. The solution found was to invoke an encoding function that encode the 6 in binary using four digits (`Version := int2bit (6, 4)`). Even though this was not particularly a problem, -there was a *work-around*- the solution does not seem natural. It is natural for a developer to expect that the language solves these conversions transparently, at least, providing default rules that can be overridden.

Another relevant limitation found was that we were not able to specify a template with "any number of RTE" (note the difference with a recursive type with any number of RTE). The template shown in figure 2.9 is defined for a RIPng packet with exactly two RTE. Pattern matching rules embedded in TTCN-3 might allow definition of repetitive parts of structures that might help decreasing the number of data types and templates defined. The problem indeed was not type definition, but matching incoming messages to the template. Using the default internal CoDecs it was not possible to solve this issue. We learned about external, user developed, CoDecs after we finished the development. This was one of the reasons that made us more conscious about the relevance of CoDecs and motivated the research on the consequences they have on the ATS and vice versa.

Upon message reception, the message classification presented several difficulties, both for handling interleaved reception of RIPng packets and ND ones. This fact conspired against legibility of the test. It is desirable to have some aggregation of "similar" packets. In this way, logical separation of message reception and handling

```
template IPv6PacketType RIPngRequestTable_tp
  (IPv6AddressType source, IPv6AddressType dest,
   IPv6AddressType P1, UInt8 PF1, IPv6AddressType P2, UInt8 PF2) :=
{
Ipv6Header := { Version := int2bit (6, 4),
                TrafficClass := 0,
                FlowLabel := int2bit (0, 20),
                PayloadLength := 0, // CALCULATED BEFORE SENDING
                NextHeader := NextHeaderUDP,
                HopLimit := 255,
                SourceAddress := source, // TEMPLATE PARAMETER
                DestinationAddress := dest // TEMPLATE PARAMETER
 }
Data :=  { UDPHeader := {
                         SourcePort := 777, // NEVERMIND
                         DestinationPort := 521, // SERVICE PORT
                         Length := 0, // CALCULATED BEFORE SENDING
                         Checksum := 0, // CALCULATED BEFORE SENDING
                         Payload := { Command := 1, // RIPng Request
                                      Version := 1,
                                      MustBeZero := 0,
                                      RTE := { // First RTE
                                               IPv6Prefix := P1,
                                               RouteTag := 0,
                                               PrefixLen := PF1,
                                               Metric := 0
                                             },{ // Second RTE
                                               IPv6Prefix := P2,
                                               RouteTag := 0,
                                               PrefixLen := PF2,
                                               Metric := 0
                                             }
                                     }
                        }
        }
}
```

Figure 2.9: TTCN-3 template for a RIPng packet

would lead to more structured ATS. Even though AltSteps are good for aggregating and factorizing reception of messages, we did not achieve legibility.

By that time, we wanted some kind of inspection of unknown packets to be provided from the ATS. We did want to use internal CoDecs in some way that we could interactively take decoding decisions from ATS, but it was not possible. Reception message queues are processed sequentially. Upon arrival of a non-matching packet, the reception queue stalls. A "wild-card" default packet matching rule was introduced, but TTCN-3 does not provide methods for inspecting the unknown packet. Reception of unmatched packets was logged and the analysis had to be done with external tools like `Ethereal` (`http://www.ethereal.com/`), something that was important during test debugging and log analysis. The ability of Ethereal (now Wireshark) to decode IPv6 packets was exploited later.

2.1.2.4 Test execution

From the methodological point of view we intended to perform stepwise refinements of our ATS until producing the definitive one. Spiral patterns or incremental iterations could not be performed in the way that they should. The amount of modules and things to be generated delayed the first ETS test production. The time elapsed until we had the first executable version of the test made that several different pieces of testing code had to be debugged at once. This produced a new delay in the feedback for refining the test suites.

The lack or building blocks prevented us from concentrating only on RIPng templates and test cases. Representation of network topology, like routing tables, was needed. The lack of IPv6 extensions or libraries also forced us to model from simple things, like IPv6 packets, to complex behavior like ND algorithms. We are aware that this was our first TTCN-3 implementation, but all the facts suggested that the *test development cycle* was too big and only few iterations could be performed. Lack of availability of IPv6 libraries for TTCN-3 refrained the community to adopt it. Test cases had to be developed from scratch. Even nowadays, there is no clear agreement in how to model IPv6 data in TTCN-3 and how to factorize complexity. A few (IRISA, ETSI, etc.) have developed and published IPv6 TTCN-3 test specifications, but it is very difficult just to reuse something without adopting the whole design philosophy.

The figure 2.10 shows the effective RIPng test development cycle and the main tasks needed for closing it. It is worth comparing our initial test development plan (see figure 2.3) with the actual work done. Our experience suggests that network layer support from the tool is needed to reduce the gap and, consequently, development overhead.

The tests performed in our laboratory were done against both a `GNU/Linux` system running `Zebra/RIPngd` and `FreeBSD` system running `routed6`. From the test development point of view, Danet's tool gave the required support for analyzing and debugging purposes. From the test execution point of view we found that log information was hard to analyze. One possible reason is that our changes at PCO were not propagated by the tool to the log files. Thus, Data Link Layer information was stripped from the

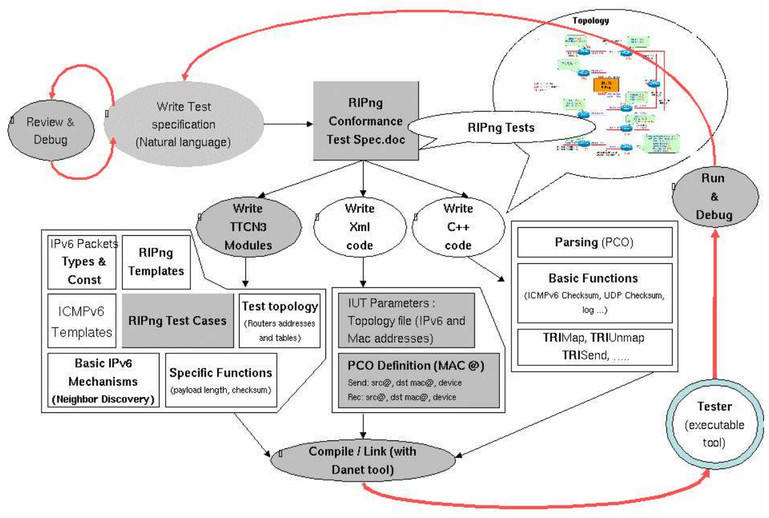

Topology
Review & Debug
Write Test specification (Natural language)
RIPng Conformance Test Spec.doc
RIPng Tests
Run & Debug
Write TTCN3 Modules
Write Xml code
Write C++ code
IPv6 Packets Types & Const
RIPng Templates
ICMPv6 Templates
RIPng Test Cases
Test topology (Routers addresses and tables)
Basic IPv6 Mechanisms (Neighbor Discovery)
Specific Functions (payload length, checksum)
IUT Parameters : Topology file (IPv6 and Mac addresses)
PCO Definition (MAC @) Send: src@, dst mac@, device Rec: src@, dst mac@, device
Parsing (PCO)
Basic Functions (ICMPv6 Checksum, UDP Checksum, log ...)
TRIMap, TRIUnmap TRISend, ...
Tester (executable tool)
Compile / Link (with Danet tool)

Figure 2.10: Test development cycle

packets and did not reach log files.

TTCN-3 language and the tool provided adequate support for issuing a verdict, but we found it difficult not only to explain it but to extract information that eases debugging. When testing for conformance, it is important to produce feedback that helps the product improve compliance. We found it difficult to analyze execution traces. They were useful for test suite debugging, but not for SUT conformance debugging.

Five test cases were developed in time for their presentation at PlugTests 2004 and the rest of the test cases were ready and run at the TAHI Interoperability event. Generated tests were successfully executed during PlugTests and the Interoperability event, in October 2004 and January 2005 respectively, with interesting results. But still, we found it not easy to use TTCN-3 tools compared to what we can do with `v6eval`.

2.1.3 Some lessons learned

The objective of the experience presented was to gain experience using TTCN-3 language and tools while addressing a real and pending conformance test problem. As stated before, one important reason behind the experience was to determine TTCN-3 maturity and its ability as a provider-independent tool and language for defining test suites. Even though we addressed portability of the ATS, we run into internal CoDec issues, which are not portable. Internal CoDec supplied by the tool are tool vendor dependent, non-standardized. To achieve portability we need more than just an ATS, but to provide the surrounding entities required to generate a full TTCN-3 test system.

We found that there are no standard extensions to handle IPv6 level data. It is also noticeable that there is no explicit support in either TTCN-3 or the tool for lower layer ports, which is required not only for IPv6 testing but for other Ethernet transported protocols. Moreover, there is no easy mechanism for standard IPv6 address representation. For modeling network layer protocols, the tester network stack has to be disabled. At that moment all IPv6 implementation details, including packet assembly/disassembly, ND became part of our test and had to be developed. It is desirable that an IPv6 oriented test tool provide as many tools as possible to the expert to help him concentrate on the test purpose. Even though we partially succeeded, our test suites relied on PCO behavior not defined in TTCN-3 standard language. Thus running the tests over an of-the-shelf TTCN-3 tool might be impossible. TTCN-3 code alone is not enough to fully specify a test system.

All our results indicate that it is not possible to provide standard TTCN-3 test suites for IPv6 protocols based on our test architecture built on multiple host emulation from a single test node. There is no standard requirement on TTCN-3 tools that support our needs. Experimental results suggest that the minor changes performed to the tools would benefit TTCN-3 usage (maybe an IPv6 specialized version of the tools). Field experience supports that the ability to emulate a complex network from a single host is beneficial from the point of view of test execution and is worth considering it as a requirement for the TTCN-3 language.

Addressing further aspects of the RIPng test suite requires usage of other IPv6 features not implemented in the tool and not easily developed. RIPng relies on the IP Authentication Header and IP Encapsulated Security Payload to ensure integrity, authentication and confidentiality of routing exchanges. IPv6 stacks must include IPSec support, used by RIPng, and we have to manually code IPSec from scratch in TTCN-3 for testing SUT security capabilities. Language features to encapsulate behavior and produce libraries are not available in TTCN-3 as they are on other programming languages. We found no easy way to achieve reusability and to scale complexity in a regular divide-and-conquer fashion.

It seems that TTCN-3 template definition alone was not versatile enough to allow efficient matching of incoming data. Built-in internal CoDec based solution is not portable. It might be interesting to have hierarchical incoming data matching or at least being able to group similar matching rules. This has a direct impact on ATS legibility as the number of entries in matching statements grew considerably. We think that the problems of expressiveness would remain even if we use several PTC instead of a single MTC. The experience of such implementation would help understanding other TTCN-3 aspects, while contrasting single tester *vs* parallel testers on the same matter.

We found no way to define recursive or iterative data templates. Repetitive structures (like routing tables) are sets of individual RTE. The definition of individual templates for packets with one, two, three, or more RTE again made the code difficult to maintain, unnecessarily large and hard to read. Repetitive tasks, like checksum calculation and verification might be eased if templates would accept dynamic definitions (like accepting functions in their definitions). Again, this would require a redefinition of the runtime architecture of the language and the language itself.

As a consequence of previous limitations, we were unable to find a pleasant methodology for test creation. It is difficult to abstract parts of the components and protocols for re-using in future implementations. It takes more time than expected to produce runnable ETS. This fact makes that feedback from real execution returns late in test development cycles and the risk of delay due to redesign need is high.

2.1.4 Main findings of our hands-on experience

We have presented the most important lessons we found when applying the young TTCN-3 language to produce test suites for RIPng protocol. We were able to meet the schedule and the resulting test suite was successfully presented at 50^{th} ETSI Plugtests event and at TAHI Interoperability event. We showed that it is possible to develop test suites using TTCN-3, but under special circumstances like having access to some parts of the tool source code and being able to change its implementation.

Success of the language is tightly related to the availability of tools and their capacity to cope in time with the requirements of different fields of application. A careful analysis of enhancement requests has to be combined with pushing industrial requirements. Widespread availability of tools would speed-up this process.

There are also pending issues regarding language constructs and style that would lead to readable ATS.

Several important decisions were taken without enough study and experimentation. The following section addresses detailed study of identified problems, specially of the relationship between the TTCN-3 code and portable coding/decoding functions.

2.2 TTCN-3 System Adaptor traffic capturing and analysis extensions

From our initial experiences with the TTCN-3 language, we found that TTCN-3 supports adequately verdict issuing. A powerful and adequate handling of verdicts is part of the language. Despite of this, in many scenarios it is required to provide more information than only the verdict. Additional information has to be documented, which provides enough confidence to the implementer and third parties that the verdict properly reflects the quality of the implementation. Additional information becomes even more relevant when the verdict is not `pass`: the implementer wants to know why the implementation is not conformant. It is arguable if the role of a testing laboratory is to provide *debugging* information or not, but it is clear that enough information has to be provided so as to support the verdict issued beyond any doubt.

The objective of this Section is to present some techniques and extensions that proved useful, both when debugging the test suites and when documenting a conformance statement. Section 2.2.1 presents current practices and TTCN-3 limitations which motivates this work. Afterward section 2.2.2 presents the solution developed. In section 2.2.3.1 the results are presented.

2.2.1 Addressed limitations

TTCN-3 architecture, as shown in 2.2 and 2.12 shows the conceptual layout and interfaces used by the different interacting entities. Refer to section 2.1.1.2 for a more detailed description of the different entities and interfaces. It can be seen that communication with the SUT is done through the TRI interface, while logging is done through the TCI interface. Despite of the level of abstraction, detailed message information that needs to be logged has to traverse these interfaces, forcing the ATS to be complex and detailed enough to handle all possible data.

Another limitation was that by the time of this work, standard logging functionality of TTCN-3 was limited to constant strings only. Even though all vendors provided their own extensions to support more powerful logging, they were not standard. Any solution selected threatened ATS portability, thus, we decided to address this limitation in a tool independent manner.A

Another limitation of the second version of TTCN-3 concerned logging capabilities. Logging extensions of TTCN-3 v2.2.1 do not specify a format for log files. Log file format is important for automatic analysis of test suite execution. Lack of standardization not only threatens portability amongst vendors. It makes test laboratories to review their test analysis routines whenever a new version of a tool becomes available, as changes in the log file format might happen.

Important enhancements in logging capabilities were made in TTCN-3 v3.1.1. Log file format was standardized into XML through the TCI-TL interface. This defines an adequate interface between TTCN-3 output and in-house analysis tools.

Neither version considers a standard way of coding a sort of severity for each of the log entries. Logging in a complex test suite can produce large amounts of entries, which might provide different levels of information or detail in different moments of time. Detail of logging might not be the same when debugging the test suite definition and when being run against a SUT. Even though this is not a strong requirement, is a tool that helps the test expert.

The field of application of our methodology is Internet Protocol testing and some de facto standards were to be preserved. The Ethereal [ETH06] tool (currently Wireshark) is the standard tool for traffic analysis, and the standard traffic capture format -pcap files- is required for storing test execution traces. The general way of obtaining traffic traces is to launch a traffic capture session as part of the preamble and finish it during the postamble. This practice introduces some problems for log analysis: TTCN-3 log and external traces have to be synchronized. Log synchronization might not be a problem under a single test node, but when the test activity gets distributed might lead to inconclusive verdicts.

2.2.2 SA extension

The System Adaptor (SA) extension developed was originally intended to provide an Ethernet layer port implementation. When tests were executed, traffic was captured and registered with a packet sniffer for off-line analysis. Several problems arose due to the lack of synchronization. This required a manual, tedious, error prone and lengthy intervention of the test expert. Combining test execution, traffic sniffing and logging resulted as the next logical step.

2.2.2.1 Traffic capture and store

The library `libpcap` was already being used, so the main addition done here was not only receiving the traffic, but storing it. The format selected for storage is the regular pcap format. This was a straightforward decision as it was not only part of the library that we have been using, but also the standard format used by the TAHI group for documenting packet exchange in their tests. In this way we are able to work following accepted Community's practices with TTCN-3.

Another important benefit of this solution is that we can add extra information, while still preserving all packet information and reuse analysis tools. It would not be the case of re-using a tool like ethereal for the analysis of the XML log file generated by TTCN-3 v3.1.1 (unless it would be compatible with the not yet standardized PDML format).

2.2.2.2 Log extensions

The pcap file format is extensible, and allows user-defined information to be stored. The extensions performed enabled the coding of test-suites that transfer most of the logging requirements into the dump files. A customized header, called TIPI[1] header, was prepended to standard packet format inside a pcap file for each packet or event stored. This solves the need for synchronization of separated data sources at analysis time.

The implemented solution also provides a single source of information for analysis, without the need for vendor specific logging extensions. This is not only an advantage in from the point of view of portability, but also considering information handling. It also offers the benefit of working over a well known, standard and stable data storage with several tools able to use it. In this way, analysis tools developed will remain valid in time even if tools evolve and change. Before data can be sent to the logging component of TTCN-3, it has to be accessed from inside the TTCN-3 language, converted into values of specific types and only after that, used to invoke logging primitives. This procedure requires that incoming bitstring message has to be decoded according to a certain decoding hypothesis and converted into a value. There is no certainty that the decoding of a non-conformant message enables the test specification to decode it as a conformant message. After the uncertainty of possible wrong decodings, logging from TTCN-3, through the standard interface, might be meaningless without the actual traffic capture.

The extension enables the joint logging of the expected packet and the actual received one. This fact enables the expert to analyze data in a familiar format using standard tools that handle pcap files like `ethereal` (`http://www.ethereal.com`). Moreover, it is also possible to log verdicts associated to response arrival into the pcap file, as will be presented in the following subsection.

As mentioned before, additional information can be logged inside the TIPI header that eases analysis task. The transmitted packet addresses are the physical ones used

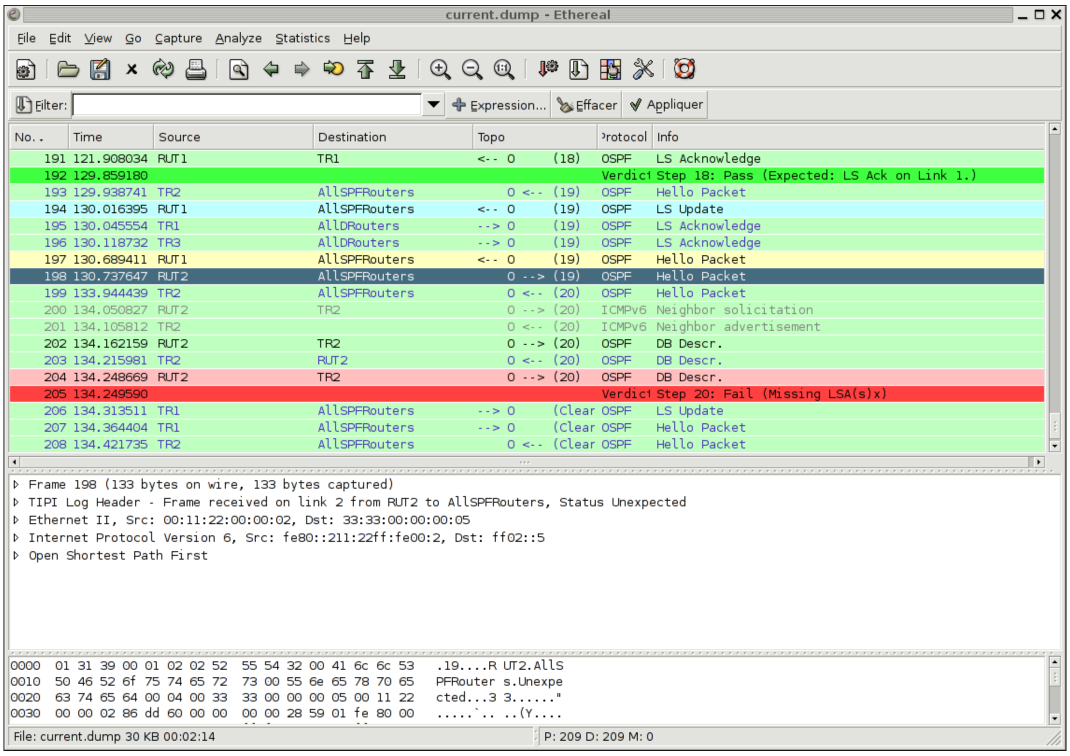
[1]honoring http://www.irisa.fr/tipi/

by the protocol, but in the TIPI header we can log the abstract names for source and destination. This gives higher level of abstraction to low level data and enables a broader group of experts to work on it. It is possible to associate the link in which the information was collected with the meaningful high level name according to test purposes. Another relevant type of meta-information that we include is the status of the packet. This informs us if it was expected or not according to the test specification. Status may also hold indication regarding debugging data, warning, etc.

2.2.3 Data analysis

In the same way that tools like `tcpdump` or `ethereal` are used for regular pcap file decoding and inspection, the corresponding tool for post-test analysis had to be developed. Architecture of `ethereal` dissectors and user friendliness of the tool were important reasons for selecting it as the base tool for analysis.

The work consisted of coding a dissector, implemented as an external plugin, capable of understanding the TIPI header, while reusing the regular dissectors for the Ethernet packets captured. TIPI extensions allowed coloring of packets according to their relevance in the testing, information carried and so on.



```
No. .  Time        Source  Destination    Topo          Protocol Info
   191 121.908034  RUT1    TR1            <-- 0    (18)  OSPF    LS Acknowledge
   192 129.859180                                        Verdict Step 18: Pass (Expected: LS Ack on Link 1.)
   193 129.938741  TR2     AllSPFRouters       0 <-- (19)  OSPF    Hello Packet
   194 130.016395  RUT1    AllSPFRouters  <-- 0    (19)  OSPF    LS Update
   195 130.045554  TR1     AllDRouters    --> 0    (19)  OSPF    LS Acknowledge
   196 130.118732  TR3     AllDRouters    --> 0    (19)  OSPF    LS Acknowledge
   197 130.689411  RUT1    AllSPFRouters  <-- 0    (19)  OSPF    Hello Packet
   198 130.737647  RUT2    AllSPFRouters       0 --> (19)  OSPF    Hello Packet
   199 133.944439  TR2     AllSPFRouters       0 <-- (20)  OSPF    Hello Packet
   200 134.050827  RUT2    TR2                 0 --> (20)  ICMPv6  Neighbor solicitation
   201 134.105812  TR2                         0 <-- (20)  ICMPv6  Neighbor advertisement
   202 134.162159  RUT2    TR2                 0 --> (20)  OSPF    DB Descr.
   203 134.215981  TR2     RUT2                0 <-- (20)  OSPF    DB Descr.
   204 134.248669  RUT2    TR2                 0 --> (20)  OSPF    DB Descr.
   205 134.249590                                        Verdict Step 20: Fail (Missing LSA(s)x)
   206 134.313511  TR1     AllSPFRouters  --> 0    (Clear OSPF    LS Update
   207 134.364404  TR1     AllSPFRouters  --> 0    (Clear OSPF    Hello Packet
   208 134.421735  TR2     AllSPFRouters       0 <-- (Clear OSPF    Hello Packet

▷ Frame 198 (133 bytes on wire, 133 bytes captured)
▷ TIPI Log Header - Frame received on link 2 from RUT2 to AllSPFRouters, Status Unexpected
▷ Ethernet II, Src: 00:11:22:00:00:02, Dst: 33:33:00:00:00:05
▷ Internet Protocol Version 6, Src: fe80::211:22ff:fe00:2, Dst: ff02::5
▷ Open Shortest Path First

0000  01 31 39 00 01 02 02 52  55 54 32 00 41 6c 6c 53   .19....R UT2.AllS
0010  50 46 52 6f 75 74 65 72  73 00 55 6e 65 78 70 65   PFRouter s.Unexpe
0020  63 74 65 64 00 04 00 33  33 00 00 00 05 00 11 22   cted...3 3......"
0030  00 00 02 86 dd 60 00 00  00 00 28 59 01 fe 80 00   .....`.. ..(Y....

File: current.dump 30 KB 00:02:14          P: 209 D: 209 M: 0
```

Figure 2.11: Analysis of extended dump file with Ethereal

Figure 2.11 shows the result of a enhanced pcap file being analyzed with the ethereal tool. A real section of an OSPFv3 test is shown. On the second line of the Packet List area of Ethereal's tool, numbered 192, it is shown a `pass` verdict after the right reception of an expected packet. Several lines below, in line number 205, a `fail` verdict is shown after a wrong message.

The Packet Details area of Ethereal's window is showing the contents of packet 198, which is an indication of an unexpected packet, but that does not alter the verdict according to the test codification. In this moment we can analyze it and determine that is a regular OSPF-Hello packet and validate the test. The information carried by the TIPI header lets us easily know that it was captured on the link number 2, and that it was sent from the router RUT2 to the rest of OSPF routers, without knowing Ethernet or IPv6 addresses of each router. The Packet Bytes area of Ethereal's window shows the contents of the packet byte-a-byte.

Extensions to `ethereal` were performed in such a way that the look and feel of the application was preserved.

2.2.3.1 Results

The implemented solution allowed us to address several different problems existing in TTCN-3 v2 and others still remaining in the TTCN-3 v3. The solution allowed to avoid problems due to limitations of generic CoDec, extend logging capabilities and also to facilitate test execution.

The generic CoDec, if provided with a tool, might not be suitable for matching and decoding properly complex protocols. During several stages of test development and execution it was required to compare log traces with packet capture manually. This task was eased as all information is jointly combined in a single dump file. This work does not address the inherent problem of CoDec lack of standardization, but provides means for easing log analysis and verdict issuing.

Test execution is simplified as there is no need to synchronize test execution with external data capturing: it is only required to deploy test components and execute the ETS. All execution information is condensed into a single self sufficient enhanced log + packet capture file. The log file format is based on well know standards and is stable, which validates long time efforts to produce automatic analysis tools. Only the developed plugin is required for standard ethereal tool to analyze the log files, even without recompilation of the tool. It reuses powerful existing packet analyzing tools graphic and dissecting capabilities. Some of this problems are addressed in the TCI-TL extensions of TTCN-3 v3 definitions, but tools for analysis of those logs have to be developed from scratch.

Finally, meta-information support, like status of stored information is added. This feature, not yet considered in TTCN-3 standards, is very useful for test analysis as it eases the work of the test expert. It provides means for classification of information according to relevance, coloring of logs and a way of combining high level of abstraction information with low level, bit oriented one. Even though it might be possible to use TCI-TL XML extensions of TTCN-3 v3 to provide the required meta-information, standardization on the output format should be addressed to avoid incompatibility of result analysis.

2.3 Search for ATS portability and ETS derivation ease

TTCN-3 is a strong-typed language which presents some difficulties to the test developer when trying to work with complex, low level oriented data. Network protocols present several hard to predict behaviors related to flow flags, options and other aspects that require the ability to handle unknown sizes, number of options, etc. TTCN-3 language provides basic matching capability based on wildcards like ? and *. Portability of the test specification cannot be achieved based only on the behavior expressed by the TTCN-3 ATS. There is a tight relation between these matching capabilities aspects and non-standardized tool extensions or with user provided coding/decoding functions.

The way that TTCN-3 is designed to address the decoupling of abstract and executable operations is by means of concern separation implemented through standardized software API. Coding of TTCN-3 values into transmittable messages and decoding of bitstrings into their TTCN-3 representation has been removed from the core language itself and is now done in an external component. These operations of coding and decoding are relayed to a specialized component named CoDec. There are internal CoDecs, which are not standard, and tool provided. To reach portability, it is possible to specify external CoDecs, which are developed together with the TTCN-3 ATS, but are coded in C++ or Java language.

The CoDec is interfaced with the TTCN-3 Executable (TE) through the Test Control Interface-Coding Decoding (TCI-CD) interface. As CoDec are not standard in TTCN-3, required ones *might* be present or not in tools. This is due to the fact that even though the TCI-CD interface is standard, its presence is not mandatory. In the field of Internet Protocol testing, tool-provided generic CoDec were not as flexible as we would have liked. TTCN-3 allows the possibility of implementing new CoDec, specific to the communication problem being addressed. They have to be coded in a "lower level" programming language like Java or C++, platform languages. The concept underneath the word platform shows that the choice between C++ and Java is not only a matter of taste, but it defines the support that you may get from the environment selected.

There is also different availability of libraries and tools for each of those languages. The design of a test system architecture that separates implementation details from the test definition itself helps achieving a high level of abstraction in the test definition language. On the other hand, it imposes additional complexity to the test development process: handling of communicable types has to be done both in the abstract TTCN-3 specification and in the platform language specialized CoDec. Every time the low-level types are reviewed, pieces of highly coupled yet independent code developed in different languages have to be altered and kept synchronized manually.

This Section presents and discusses the experiences gathered producing test suites. As stated before, the field of application was the IP next generation (IPv6 protocol suite). Coding decision options and how we solved the puzzle to allow reusability and maintainability of test suites are presented too. During this process, a framework for the testing of IPv6 based protocols was defined and implemented. This framework, named *IPv6 Testing Toolkit*, provides enough flexibility for software reuse with minimal or no code modifications. Existing works like [VGSB$^+$99, WLY03] were considered during the development.

The section is organized as follows. In Section 2.3.2 we present intermediate works that addressed the problem from a different approach and help motivating current framework. Section 2.3.3 summarizes the main problems addressed, puts together the experience gathered through previous experiments and points out the decisions that lead to the development of the toolkit. Afterward, in Section 2.3.4 the CoDec Generator is introduced. The toolkit with a few examples is shown in 2.3.5. The work is summarized in Section 2.3.6 and where we present the main contributions done.

2.3.1 TTCN-3 and low level communication

In this section we will introduce further TTCN-3 concepts that are relevant to this section. Parts of the information presented here were introduced before in 1.3 and in 2.1.1.2. Some of the information presented here is part of the background of a TTCN-3 expert, we include it because there is not much documentation about it. The implicit message here is not presented in TTCN-3 tutorials we have seen. Standard specification of the language specifies the Interfaces and API, but does not explain the underlying reasons for the design of the language. Understanding how to take profit of CoDec and platform language presents a difficult, undocumented and obscure challenge to any team trying to gain experience on the usage of TTCN-3 language. This knowledge is hard to acquire, belongs to groups working on the field and is not as widely published as a beginner would like. Hence, we consider it a contribution to the community.

2.3.1.1 Architecture of an Executable Test System

Before going into details, let's just review some TTCN-3 concepts. According to [ETS05c], a TTCN-3 test system can be thought conceptually as a set of interacting entities, each one implementing a specific test functionality. Figure 2.12 shows the general structure of a TTCN-3 test system. We will focus on the main concepts addressed by this work.

The TTCN-3 Executable (TE) interprets and executes TTCN-3 modules. The Test Logging (TL) entity performs test event logging and presentation to the Test System User. SA, which stands for SUT Adaptor (System Under Test Adaptor), "adapts" communications between the TTCN-3 system and the SUT. The Platform Adaptor (PA) implements external functions and provides a TTCN-3 system with a single notion of time. TE can be distributed among several test devices. The Component Handling (CH) implements communication between distributed test system entities. The Test Management (TM) entity is responsible for overall management of a test system. Finally, the Coding and Decoding (CD) entity is responsible for the encoding and decoding of TTCN-3 values into bitstrings suitable to be sent to the SUT. All these definitions can be found in [ETS05a, ETS05b, ETS05c].

TTCN-3 standards do not define some other concepts associated to the TTCN-3

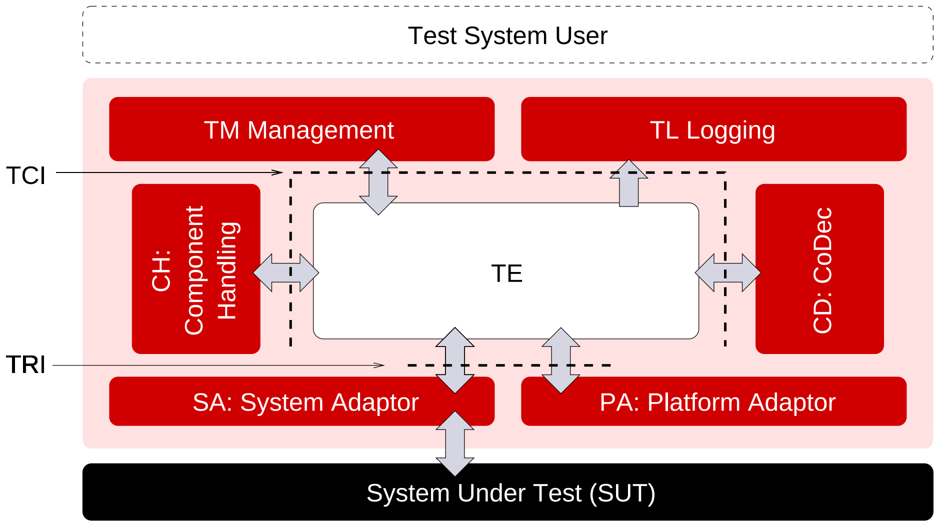


Figure 2.12: Conceptual architecture of TTCN-3

world, which are required here. Due to the fact that our approach is the application of TTCN-3 technology in a changing research environment, we are concerned with implementation and field problems too. Some of those problems are not addressed by TTCN-3 standards. Tool vendors provide their solutions, standard extensions, which lead to non-portable test case specifications.

2.3.1.2 Communication with the SUT

One of the key ideas behind the use of TTCN-3 language is to separate abstraction of the test specifications from execution details. Gaining abstraction in the test specification allows test specification reuse, makes test cases easier to understand and maintain, while being powerful enough to handle all the details and complexity required for addressing almost any kind of testing activity. But the *dirty* work cannot be avoided, and has to be done somewhere. Abstract definition of the test case has to be augmented until an executable form is reached. Despite the abstraction of the test specification, the gap has to be bridged until we can execute a test case.

As an example, an abstract `send` operation has to be converted into an operating system call that writes some information on a wire, maybe at the level of a serial interface or a network socket. Sending information is the abstract operation and writing to a socket is the executable operation. This mapping is unavoidable and has to be provided as part of the Test Specification. TTCN-3 language addresses this problem with two Application Programmer Interfaces (API): Test Runtime Interface (TRI) and Test Control Interface (TCI). These interfaces allow the specification of low level details, required for the actual test execution.

The TTCN-3 standard provides two definitions of each API, one using Java, and an equivalent one in C/C++. Strictly speaking, there is only one definition for the API and two mappings into C and Java language. Even though the TTCN-3 standard does not provide any restriction, and a TTCN-3 tool might implement both C/C++ and Java API, the state of the art is that tool vendors either specialize in Java or in C/C++.

Data representation in TTCN-3 is very similar to the Abstract Syntax Notation (ASN). Some protocol standards define the message construction using this syntax, and thus can be natively supported by the TTCN-3 tool provided that standard encoding methods like Basic Encoding Rules (BER) are used. Other protocols, many from the Internet Engineering Task Force (IETF) do not describe the message construction in ASN notation, and might use ambiguous notation as English language. The task of converting abstract types and values into transmittable messages is removed from TTCN-3 language and relied to external components.

Components, or interacting entities, are distributed and communicated through the TRI and TCI interfaces. As an example, a send operation in TTCN-3 language, implicitly uses non TTCN-3 codes through TCI and TRI. Through the TCI, the high level representation of a message is converted into a transmittable message, represented as a bitstring. The bitstring is then relayed through the TRI to the component that is ultimately responsible of the transmission. These components and others are coordinated from the TTCN-3 Executable (TE) implicitly during the execution.

We focus our analysis mainly on coding and decoding operations. These are relayed through the TCI to a specialized component named CoDec. The CoDec is interfaced with the TTCN-3 Executable (TE) through the Test Control Interface-Coding Decoding (TCI-CD) interface, defined in [ETS05c].

The way the complexity and the work are distributed is elegant and powerful, but we found that the cost associated to it is not negligent. Two different programs, coded in different languages, developed and maintained by two different development groups have to be synchronized through the TCI-CD interface. During the first stages of the test design and development, the TCI-CD offers an initial milestone and synchronization for the two parts. After the first version of the test suite is complete, every change performed that affects the data representation, level of abstraction of the messages or design of the test, affects both parts. The tight coupling of TTCN-3 data types and their corresponding CoDec generates this problem. This fact affects the whole lifecycle of the test. Elegant and simple solutions on one side of the TCI-CD might force obscure and difficult coding on the other side of the TCI-CD.

2.3.2 Highlights of intermediate solutions

With the goal of producing an adequate framework for IPv6 protocol testing, several different approaches for test design and development were carried out. Some of them are worth to be described, some others are not.

During the RIPng experiments we learned that coding and decoding issues were highly relevant for portable and efficient test specifications. The problems ranged from IPv6 network addresses manipulation capabilities to reception of unexpected -but still valid and conformant- data. State-of-the-art IPv6 testing requirements do not impose highly complicated signaling patterns, but very detailed composition of messages and a bitwise inspection of incoming messages. TTCN-3 language design addresses mainly the high level part of the testing problem, relaying the low level, bit oriented work to specialized pieces of code not specified in TTCN-3.

2.3.2.1 Initial Problems

The test development process induced by TTCN-3 mainly splits the work in two different development tasks. The first one directly related to the definition of the message sequencing and exchange, related to the abstract idea of test execution. The second one addresses crafting and bitwise coding and decoding of exchanged messages. These two different tasks, even though tightly related, are addressed by different experts with deep IPv6 skills, using different languages: the first task requires TTCN-3 specification, while the second one, platform language codification, in our case C++.

These two different tasks are combined through the TCI-CD interface, a TTCN-3 specialized interface for the decoupling of abstract and low level message handling. Its goal is to permit the data exchange between the TTCN-3 data structures and the platform language, which is in charge of performing low level or specialized tasks. To avoid obscure data manipulation practices, we decided to completely map TTCN-3 types into platform language ones and vice versa. In this way, we would share a common modeling of the communication messages and objects both in the platform language and in the TTCN-3 test specification. Low level manipulation would be done in the C++ view of the data, while test related decisions would be taken on the high level model done in TTCN-3 types. The link between these two representations is given by the encode and decode operations of the TCI interface, which were developed too.

After our first complete implementation of a test suite following this approach, published in [SBFV05], we realized that this process (named CoDec development from here on) is tedious and error-prone. Whenever there is a C++ or TTCN-3 requirement that forces some change in the type definition, the counterpart also has to be corrected accordingly. Differences in the expressiveness of the type definition structures of both languages induces non transparent data transformation procedures.

Additionally, IPv6 is not just a protocol, but a protocol suite. It forces us to handle different, simultaneous, not always related IPv6 message exchanges. This forces us to be able to handle all possible incoming IPv6 messages, even though we are interested in testing some specific behavior. Transmission is not a problem, as we are in control of which messages to transmit.

2.3.2.2 Reusing an existing tool: Ethereal

Trying to minimize the complexity of the CoDec development, the decision taken was to make an attempt to use an existing tool for solving the decodification of incoming packets. It is important to note that the main problem arises upon packet reception, where we need a good deal of flexibility. Conformant packets might arrive in different orders and with several variations in their codification, we should be able to accept all possible conformant combinations while rejecting all non conformant ones. Transmission is simpler, as we control what we want to transmit, how and when.

The goal is to avoid the complexity of manually decoding an arbitrary incoming IPv6 packet and having the task done by the Ethereal tool [ETH06]. Ethereal is a well known tool extensively used in the IP world. It provides several benefits like: it is well known, it evolves with new protocols, it is maintained, it is community-validated and it is free. Considering all these benefits, we decided to perform the low level decoding of incoming messages using Ethereal and interface it with TTCN-3 through the TCI-CD interface.

Several transformation formats and tasks were required to interface Ethereal with TTCN-3. Data received through the System Adaptor reaches the TCI-CD interface in a TTCN-3 format. Even though it is the bitstring representation of the packet, it is received as a TTCN-3 string. It has to be transformed into something understandable by Ethereal. Then, Ethereal's output has to be parsed and used to assemble the TTCN-3 objects that will hold the received message. Amongst the different Ethereal output formats, PDML (Packet Details Markup Language) [PDM06] format was chosen, an XML representation of network packets. The library `libxml` was used to parse the PDML description of the packet and have access to all of its parts.

Even though it was possible to reuse the tool and avoid the complexity of packet parsing, interfacing Ethereal is not a minor task. Most of all, not all problems were solved using Ethereal.

Problems due to the use of Ethereal
Even though Ethereal tool greatly solved the problem of parsing incoming messages, it does only provide that. Message transmission was done independently from Ethereal, by a generic function written from scratch. This solution lacks of a symmetrical treatment for transmission and reception operations. The code used for sending data is different from the one used for receiving. All changes have to be replicated in different places that implement different API.

Moreover, PDML was neither standard nor stable. Every time a new version of Ethereal is released, we had to verify that our PDML parser was still valid, thus the mapping had to be reviewed every time Ethereal is updated. Ethereal also did not decode parts of the packets which are important for our purposes (i.e. content of padding fields), thus, it was necessary to patch Ethereal to meet our requirements. The amount of C++ code to maintain did not shrink significantly, and the solution become more complicated for deploying, as an additional external dependency was added. The complexity was shifted from packet parsing into interfacing and data transformation, but still a simple and elegant solution was missing.

It was good to see that we could reuse existing and good tools together with TTCN-3. Despite that fact, the sum of all these problems suggested us to abandon this approach.

2.3.3 Summary of main addressed problems

This section summarizes technological needs and problems faced before the development of the CoDec Generator. It shows relevant problems and solutions found, our understanding of what is required from a testing tool and the issues that motivated our decisions.

It is worth mentioning that the problems introduced by the use of Ethereal, are neither due to Ethereal itself nor the reuse of existing tools, but to the fact that we had to interface things not meant to be interfaced together. Not all the different usages of Open/Free solutions were discarded as happened with the previous example. Functional extensions were successfully added to Ethernet ports using the libraries `libpcap` [LIB06a] and `libnet` [LIB06b], as presented in [SBV06]. These extensions are now included in all our tools. What was addressed and solved are some specific requirements of IPv6 protocol testing, but they could not solve the main issues regarding an adequate protocol testing framework. The main problems that were faced and had to be addressed are presented in this Section.

2.3.3.1 CoDec specific problems

As stated before in Section 2.3.2, CoDec development, integration and maintenance represent the main issue for us using TTCN-3 for IPv6 testing. When test suite descriptions are developed while the protocol specification is being developed too, it is very important that the test specification language is easily maintainable. When changes are produced in the test suite specification, the test system must be modified accordingly. When an updated specification of the protocol gets approved, the test system must be adjusted to meet it. These activities impose additional constraints on the test environment.

Some problems were found on TTCN-3 regarding test case maintenance. This is due to the fact that manual synchronization of types has to be done in two languages: platform and TTCN-3 languages. When changes in the test specification are required, the TTCN-3 ATS needs to be adjusted, and consequently, related parts in the CoDec have to be adjusted too. Probably some change in the updated standard of the protocol requires a change in the codification of messages that update the CoDec and TTCN-3 code have to be adjusted too. Moreover, there is a group of operations whose natural place is not the TTCN-3 ATS. For example, operations like checksum and length calculation can be seen more like a transmission problem than a test logic problem, thus the coding process is a natural place for performing these operations. When we are not testing the checksum algorithm itself, it becomes a transmission problem. If we specify it in the ATS, we lose abstraction. We would like to integrate these operations to the test development process in a more automated way.

Another aspect that becomes clear after working with CoDec implementation is that the TCI-CD is only an API designed for data exchange, not for data manipulation. Standard TTCN-3 data manipulation from the platform language is required for easy and efficient CoDec development. We realized that there are no standard libraries for manipulation of TTCN-3 data in an intuitive, efficient and uniform way across the different types. Standard operations in languages like C++ (i.e. casting, type conversion, etc.) cannot be performed in TTCN-3 in a simple and type independent way. To ease CoDec generation we find it necessary to be able to develop a library that provides a value-type handling similar to the one used in the platform language.

At a certain moment in time, we were involved in three different groups coding different IPv6 test suites for independent protocols. We faced severe problems for IPv6 core protocol type definitions due to the fact that there is not a methodology or tool support for separating test specific issues from standard (library-like) routines or processes. Handling new protocols implied defining new data types functions and modifying the CoDec accordingly. This led us to maintain three different CoDecs that shared most of their sources.

Several other small factors also accounted, but we want to point out as a last relevant problem that our near-future requirements imposed us the need of a strong workload on the CoDec side. Encryption and Security handling can be seen as further layers of encapsulation of message encoding operations. This requires that good software engineering practices are applied to all the software development process. After our abstract specification tool becomes a programming language, software engineering practices can be taken from granted. TTCN-3 does not provide means for adequate handling of these operations and completely manual implementation of all operations would become not feasible, or at least, extremely complex.

2.3.3.2 Empirical observations

Apart of previously mentioned problems, our TTCN-3 experience also showed empirical facts that oriented us in subsequent decisions. The first one, that is almost evident, is that there is a high level of redundancy between TTCN-3 and C++ code. Due to the fact that the development process applied started from the abstract side, it is the C++ code that repeats TTCN-3 structures. The C++ code which implements the CoDec is only a mean for representation conversion between physical messages and TTCN-3 data types.

Other relevant observation performed is that TTCN-3 type definition already holds most of the information required for coding and decoding. Most of the platform language code mainly repeats TTCN-3 code. The addition is a few metadata information (like type length for some data types) and specific algorithms for coding/decoding particular fields in non-standard ways. Also precedence in coding/decoding operations has to be specified, as calculating the length field and afterward the checksum is not the same than the reverse order.

2.3.3.3 Approach followed

The objective was to simplify test suite development process by minimizing CoDec development and maintenance work. This can be achieved by separating all that can be automated from what really has to be provided (because cannot be expressed in TTCN-3 language). This separation can be done by extending TTCN-3 with the addition of the missing logic, dependencies and semantic which are not present in the standard language.

The tool that automatically performs these tasks will be referred as *the CoDec generator* from now on and will be the subject of Section 2.3.4. The initial cost of development of the CoDec Generator is higher than simply developing a CoDec for a single test suite. The main advantage of the CoDec Generator is that it can be reused through different tests. In this way, the test dependent part of the CoDec remains independent from the CoDec generator. The CoDec generator becomes a part of our test platform and test development process. Only the platform language code and logic added to the TTCN-3 abstract specification is part of the test suite. The pieces of platform language code which serve as input for the CoDec Generator will be referred as *codets*.

2.3.4 The CoDec Generator

The CoDec Generator is a generic tool that fully automates the task of CoDec development. It takes the TTCN-3 code and *codets* (additional logic developed in the platform language) and produces a CoDec that implements the TCI-CD interface, providing the required coding and decoding facilities. Even though it was developed while addressing IPv6 protocol testing, the CoDec generator was carefully designed and developed as an "universal" tool, and can be used for CoDec generation in any testing domain. The only bias introduced by our IPv6 requirements is the order in which features were developed and that the platform language for which it is currently implemented is C++.

The underlying idea behind CoDec Generator was already presented as our work methodology during previous sections: each TTCN-3 type is mapped to a platform language object, standardized automatic conversion is provided and customized conversion means are provided using codets. The idea of having different levels of abstraction, and thus, different data models in the CoDec and in the TTCN-3 abstract test specification was discarded as it would always require a very specialized CoDec. In such case, the CoDec expert and the TTCN-3 expert would have different views of the problem and would not even share a common data model of the problem.

The mapping implemented by the CoDec Generator is not performed directly to platform language objects, but to a hierarchy of objects designed to provide a comfortable framework for data handling inside the CoDec. We will return to this point in 2.3.4.2. The rest of this section provides a quick glimpse of the CoDec Generator.

2.3.4.1 Architecture

The CoDec Generator implements a TTCN-3 parser, built using bison [BIS06] / flex [FLE06] GNU Open/Free tools. It has been ported and tested in GNU environments both in Linux and Windows systems. The implemented parser is responsible of extracting basic type information and structure from standard TTCN-3 code. Even though only type information is strictly required for CoDec Generation, a parser that accepts the complete TTCN-3 language was developed, which is another spin-off of this work. Type information is further augmented (as shown in 2.3.4.3) with codets that perform specific

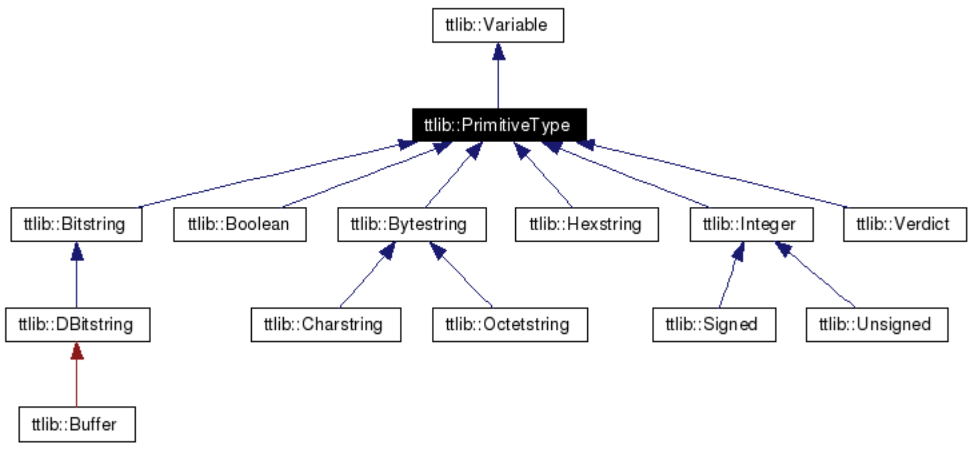


Figure 2.13: Platform basic type hierarchy.

operations between the TTCN-3 object and the low level, platform language managed codification.

The way in which the CoDec Generator is integrated into the native TTCN-3 framework is straightforward and simple. As it does not produce changes into the TTCN-3 code, no particular care has to be taken while developing the TTCN-3 ATS. The CoDec Generator should be invoked prior to TTCN-3 link-edition phase, so the actual CoDec is generated for the ETS. The CoDec Generator may produce platform language sources, objects or libraries, according to the TTCN-3 tool requirements. Depending on the options provided by TTCN-3 tool, it can be included into user defined link edition commands and invoked transparently from the tool environment.

2.3.4.2 Interfacing the CoDec Generator with the TCI and TRI

Apart from the CoDec generator, other tools that offer required functionality and services to the solution are provided. The T3DevKit provides also a library that implements and exports basically functions for data type management and data coding and decoding. The library provides adequate definitions that allow the mapping of TTCN-3 types and data structures into special platform objects. Platform objects were developed using platform language (C++ or Java) Object-Oriented properties (inheritance, polymorphism, etc.) so as to allow homogeneous and simple access to all types. The figure 2.13 shows the class-inheritance diagram for the objects that map TTCN-3 primitive types.

This ensures a minimum interface (set of member methods) available for all the objects. All objects implement their own `Encode()` and `Decode()` methods, main operations required for a CoDec. Methods like `GetValueHexa()` and `SetValueHexa()` are intended to provide a uniform handling of the value, regardless the object itself. The implementation would be subclass dependent as the semantic might differ from a

```
type union ICMPv6OptionSingleType {
    SLLOptionType                     SLLOpt,
    TLLOptionType                     TLLOpt,
    RedirectHdOptionType              RedirectOpt,
    MTUOptionType                     MTUOpt,
    PrefixOptionType                  PrefixOpt,
    ICMPv6UndefinedOptionType         UndefinedOpt,
    AdvertisementIntervalOptionType   AdvertisementInterval,
    HomeAgentInformationOptionType    HomeAgentInformationOpt
}
```

Figure 2.14: TTCN-3 type definition for a ICMPv6 options field

`Charstring` to an `Integer`, but a uniform way of accessing primitive types is provided.

All variables, specializations of class `t3devlib::Variable`, provide a method `Dump` that allows textual representation of the instance value. This method receives as a parameter an output stream and is intended to provide an aid for CoDec debugging. Providing a comprehensive guide to the library is beyond the scope of this section. The complete tool, comprising the CoDec generator, libraries, examples and documentation are published on-line as a gforge project in `http://t3devkit.gforge.inria.fr/`, currently maintained by Anthony Baire.

2.3.4.3 Platform language code extensions: CoDet

In the general case, automatic TTCN-3 type conversion into platform objects and back is not feasible for complex protocols. Many protocols not only handle unknown size payloads, but inclusion of unknown options, making it difficult to handle simple type matching and standard codification rules. To help the (de)coding process, the CoDec Generator accepts codets that perform specialized handling. This allows the test developer to separate (de)coding logic from test logic and also to place logic that naturally belongs to (de)coding process in the CoDec. This approach follows the same design principle as TTCN-3, but addresses relevant software engineering aspects. If no additional input is provided, the CoDec Generator will produce -if possible- a CoDec that directly maps TTCN-3 types into bitstrings and vice versa.

Different options of "logic extensions" were considered during the design of this version of the CoDec Generator. Probably the most appealing ones were those that extended TTCN-3 language. We faced problems: access to a compiler was required so as to modify it and implement our extensions; our ATS would be non-portable unless ETSI accepts our proposal and modifies the standards. Another option considered was to perform the extensions inside comment blocks. In this way our ATS would still be specified in standard TTCN-3 language, but parsing becomes more complex and non standard. The implemented option considers independent files for both TTCN-3 code and the extensions.

Figure 2.14 shows the TTCN-3 type definition for the option field of ICMPv6 and Figure 2.15 shows the codet to be executed prior to the actual decoding of the field,

```
inline void ICMPv6OptionSingleType::PreDecode (Buffer& buffer)
  throw (DecodeError) {
       UInt8 type;
       int position = buffer.GetPosition();
       buffer.Read (type, 8);
       buffer.SetPosition (position);
       SetHypChosenId (map_icmpv6_opttype.GetValue(type));
  }
```

Figure 2.15: Codet for determining the option type for `ICMPv6OptionSingleType`

specifically, for guessing the type of the option field. We can see that the matching is done based on TTCN-3 type names and predefined member names. These member names correspond to the TTCN-3 type that is being coded and the moment that the operation is to be performed. We will refer to these possible entry points as *codec hooks*. Possible codec hooks for decoding are: `PreDecode`, `PreDecodeField`, `PostDecodeField` and `PostDecode`. The CoDec generator will replace standard handling for the customized one, according to the definitions provided, if present. The symmetric processing is applied during coding time, and the possible codec hooks are: `PreEncode`, `PreEncodeField`, `PostEncodeField` and `PostEncode`.

It can be seen on the function definition at figure 2.15 that the CoDec Generator also provides a framework for handling `DecodeError` exception issuing. `EncodeError` exceptions are handled too.

2.3.4.4 Relevance of automatic CoDec generation

The CoDec Generator is a generic tool that automates the CoDec development task. It is based on a parser that extracts required and available logic already present in standard TTCN-3 ATS and complements it with codets, pieces of platform language code, to build the effective CoDec. As it extracts most of type information from TTCN-3 ATS, the task of repeating TTCN-3 type structure in the platform language is done automatically, removing the error-prone task of type structure synchronization from the test developer. Only codets need to be maintained. The amount of test-specific code becomes smaller, making it easier to maintain and evolve. This impacts directly on the development and maintenance costs of the test lifecycle.

2.3.5 The IPv6 Testing Toolkit

The *"IPv6 Testing Toolkit"*, `http://t3devkit.gforge.inria.fr/`, is a tool developed inside the research team that implements described solutions. We took part in its development team, obtaining the registry IDDN.FR.001.030006.001.S.A.2006.000.10600 by the Agence pour la Protection des Programmes. It is a set of data, functions and basic mechanisms dedicated to TTCN-3 test development and execution. It is built on the top of the toolkit T3DevKit, that is distributed together with the CoDec generator that addresses IPv6 specific issues. The original idea was to provide a library that offers a higher level of abstraction and specialization to the language TTCN-3. The toolkit was developed while researching on the development of IPv6 test suites. As TTCN-3 definition does not provide means for compiled and packaged code distribution, the toolkit is a collection of tightly coupled Open/Free tools, C++ libraries and pieces of TTCN-3 routines and type definitions.

2.3.5.1 Scope

The objective of the IPv6 testing toolkit is to allow quick design of IPv6 test suites with low maintenance cost. It was designed and split in two separate and independent components, the development kit itself, which includes the CoDec generator and is application independent, and its IPv6 specialization. To achieve that, it addresses TTCN-3 design and maintenance issues, as much as providing off-the-shelf type and data structures required for adequate protocol handling. For our team, it was not possible to address a stable data type definition until we solved the CoDec generation issue. At a certain moment in time, there were 3 test experts developing different abstract test suites in parallel for different IPv6 protocols. Until the CoDec Generator was developed, it was impossible to share efficiently the code base of IPv6 definitions. Once the CoDec Generator was available, it became possible to share common definitions of base types and tools through a version managed source repository.

Team efforts on IPv6 testing required working in different parts of IPv6 protocols, ranging from core protocols to OSPFv3 and Network Mobility. Details of protocols implemented are given in 2.3.5.2. Figure 2.16 shows a graphical representation of the components, that helps understanding their correlation.

Main types and data structures are readily available and can be used as sort of building-blocks to design test suites. This cannot be achieved completely as TTCN-3 does not provide mechanisms for function definition overriding or library-style distribution.

2.3.5.2 The IPv6 library

As an specialization of the `t3devlib::` (library presented in 2.3.4.2), the `t`esting `t`oolkit `lib`rary for `IPv6` complements the former by adding functions and tools for handling IPv6 protocols. From the C++ point of view, it is a namespace that contains all components required for IPv6 test handling, and does not require to be completely associated to the concept of a library. Functionally, it comprises TTCN-3 types, functions and C++ codets required for handling IPv6 testing in any toolkit-like environment (not only for our TTCN-3 tool).

The `t3devlib::ipv6::` namespace collects components that were used to address the testing of the following standards: IPv6 & ICMPv6 (RFC2460, RFC2461, RFC2462, RFC2463), IPv6 options (RFC2770, RFC2711), Mobility (RFC3775, RFC3963), IPSec (AH RFC2402, ESP RFC 2406), IPv6 over Ethernet (RFC2464), Routing protocols

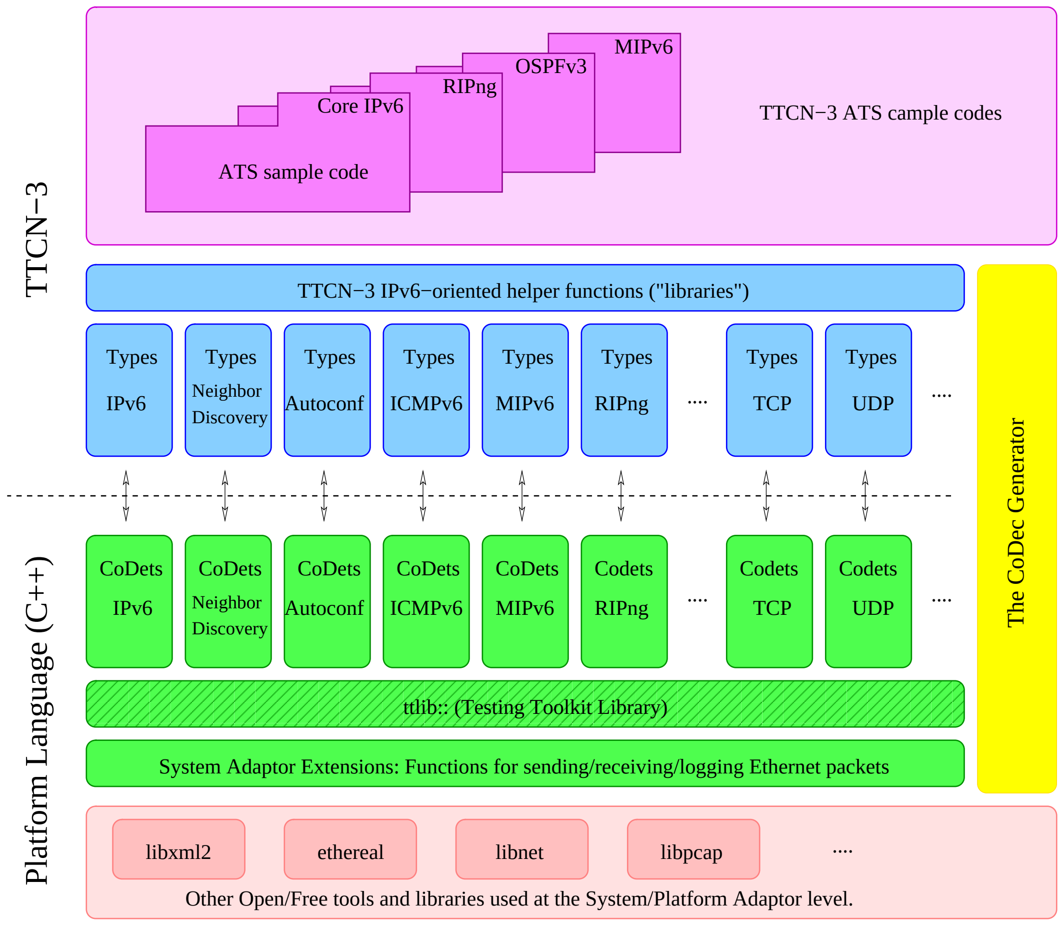


Figure 2.16: Graphical representation of the IPv6 testing toolkit elements

(RIPng RFC2080, OSPFv3 RFC 2740, BGP4+ RFC1771, RFC2858), Transport Protocols (UDP RFC768, TCP RFC793).

Implementation aspects, like endianness, are implemented for standard IPv6 over Ethernet coding and might require to be revised before porting the libraries to other deployment scenarios.

2.3.5.3 An IPv6/ICMPv6 example

The objective of the example is to show the usage of the toolkit for two simple operations in ICMPv6 packet transmission: length and checksum calculation. Even though these operations have to be performed on a packet-by-packet basis, they were hard to implement in our first test suites [SBFV05]. It was understood that it is an accessory to the test suite, but generic CoDecs provided by tool vendors do not support this kind of specialized operations easily. On the other hand, TTCN-3 is not adequate for this kind of bit-oriented calculations, and it is clearly a transmission problem. We will do a high level presentation on how to address this problem using the toolkit, without getting into technical issues which are beyond the scope of this section. Technical details and a tutorial are distributed with the toolkit.

We will just concentrate on the functional handling of properties and leave aside complete type description and function invocation details. Figure 2.17 shows an extract of the codet that is executed at `PostEncode` time. `PostEncode` is the right moment for length calculation, because it is the last access provided before encoding is finished and low level representation is returned to the TTCN-3 invoking call. At this point, it is supposed that all upper level protocol data is already assembled, source and destination IPv6 addresses too. Length can be thus calculated, and only afterward the checksum (because it covers also the length field, that has to be filled beforehand).

The tags `<snip>` indicates parts of the source code that were removed for the sake of simplicity. The first thing that is performed is the length calculation and stored in the buffer in the right position. More code is removed to keep the example simple. The checksum calculation function, `ChecksumIPv6()`, is part of the library `t3devlib::ipv6::` and can be simply invoked. Afterward the checksum is stored in the transmission buffer and the packet assembly is completed.

The removed pieces of code are not relevant to the ICMPv6 checksum calculation and are omitted for readability. This example shows a good compromise between what can be automated, what could be provided by the toolkit and what has to be specified in the platform language in the form of a codet. This codet is part of the toolkit and should be reused, unless it is required to replace it due to some test requirement that affects its calculation.

2.3.6 Summary

The CoDec generator is a specific tool, that addresses the problem of CoDec writing. To achieve this, a whole framework and methodology was designed and implemented.

```
  inline void FrameType::PostEncode (Buffer& buffer)
                                     throw (EncodeError) {
<snip>
        // IPv6 layer: compute the payload length if not given
        Ipv6HeaderType& ipv6 = layer.Get_ipv6();
        if (ipv6.computeLength_) {
                UInt16& len = ipv6.Get_PayloadLength();
                buffer.SetPosition (beginning_of_layer[id]);
                len.SetValue (buffer.GetBitsLeft()/8 - 40);
                buffer.SetPosition (buffer.GetPosition() + 32);
                buffer.Write (len);
         }
<snip>
         // ICMPv6 layer: compute the checksum if not given
         ICMPv6MessageType& icmpv6 = layer.Get_icmpv6();
         if (icmpv6.computeChecksum_) {
                Unsigned& checksum = icmpv6.Get_Checksum();
                checksum.SetValue (
                      ChecksumIPv6 (ip6_pshdr, buffer, id, 2)
                      );
                buffer.SetPosition (beginning_of_layer[id]);
                buffer.SetPosition (buffer.GetPosition() + 16);
                buffer.Write (checksum);
          }
<snip>
  }
```

Figure 2.17: ICMPv6 Codet fragment for checksum and length calculation

The t3devkit holds all the extensions which are required for building the executable test suite from the TTCN-3 abstract specification, augmented with codets.

The t3devkit also addresses the interfacing with TTCN-3, as required by the standard, implementing the TCI-CD, TRI-SA and TRI-PA interfaces. This provides a global coherence for data representation for all the interfaces required for execution directly related aspects. TTCN-3 C/C++ API is not as mature as the Java one in the TTCN-3 specification. It does not offer a simple to use, coherent and intuitive environment. It just provides an API. To overcome this limitation, an Object Oriented framework is provided, as presented in 2.3.4.2, rather than a basic C++ wrapper. A more detailed explanation could be found in the documentation distributed with the toolkit. The framework provides OO handling of TTCN-3 values from C++, and offers services like memory management and communication with the TTCN-3 Runtime System, even the dispatch of TRI calls between every port instance.

To optimize the cost of test suite development and maintenance, the t3devkit provides a set of debugging functions for tracing and controlling the operation of the CoDecs, Platform Adaptor and System Under Test Adaptor components of the tester. This thesis work contributed with this work, which was addressed by the whole team at the laboratory. These capabilities, even beyond the scope of TTCN-3 definition, proved absolutely useful for test case maintenance and development lifecycle.

The complete toolkit is built based on a set of scripts that makes it independent of the tool vendor, provided that it is based on C++. This is also a hard task, as TTCN-3 standard does not describe the way files should be arranged, extensions, or even, how to glue TTCN-3 code together in the form of a library. Tool vendors use their freedom to implement this in different and incompatible ways.

The result of this development is that we now work with a stable base of TTCN-3 code for our code development. Maintenance is reduced, as it is simple to reuse and factorize code. As we increase the reuse, we can take for granted functionality and quality of the code. Development cycles are smaller, cost of test case development and maintenance shrank and we rely on a stable development platform.

Some of the contents of this section were presented, published or are accesible in different ways. The t3devkit is available from `http://www.irisa.fr/tipi/tools_en.htm`, and is distributed under the CeCILL-C license.

2.3.7 Final words on automatic CoDec generation

The TTCN-3 testing toolkit (T3DevKit - CoDec generator with the corresponding set of libraries, type definitions and tools) addresses test suite lifecycle factors. Usage of the toolkit itself is more than just knowing a library, because it addresses TTCN-3 gray areas in the transition from TTCN-3 Abstract Test Suites (ATS) to Executable Test Suites (ETS) and proposes a solution to problems related to development, reuse and maintenance. It proposes a methodology for describing communication objects and translating them into transmittable bitstrings and vice versa. The toolkit main building element is the CoDec Generator, a tool that addresses the problem of CoDec development. It was successfully applied to IPv6 test case design and development, but it is technology neutral in design and philosophy. The tool is now freely available, distributed over the Internet and is being applied to other domains too.

The result is a set of free tools, either for IPv6 protocol testing or general protocol testing. CoDec development complexity was moved into the CoDec Generator, but it can be reused and maintained independently of the test suites. Test suite "source code" now only contains TTCN-3 ATS and codets, which are platform language extensions helpers for the CoDec Generator. The main practical result is that these tools simplified our tasks of test development and maintenance.

Ongoing research includes porting this philosophy of work to the TCI-CD Java API. We were also working on different ways for codet specification. A platform independent codet language is under analysis too, but as we intend to integrate it into TTCN-3 language, we need to have access to a compiler to achieve it.

2.4 Effective use of the extended TTCN-3 architecture

With our first experiments, we showed [SFRV05, SBFV05] that TTCN-3 is capable but not suitable for IPv6 protocol testing. Based on those results, we identified the main problems and developed a methodology and a tool, the T3DevKit, to address reusability and maintenance of the specifications. With the extended TTCN-3 framework, we found that the development of test suites for IPv6 core protocols become simpler. We decided to see how can we use the new tool to scale in protocol complexity. The problem addressed in this Section is to see how to use the extended TTCN-3 framework when designing test suites for protocols that require further message assembly and handling complexity.

Different new test case architecture alternatives become available, and the approach was to obtain empirical data to determine validity of solutions. We aimed at scaling in the complexity of the protocol addressed, thus, we searched for a protocol with complex data manipulation and handling. A protocol that would seem unrealistically testable with our initial approach, but that would allow us to show CoDec generation and methodological benefits and move one step further in complexity. The protocol selected was IPv6-IPsec. The selection forces us to embed cryptographic routines in our test specification, link the ATS with cryptographic libraries and reuse existing IPv6 libraries and functions.

The rest of this Section introduces relevant IPsec aspects, analyzes and describes the different test design options. We summarize the Section with recommendations based on our findings for effective test design.

2.4.1 IPsec testing with TTCN-3

The most popular enhancement of IPv6 is the growth of the IP address space, but several other changes are introduced. One important improvement is that the security aspects are included in the specification. In the IPv6 suite, confidentiality and authentication mechanisms have been specified since the initial drafts. Thus testing IPv6 must include the testing of the new Internet Protocol security features. This is already the case in the world wide IPv6 Ready Logo certification program that provides test suites for IPsec (Internet Protocol Security) [SK05, Ken05, Kau05]. IPsec is a set of protocols that provides cryptographically based security at the IP layer, protecting the network and upper layers. The services offered by IPsec includes: confidentiality, connectionless integrity and data origin authentication.

Test cases to be applied are taken from the IPv6 Ready Logo, as we did before. The test cases themselves exchange only few messages between the tester and the Implementation Under Test (IUT), and could be considered quite simple to implement but they hold the inherent complexity of the encryption, decryption and authentication/integrity algorithms, among others. IPsec specification (by means of an RFC - Request for Comments) indicates which authentication/integrity and encryption algorithms are used. The different and already existing encryption algorithms are described elsewhere. The RFC 4301 [SK05] does not specify the algorithms themselves, but describes how to use them in order to assemble and disassemble IPsec messages. As encryption algorithms are used as building blocks, their implementation is not considered as part of the test purposes. Thus, in the test specification, these algorithms are not required to be implemented in TTCN-3, they are not part of the abstract test specification. Already existing libraries that implement the required algorithms are used.

This Section compares different methodological approaches to reuse existing functions and distribute complexity of the task across TTCN-3 standard interfaces. One possibility is to model the encryption stage as an operation performed and specified in the TTCN-3 Abstract Test Specification (ATS) of the test case. Other possibility is to consider the encryption as a transmission problem, consequently, making the TTCN-3 ATS unaware of the encryption/decryption task. Different decisions lead to different tester configuration and Executable Test Suites for the same test requirement. We explore how these ATS design decisions impact the ETS, simplifying or hardening the test development process. Pros and cons are discussed.

This work should help the reader to understand deeply the different interfaces present in TTCN-3 and how to use them effectively to address particular problems. Different decisions lead to different capabilities and expressiveness of the TTCN-3 ATS. All the experiences uses the T3DevKit, showing its versatility and power. Practical results are presented.

The work is organized as follows. Section 2.4.2 highlights the principal aspects of the IPsec protocol and presents a general description of IPsec tests. Section 2.4.3 introduce the test selected to be implemented, the requirements and available tools used. In Section 2.4.4 the two methodological approaches implemented are introduced. Their detailed implementation are presented in sections 2.4.5 and 2.4.6. They are compared in Section 2.4.7. Conclusions are presented in Section 2.4.8.

2.4.2 IPsec highlights

IPsec is a suite of security protocols that offers access control, connectionless integrity, data origin authentication and confidentiality, among other services, for IPv4 and IPv6. These services offered protect the IP layer and upper layer protocols.

2.4.2.1 Protocol description

Two protocols are used by IPsec to provide security: Authentication Header (AH) and Encapsulating Security Payload (ESP). AH provides connectionless integrity, data origin authentication and optionally anti-replay service. Beside this, ESP may provide confidentiality too. Both protocols also provide access controls by the use of cryptographic keys, that can be distributed manually or automatically. AH and ESP are used in conjunction with a set of cryptographic algorithms specified in RFC 4305 [Eas05].

Both protocols, AH and ESP, can be used alone or can be combined. ESP can be used to provide both functionalities, integrity and confidentiality, or it can be used to provide only integrity, the same functionality provided by AH. This makes AH to be not only a specification requirement, but an option.

The IPsec protocols can be used in two modes: transport and tunnel. In transport mode security is provided for the upper layer protocols and not for the IP header. In the case of AH some portions of extension headers are also covered. In tunnel mode the security protocols are applied to the entire IP datagram, including the IP header.

The security protocol (ESP or AH), the mode, the cryptographic algorithms, how to combine the specified protocols and services and the traffic that will be protected, are specified by the Security Associations (SA) and the Security Policy Database (SPD).

As defined in [Kau05] an SA is a simplex “connection” that affords security services to the traffic carried by it. For a typical communication two SA are required, one for each traffic direction. Also, if AH and ESP protocols are combined, two SA must be created, one for each protocol. Each SA is an entry in the SA Database (SAD). In the SA the security protocol and the mode are specified among other parameters that defines the connection.

The SPD control whether and how IPsec is applied to traffic transited or received. The SPD must be consulted while processing the traffic, incoming or outgoing, even in traffic such IPsec protection is not required.

2.4.2.2 General Test description

The IPv6 forum implements the IPv6 logo with the objective of give confidence to users that IPv6 is available and ready to use. They provide a suite of tests that should be passed to get the logo. Specification conformance and Interoperability are tested. For this work we concentrate in the conformance test suites specified by the IPv6 Ready Logo Technical Committee (v6LC).

IPsec testing is about IPsec, and not about IPv6 testing. IPsec implementation is strongly encouraged in IPv6, but different parts of the protocol suite are tested separately. By the moment IPsec is addressed, IPv6 must have been tested before, and must have got a hundred percent of pass verdicts. The same principle of separation of concerns is applied to the encryption and privacy providers, with the difference that there is no test on the suite that addresses their correctness.

IPsec tests address, as said in Section 2.4.2.1, the two different modes present in IPsec: tunnel and transport. The mode requirement depends on the targeted usage. For each of them, it tests the different combinations of encryption algorithms and the authentication ones. For both algorithms' types two categories are defined: *base* and *advanced*. The algorithms included in the *base* category are mandatory for all equipment and the ones included in the *advanced* are required only for equipment that supports these algorithms.

Manual key configuration is used, but dynamic negotiation of keys nevertheless, is an accepted alternative, using Internet Key Exchange (IKE) protocol. Although, IKE is addressed in a different test suite, devoted to it.

From the two security protocols used by IPsec, AH and ESP, only ESP is required and tested.

During the execution of the selected test case, an IPsec-ICMPv6 Echo Request message is sent to the Node Under Test (NUT). The NUT must receive the message, process it and return an IPsec-ICMPv6 Echo Reply message.

2.4.3 Requirements on the testing platform

This work takes from granted the IPsec test suite definition published by the IPv6 Ready Logo Technical Committee. It is not addressed *what* to test in order to ascertain the correctness of an IPsec implementation, but *how* to do it with TTCN-3.

IPsec test specification is published by the IPv6 Ready Logo as an English written document [TES07], complemented with some graphics. English ATS is translated into TTCN-3 specification, with additional, test specific, functions implemented through the standard TTCN-3 interfaces. Main requirements include IPv6 data type handling and cryptographic routines.

A TTCN-3 test system can be thought conceptually as a set of interacting entities, each implementing a specific test functionality. Figure 1.3 shows the general structure of a TTCN-3 test system. We will focus only on the main concepts addressed by this work.

The TTCN-3 Executable (TE) interprets and executes compiled ATS. SA, which stands for SUT Adaptor (System Under Test Adaptor), "adapts" communications between the TTCN-3 system and the SUT. The Platform Adaptor (PA) implements (amongst others) external functions. External functions are convenient ways of executing platform language code, in our case, ANSI C. The Test Management (TM) entity is responsible for overall management of a test system. Finally, the Coding and Decoding (CD) entity is responsible for the encoding and decoding of TTCN-3 values into bitstrings suitable to be sent to the SUT. All these definitions can be found in [ETS05a, ETS05b, ETS05c].

We will use the following simplified examples to give a grasp of the semantic behavior. It is important to understand the interaction of these elements during a `send` operation. The runtime behavior of a send operation is to take the template, hand it to the associated CoDec through the TCI/CD interface and obtain its representation as a BinaryString. The bit-oriented representation is passed then to the TRI/SA function that implements the `port` implementation, ultimate responsible of the transmission.

Another important operation is the invocation of an external function. External functions provide ways to extend TTCN-3 language with platform language functional-

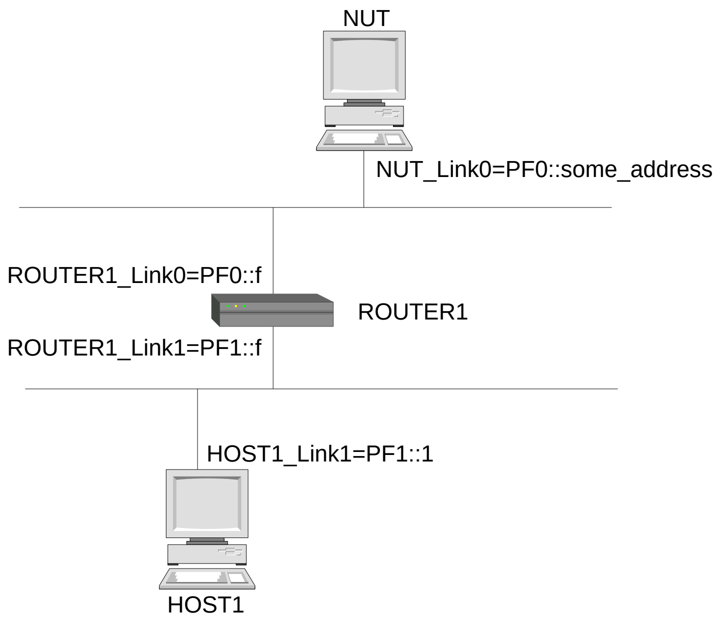


Figure 2.18: Test Topology.

ity. When an external function is invoked from the TTCN-3 ATS, the runtime behavior requires that the TTCN-3 variable is converted into its BinaryString representation using the associated CoDec through the TCI/CD interface. The bit-oriented representation is passed trough the TRI/PA interface to the registered external function, that will perform the expected operation over the bitstring. Upon the function return, the BinaryString is again used to invoke the corresponding CoDec, this time to obtain a TTCN-3 variable out of the bitstring, through the TCI/CD interface again.

These are the basis for extending TTCN-3 functionalities with platform language. Section 2.4.4 explains different ways of using this API to generate IPsec messages.

2.4.3.1 The selected test case

This work bases its results on experiments made basically on a specific test case, number 5.2.3 of [TES07]. The test case has an large and detailed preamble detailing Security Association Databases (SAD) configuration and Security Policy Databases (SPD) for each node. Exchanged packets are also detailed.

This work is focused in the implementation of transport mode test, with encryption algorithm 3DES-CBC and authentication algorithm NULL. Figure 2.18 shows the test topology: the way involved nodes are corrected.

Figure 2.19 extracted from [TES07], describes the test procedure (scenario) and verdict criteria. It is noticeable that the procedure consists of a stimulus and a response, with a statement regarding of the correctness (judgment).

```
Procedure:
HOST1_Link1(TN)         Target(NUT)
|                            |
|--------------------------->| ICMP Echo Request with ESP
|                            |
|<---------------------------| ICMP Echo Reply with ESP
|                            |     (Judgment #1)

1. HOST1 sends "ICMP Echo Request with ESP"
2. Observe the packet transmitted by NUT

Judgment:
Judgment #1
Step-2: NUT transmits "ICMP Echo Reply with ESP"
```

Figure 2.19: TestCase 5.2.3: TransportMode ESP=3DES-CBC NULL

2.4.3.2 Available tools

At the time this work began, there were no TTCN-3 IPv6 IPsec libraries or tools available on line that could be reused. IRISA's T3DevKit [t3d07] with IPv6 examples and ETSI's TC MTS-IPT [ETS07] TTCN-3 IPv6 test tools and suites were publicly available. Both of them seemed a suitable starting point for our development. IRISA's T3DevKit was selected due to existing knowledge of the tool and to the fact that no particular aid to IPsec testing on ETSI's public Abstract Test Suites (ATS).

As encryption routines are not part of the test purpose, it was decided to reuse existing ones. GNU Libgcrypt was selected because it is freely available, there are good examples of its usage and there is experience of its IPsec usage.

The rest of this Section analyzes these building blocks and the test development challenges to be addressed.

2.4.3.3 T3DevKit

The T3DevKit is a helper for implementing TRI-PA, TRI-SA and TCI-CD interfaces in order to build the executable test out of a TTCN-3 abstract specification. It provides the T3CDGen, an automatic generator that extracts type definitions present in the TTCN-3 source files and generates most of the C++ code needed to implement the logic of the TCI-CD interface. The T3DevLib provides a framework of C++ classes that eases TTCN-3 data type handling, together with port and timer definitions suitable for working over Ethernet networks.

2.4.3.4 Libgcrypt

Libgcrypt is a general purpose cryptographic library which works on POSIX systems. It is built based on the code from GNUPG and provides functions and support for several cryptographic ciphers, hash algorithms, message authentication codes (MAC), etc. It has a broad user base and provides all the functionality required for implementing IPsec.

2.4.3.5 What can be reused and what has to be developed

With the tools selected, we have enough building blocks to simplify our abstract test design and development process. We already have implementations for IPv6 packets, ICMP messages, UDP datagrams and TCP segments. Most of this code can be reused to perform the IPsec test cases, and some just needs to be adapted with minor modifications. Implementations for 3DES, SHA and other cipher related functions are also available. What is required now is to glue all these things together and to build the ETS.

It is clear that IPsec TTCN-3 data types have to be developed, together with their corresponding encoding and decoding functions. Also the TTCN-3 templates that will be used for the tests have to be defined. Beside this, we have to integrate the new code implemented and the reused one. Figure 2.20 shows the TTCN-3 data types defined for representing the ESP message.

```
type record ESPMessage {
        octetstring     SPI length(4),
        UInt32          SeqNum,
        EncPayload      Payload,
        octetstring     ICV optional
  }

type record EncPayload {
        IPDatagram Data,
        integer TFCPadding optional,
        octetstring Padding length(0..255) optional,
        UInt8 PadLength,
        UInt8 NextHdr
  }
```

Figure 2.20: TTCN-3 data types for ESP message

Further explanation of the test cases implemented and details of the implementations are presented in the following Section.

2.4.4 Alternatives on test case design

CoDec task is to convert TTCN-3 objects into transmittable bitstrings. Particular details of the communication are removed from the TTCN-3 ATS and relayed to CoDecs. The direct usage of the T3DevKit suggests also to relay other functionalities to the CoDecs, like message size calculation or checksum computing. The natural way to extend this methodology would be to perform cipher operations on the CoDec, moving there all IPsec handling. This way of assembling IPsec messages does not seem natural, as all the IPsec assembly is done in C++ CoDecs.

Another approach is to embed cryptographic operations inside the TTCN-3 code using external functions. This approach provides more control to the ATS during the encryption process, lightening the weight of the CoDec. T3DevKit also offers a wrapper to give access to external functions, which we used.

The following subsections describe these two test development implementation strategies.

2.4.5 First approach: encryption and codification relayed to the CoDec

The natural way of extending CoDec based, existing public ATS, to address IPsec was to implement cryptographic and authentication functions in the CoDecs too. We should analyze independently transmission and reception operations.

2.4.5.1 Transmission

Performing all the encoding in the CoDec removes most of the cryptographic details from the TTCN-3 code. In this way the ESP packet is modeled in TTCN-3 without applying any cipher algorithm and passed to the CoDec. The C++ CoDets perform the corresponding cryptographic operations and assemble the packet that will be finally sent to the NUT.

```
Link1.send(ICMPv6WithESP_EchoRequest_AuthNULL(SPI_SA1, DATA));
```

Figure 2.21: Complete transmission processing in the ATS

The unencrypted message template is sent to the CoDec. The CoDec receives the TTCN-3 values and encode them into bitstring, but there are several things to be done. Before building the BinaryString with the transmittable representation, part of the message must be encrypted, and before encrypting some values must be calculated. T3DevKit provides Encode and Decode methods for each field of a packet. These methods simplify finding the fields that must be encrypted but it is an intricate task to calculate fields like checksum, padding and padlength.

The length of all the fields have to be calculated to be able to determine the padlength. Handling of the bitstring representation is not natural in C++. Even though the T3DevKit provides tools for handling the bitstring (a cursor and operations over the bitstring representation), the task is error prone. Indeed, as the T3DevKit works on the bitstring, but C++ native libraries work on memory addresses, byte oriented, several translations have to be performed from the bitstring into byte representation and vice versa. These operations are highly error prone.

Although the T3DevKit soothes the work, there is not a common representation of types and data between TTCN-3 world and C++ one.

2.4.5.2 Reception

For the reception of messages, the same design decision of moving all the cryptographic operations to the CoDec can be applied, leading to a very clear abstract specification. Figure 2.22 shows the piece of code corresponding to the test case implementation.

```
alt
  //Receive the correct answer
  [] Link1.receive(ICMPv6WithESP_EchoReply_AuthNULL(SPI_SA2, ''O))
         setverdict(pass);
         replyTimer.stop;

  //Receive incorrect answer
  [] Link1.receive
         setverdict(fail);
         replyTimer.stop;

  //Receive no answer
  [] replyTimer.timeout
         setverdict(fail);
```

Figure 2.22: Complete reception processing in the ATS

It is straightforward to follow the logic of the message reception. The `pass` verdict is only issued if the received message can be matched to the `ICMPv6With ESP_EchoReply_AuthNULL` template. In any other cases, a `fail` verdict is issued. All the logic regarding proper encryption is placed on C++ CoDets.

The power given by the T3DevKit tool to the CoDec generation translated parts of the protocol complexity to the coding operation. Length calculation can be considered a simple operation, that can be handled during encoding. Checksum calculation is not a simple operation (at least not as simple as length calculation), but the CoDec is an elegant place to perform it. We should analyze the result of removing cryptographic tasks from the CoDec in the following subsection.

2.4.6 Second approach: encryption/decryption done in external functions

By "encryption/decrytion done in external functions" we describe the decoupling of purely coding operations from semantically rich ones. Even though it is possible to discuss what is purely coding, we think that performing cryptographic operations cannot be considered a simple operation.

2.4.6.1 Transmission

The objective of this design decision is to be able to have a complete encoded value, accessible and represented in TTCN-3 variables before the final BinaryString encoding is performed. The CoDec do not need to perform operations to the objects received from TTCN-3, but just to convert the TTCN-3 value into its bit oriented representation.

```
    template ESPMessage ICMPv6ESPMessage (IPv6AddressType src,
                        IPv6AddressType dst, octetstring m_spi,
                        octetstring m_data, UInt16 checksum) := {

              SPI:= m_spi,
              SeqNum := 1,
              Payload := EncryptPayload(src, dst, EchoRequestType,
                                       m_data, checksum),
              ICV :=omit
   }
```

Figure 2.23: TTCN-3 template for ESP with external functions

The Figure 2.23 shows how we use external functions to compute cryptographic generated values of the ESPMessage. It is possible to see how external function invocation is embedded in the template definition with the `EncryptPayload()` function. The ESP message template definition includes the parameters it receives, and the ones that have to be passed to the external function responsible for performing the encryption.

```
var UInt16 checksum := GetCheckSum(PF1_1, PF0_1, EchoRequestType,
                                   DATA, NextHeaderIcmpV6);

Link1.send(ICMPv6WithESP_EchoRequest(PF1_1, PF0_1, SPI_SA1,
                                   DATA, checksum));
```

Figure 2.24: TTCN-3 checksum calculation

Before encrypting the payload, its content (the ICMPv6 packet) must be built. Consequently, its length and checksum need to be calculated and accessed from TTCN-3. Thus, we need to use external functions in this case too. We illustrate checksum calculation in Figure 2.24. The checksum is calculated calling the external function `GetCheckSum()` before assembling the packet. The calculated checksum is passed as a parameter to the template defined for the ESP message.

2.4.6.2 Reception

This approach also introduces changes, challenges and differences in reception operations, and the way received information is treated. External functions can also help validating the message and are used to decrypt the message. It is pretty straightforward to see that TTCN-3 matching mechanisms based on wildcards do not apply inside encrypted structured data fields. They have to be decrypted first.

Figure 2.25 shows actual `alt[]` used for encrypted message reception and verdict issuing. The message is received and compared to the corresponding template, shown in

```
alt{
     //Receive correct answer, unverified encrypted payload
     [] Link1.receive(ICMPv6ESPMessage_Answer_AuthNULL
                 (PF0_1, PF1_1, SPI_SA2, DATA, checksum)) -> value Myvar {
           var bitstring encpayload := Myvar.Payload;
           var UInt8 payloadLength := lengthof(encpayload)/8;
           var EncPayload payload := DecriptPayload(encpayload, payloadLength);
           if (match(payload, ICMPv6EncPayload_Answer(PF0_1, PF1_1, DATA))) {
                  setverdict(pass);
           } else {
                  setverdict(fail);
           }
           replyTimer.stop;
     }
     //Receive incorrect answer
     [] Link1.receive {
           setverdict(fail);
           replyTimer.stop;
     }
     //Receive no answer
     [] replyTimer.timeout {
           setverdict(fail);
     }
}
```

Figure 2.25: TTCN-3 code to validate received encrypted message using external functions

Figure 2.26. The resulting value is assigned to the variable **MyVar**. From this variable the encrypted payload is extracted and passed to an external function to be decrypted. The decrypted value is then decoded into the type **EncPayload** and then passed to the function **match()** (provided by TTCN-3), to be compared with the corresponding template. Whether the result is true the issued verdict is `pass`. `fail` is issued in other cases.

```
template ESPMessageAnswer ICMPv6ESPMessage_Answer_AuthNULL
                     (Ipv6AddressType src, Ipv6AddressType dst,
                      octetstring m_spi, octetstring m_data,
                      UInt16 checksum) := {

        SPI:= m_spi,
        SeqNum := ?,
        Payload := ?,
        ICV := omit
  }

}
```

Figure 2.26: ESPMessageAnswer template

The checksum is also verified with an external function invocation, defined in the template to check the incoming ICMP echo request. Figure 2.27 shows the template and the checksum calculation function.

```
template ICMPv6MessageType icmpv6_EchoReply (Ipv6AddressType src,
                      Ipv6AddressType dst, octetstring m_data) := {

      Type:=Icmpv6EchoReplyType,
      Code:=uint8_0,
      Checksum:=GetCheckSum(src, dst, Icmpv6EchoReplyType,
                          m_data , NextHeaderIcmpV6),
      ChecksumShouldBe:=omit,

      body := {
          echo :={
              Identifier := uint16_0,
              SequenceNumber := uint16_0,
              Data := m_data
          }
      },
      Options :=  omit
}
```

Figure 2.27: icmp echo reply validation template

For handling external functions T3DevKit provides an implementation of *triExternalFunction()* for multiplexing calls and presenting the data and a class for manipulating the parameters. External functions permits building the complete message in TTCN-3 types and data structures simplifying and reducing the codification in the CoDec. Comparison between the two methods is done in the following section.

2.4.7 Comparison

Two different approaches to TTCN-3 test case design of IPsec protocol test cases were implemented and shown. First we showed a direct extension of the IPv6 examples provided with the T3DevKit, that performs all the required tasks at the CoDec level. The other approach presented was using external functions to control the complete message assembly from TTCN-3, using only CoDec for data representation conversion between TTCN-3 variables and transmittable bitstrings. In the following, we compare these two approaches, according to different criteria.

2.4.7.1 ATS + TCI/TRI code design

We analyze all the aspects required to produce an executable test case, not only the TTCN-3 ATS. We compare both TTCN-3 specification and platform language readability together, in spite of the fact that maybe different groups of developers, with different backgrounds, address each part. First, we address message transmission and afterward, reception.

Without external functions, message assembly and encryption are performed in a single function. Moreover, as the CoDec (accessed from the `send()` operation) is not intended to be used as a regular function, it does not receive other extra parameters than the template to be transmitted. No control on the encryption keys or other arguments specified in the test specification is accessible from the TTCN-3 code. With the usage of external functions, message is passed to the CoDec with all the data fields calculated, thus only BinaryString encoding is required. The logical sequence of the code is simpler to understand, as separated abstract concepts are mapped to individual functions. The test case becomes simpler to implement, as divide&conquer principles apply now. Just specific functions are implemented in C++ to calculate some field values that could not be implemented with TTCN-3, yet the message assembly logic and sequence is handled from the TTCN-3 ATS.

While receiving the message for validation, an external function is used to decrypt part of the message and then compare it with the corresponding template. The complete validation process is done in TTCN-3. Without the usage of external functions, only part of the message construction is controlled from the TTCN-3 ATS. Some fields like checksum, padding and padlength are not calculated in TTCN-3 and have to be added in the CoDec. This becomes relevant for the payload generation. We need to have an ICMPv6 Echo Request message, whose creation has been delegated to the CoDec. To keep our implementation aligned and coherent with existing one, ESP assembly should be relayed to the CoDec too. This fact forces that a part of the ATS is moved to the CoDec and is not specified in TTCN-3 language. The test case is then split into TTCN-3 and C++ CoDets. To understand the test you have also to know the code implemented by the CoDec. This approach seems not to follow the TTCN-3 philosophy and tends to put too much semantic in the CoDec.

2.4.7.2 Test specification size

As we are comparing two implementations of the same specification developed with the same language it is possible to compare the methodologies based on some code metrics. The first thing that is important to consider is that existing IPv6 types and definitions were reused, and they are the biggest part of the TTCN-3 code.

		CoDec only	CoDec + Ext. Functions
TTCN-3 (loc)	Test case	81	81
	Accessory	812	797
C++ (loc)	ext	0	234
	CoDet	681	255
TTCN-3		893	878
C++		681	489

Figure 2.28: Some *loc* based software metrics

Table 2.28 shows the different parts of the code we measured. The metric used is the effective lines of code (*loc*). Comments, blank lines, lines with only block delimiters and other kinds of "empty" lines were removed, and only a single command per line was accepted. We separate the lines of codes directly required for the test case specification from all the *accessory* ones. Accessory lines of code are the existing definitions that we re-use for modeling lower layer protocols or elements not directly required by the test case. In this example it is mainly reused code with none or at most minimal adjustments.

Size of TTCN-3 code is equivalent in both methodologies, with only slight adjustments. It is noticeable that test case specific code accounts for a 10% of all the required code. Modeling of IPv6, ICMPv6, options accounts for most of the TTCN-3 code, which we managed to reuse from publicly available IPv6 ATS.

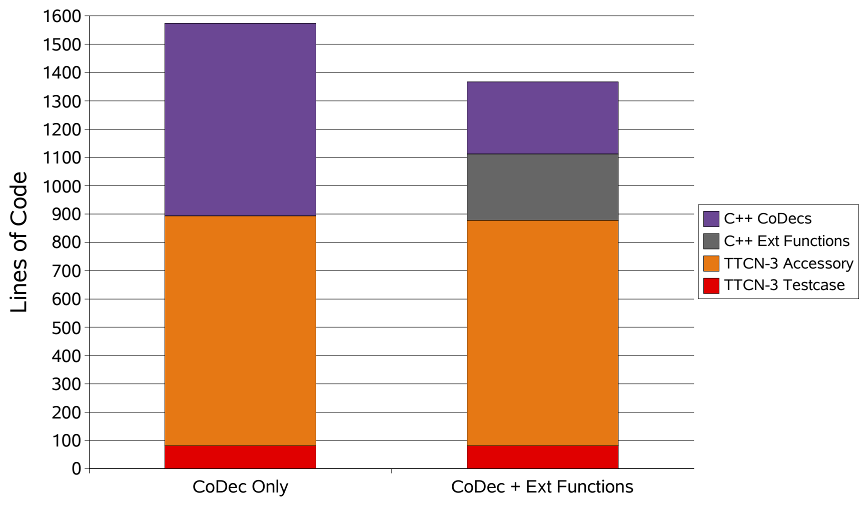


Figure 2.29: *loc* graphically compared

An alternative, graphical, representation of Table 2.28 is shown in Figure 2.29. Differences arise when we consider platform language coding, both for CoDec and External functions. It was part of the methodological approach to avoid external functions in the first implementation, accounting only for CoDec implementation. The second implementation methodology splits the complexity, but it is more than splitting it. It diminishes the number of lines of code required. The total number of lines of C++ code shrunk almost 30%, spread over a bigger number of shorter functions. These properties suggests that the code is also more maintainable.

2.4.7.3 Performance

The main drawback found in the usage of external functions is the performance overhead. Every time an external function is invoked, the TTCN-3 values are encoded and passed to the external function. Upon exit, values are decoded and passed back to TTCN-3. None of this happens in the approach that does not require external functions.

The code requires 4 external function invocations during transmission and reception, thus 4 additional code and decode cycles. The performance impact of this is relevant for time sensitive test cases, but not in general in the case of conformance testing.

2.4.8 Outcome

We acknowledge that the comparison was applied only to subset of the whole Conformance test suite, but we believe interesting conclusions can be taken. The TTCN-3 ATS developed when putting all the operations in the CoDec is very clean and readable, but we feel that important parts of the test specifications have to be moved to the CoDec. Too much IPsec behavior is not expressed in TTCN-3 language and is relied to CoDec. CoDec abstraction level -even augmented with the T3DevKit- is too low and operations are hard to maintain and implement. We think that this approach diverges from TTCN-3 design strategy.

Moving all the operations to the external functions provide a much more comfortable framework. No changes in the size of TTCN-3 were found, and it is still abstract enough, while keeping all required semantic for more detailed test case operations. The weight of the CoDec is lightened, but the number of invocations grew significantly. The total number of *loc* in platform language shrunk, making the test case smaller and easier to develop. It seems that with this approach we obtain better designed test cases, at the expense of performance degradation.

Further studies are in progress, but current findings seems to indicate that the best option is to design test cases making use of the external functions, whenever performance restrictions allows it. Despite that, we think it is a good approach to leave simple operations in the CoDec. Without trying to define what *simple* means for all possible ATS, our experience seems to indicate that all operations that are related to the experiment definition should not be relayed to the CoDec. If there is a doubt, then is better to implement that operation as an external function.

2.5 Platform language independent CoDec generation

The relevance of CoDec generation has been presented and its consequences on the ATS style, complexity and reusability. A solution for addressing automatic CoDec generation was already presented in 2.3.4. The CoDec generator is a methodological proposal to integrate additional information that eases bridging the ATS to ETS gap by bringing closer to the testcase development cycle CoDec generation issues and automating manual activities of the task.

TTCN-3 language requires runtime entities to be bound through TCI and TRI interfaces to build a runnable ETS. These runtime components are standardized either in C++ or Java languages, what we call platform languages. The CoDec generator, an automatic tool for implementing the TCI-CD entity is currently specialized in C++. In spite of the methodological and practical achievements, the CoDec generator is still a platform language dependent tool.

This Section proposes another approach for CoDec generation, a platform language independent one. The proposal is to replace the CoDets described in 2.3.4.3 with platform independent type meta-information. TTCN-3 type definition is extended with meta-information about the types. Some concrete examples based on IPv6 message oriented protocols are used to explain part of the proposed solution and for validation purposes.

This Section is structured as follows. Firstly, section 2.5.1 recalls TTCN-3 basic component architecture, the type system and matching mechanisms for incoming/outgoing data. Section 2.5.2 introduces our methodology and extensions for automatic CoDec generation, together with a minimal example. The summary is presented in section 2.5.3.

2.5.1 TTCN-3 messaging and matching

TTCN-3 architecture was globally presented in 2.3.1.2. For readability purposes, relevant aspects are recalled and extended. TTCN-3 messaging architecture, described in 2.1.1.2 and in 2.3.1, defines that the communication between the tester and the Implementation Under Test (IUT) takes place through *ports*, modeled as an infinite FIFO queue for the reception. Incoming data is queued until processed by the component that owes the port by consuming it. Two communication paradigms are implemented in TTCN-3: message-based and procedure-based. As each port has a fixed type, the kind of communication primitives supported is fixed too. Target protocols, like Internet ones, are message oriented, thus, we concentrate on the message-based paradigm.

2.5.1.1 Short description to standard message reception

In TTCN-3 message reception -and transmission- is handled through ports. The way a message is retrieved from the input queue associated to the port is issuing the `receive()` method on the port instance. The `receive()` primitive is blocking. It receives optionally a certain *template* as an argument to allow filtering or determining which messages are being waited for. This provides means for message classification upon reception, provided that all possible incoming messages are known.

In a given moment of the test execution, it is possible that the implementation sends different valid messages, making it impossible to base message reception on a single blocking primitive. TTCN-3 introduces `alt` statements, collections of `receive()` statements inside a block. When a message arrives, the different `receive()` alternatives are traversed top-down and the first matching option is taken. Each possible `receive()`

```
alt {
   [] somePort_1.receive (someAnswer (someValue_A)) { ... }

   [] somePort_1.receive (someAnswer (someValue_B)) { ... }

   [] somePort_1.receive () { ... }

   [] somePort_2.receive () { ... }
}
```

Figure 2.30: Simple message reception

statement handles each of the possible messages, solving the problem for unknown, unexpected or non conformant message arrival.

The piece of TTCN-3 code shown in figure 2.30 depicts previous concepts. The first two matching rules catch messages based on the same type and template, but differing in some value. In most of the situations it is important to be able to receive other messages apart from those initially expected. Several reasons impose that fact, ranging from other auxiliary protocols running on the wire to wrong IUT messages. Thus, message must be accepted from any of the ports `somePort_1` and `somePort_2`.

When receiving a message, there will be specific parts of the message that are known beforehand and others that will remain unknown (i.e. *time-to-live* fields or checksums in general). Within templates it is possible to indicate which parts of the message are already known and have a fixed value and to leave other parts unknown. This is done by interleaving fixed values with wildcards. TTCN-3 [ETS03, ETS05a] defines two wildcards: `"?"` and `"*"`, which stand for *AnyElement* and *AnyElementsOrNone* respectively.

TTCN-3 does not standardize requirements for minimal TCI-CD implementations by tool vendors. The reader is encouraged to review TCI and CoDec concepts in section 2.3.1 before continuing.

2.5.2 Extensions for automatic codec derivation.

Manual CoDec writing is a time consuming task and sometimes error prone. CoDec are intended to be reused through different test cases in similar test suites, thus changes generally force major code revisions, both in the TTCN-3 ATS and in the Java/C++ platform language. The proposed methodology automates the task at the expense of a few type design constraints and the addition of CoDec metadata.

As the operations performed by the CoDecs are intended to cooperate with the translated ATS, we do not need to change the TTCN-3 language itself. We need to complement the ATS specification thus, the modifications we propose are only extensions to TTCN-3 ATS comment sections. This allows a simple integration with existing tools, no modifications to standards and only extensions into comments sections, similar to other comment extensions and philosophy as those used in products like JMX, Doxygen, TestNG, JavaDoc, etc.

We will present the methodology by example, showing how to map some simple data types, compositional data structures and introducing metadata extensions. In section 2.5.2.1, different options of how to include the metadata are discussed. Other hypothesis regarding the field of application of current work are introduced. Section 2.5.2.2 provides guides to achieve an adequate data definition, compatible with the CoDec implementation. Afterward, section 2.5.2.3 discusses address-oriented issues relevant for identifying and describing dependencies between parts of the types and their corresponding bitstring representations. In section 2.5.2.4 functions relevant for determining data length are introduced. Section 2.5.2.5 introduces rules for the coding and decoding of fields at CoDec level. Finally, the extension proposal is applied to a simple TCP header type definition in section 2.5.2.6.

Current scope of the work copes requirements for IPv6 handling, while extension to complete TTCN-3 data types and constructs is expected to be addressed too, in the broadest TTCN-3 philosophy.

2.5.2.1 Hypothesis and extension options

This methodology addresses message-based communication paradigm. It was conceived for network protocol testing, and even though several concepts can be easily extended, it is not directly applicable to procedure-based paradigms.

We assume that we can address pieces of information inside messages using the byte as the unit for expressing offsets. This is a common practice in Internet related protocols. We also assume that each packet carries enough information to be decoded. This kind of practice is well known and extensively applied in most protocols and it is based on prepending each uncertain content with a well known and defined descriptor. We will propose extensions to describe this semantics later. The extensions are aimed to be triggered by `send()` and `receive()` methods. Thus, they will refer to both TTCN-3 types and templates.

The CoDec are standardized outside TTCN-3 to be coded in either of Java or C++ languages and will interface TTCN-3 code through the TCI-CD interface. Thus, the automatic derivation of the CoDec is a task that has to happen before actual TTCN-3 link-edition takes place. Moreover, it is a task left aside of the TTCN-3 language. The extensions, aimed for this task, will not be considered during TTCN-3 code execution and do not need to be included in TTCN-3 language. The extensions are dependent on type definitions. Thus the decision was made to include them in the same source files. TTCN-3 provides the keyword `extension`, as an attribute that all TTCN-3 language elements can have, specified by the user. They are aimed for handling at compile-time by tool vendors, which is not the type of use we intend. The unavailability of an open compiler that can be extended, forces the definition of extensions independent of the language compilation process. Another problem is that, for complex types, the number of rules that have to be added might impose readability issues on the code. The `extension` attribute is intended to be a single one for each language element. It forces the packing of all required definitions into a single line of code, affecting code readability as mentioned. Lastly, as `extension` attribute is not standardized, different behaviors (including impossibility of compilation) might be expected from compilers.

The selected choice is to extend comment blocks (similar to extensions found for JavaDoc, TestNG and other) so as to avoid any kind of incompatibility. The decision is to extend `/*...*/` blocks into `/*CD...DC*/` CoDec definition blocks. In this way it is possible to introduce metadata, while remaining compatible with TTCN-3 language definition and solving readability issues. An interesting solution to this problem would be if TTCN-3 would provide a block extension syntax for the `extension` attribute.

CoDec will be automatically built for types that have a `/*CD ...DC*/` definition prepended and all those reached through the transitive closure of the former.

2.5.2.2 TTCN-3 type definition and coding considerations

Type definition is tightly related to CoDec, as the reason for the CoDec is to convert *values* (instances of TTCN-3 types) into *TRImessages* (bitstring representations) and viceversa. The process of type definition has to take into account the fact that messages have to be encoded and transmitted and also received and properly decoded.

Our methodology, and the automatically generated CoDec will encode and decode messages traversing the underlying type definition tree, according to the precise definition and order of the fields. The tree will be built starting from the type definition, which will be the root. Basic types will be the leaf nodes and structured types, the internal nodes. When encoding, the leaves -and only the leaves- of the definition tree are traversed in the same order that they were defined, *"left-to-right"*. When decoding, the bitstring will be traversed and matched against known parts of the definition. As more parts of the message become known, they will help determining the structure of the message, and finally the corresponding TTCN-3 data type.

This methodology implies that we require a precise knowledge of the sizes of all the resulting encoding of each part of the message. This knowledge is not complete in all TTCN-3 types, as will be discussed and extended in the following section. By now we will only mention that it is important to avoid ambiguous type definitions at simple type level. The main problem found here is regarding the `integer` type, which has an unknown size. It has to be replaced by a known sized subtype (i.e. `int8`, `uint16`). TTCN-3 `char` type is suitable for higher level message handling, where multilingual message handling issues might be relevant. For network handling of general strings it is more suited, and recommended, the use of `int8` type.

As a compendium of the previous concepts, we might summarize the methodology with the following statement: "the test designer has to define types as close as possible to physical encoding".

2.5.2.3 Addressing parts of the message and types

When describing properties inside types regarding its components, it is required that we can univocally refer to specific parts. Our addressing keywords just follow standard keywords used in languages like Java and C++.

We will use the keyword `this` to reference the current instance of the given type that will be known only in runtime. In the case of compound types, the name of the field will be used to access its value, in the standard way. The use the standard dot notation to address elements is available. Notations `<field>` and `this.<field>` are equivalent.

Elements inside strings, arrays and `record of` definitions should be addressed using the standard positional scheme. The `some_field[4]` refers to the fifth element of the `some_field[]` array, as indexing begins with the value zero (0).

The last addressing schema defined is a physical oriented one. The keyword is `addr` and it is intended to be used to specify determined positions into the encoded message, even though it can be used while encoding. The syntax is presented in figure 2.31.

```
addr(<reference>,<offset>,<target_type>)

addr(<reference>,<offset>,<target_type>,<encoding_type>)
```

Figure 2.31: Syntax of `addr` keyword

Conceptually, `addr` is equivalent to the casting concept in C language. It will allow us to interpret a specific part of a message according to a given `<target_type>`. It will be the one used for decoding and interpreting the raw bitstring addressed upon reception. The part of the message addressed is given by an initial `<reference>`, like `this` or `this.<field>`. The `<offset>` is the number of bytes skipped from the reference until the beginning of the `<target_type>`. `addr` is overloaded and can receive the optional `<encoding_type>` argument, intended for non standard encoding.

2.5.2.4 Unknown field length functions

We have discussed how to handle simple types and how to avoid length ambiguities by the right selection of basic types. Unfortunately it is not always possible to determine beforehand the size and type of the information to be transmitted.

The encoding process is straightforward, as all the values have to be instantiated before issuing the `send()` method. The problem arises when there is a raw bitstring to be decoded corresponding to multiple valid responses to a given stimulus. An hypothesis to this proposal is that outside the unknown part of the message but in a well known position, there exist some encoded description that makes the decoding process not ambiguous. As an example, Internet Protocol version number is represented in the first nibble of the message in the same position for both protocols IPv4 and IPv6. In this way we can know the version of the protocol before further analysis. The payload length is encoded in a field before the payload in the message, in a well known position. The latter is different in IPv4 and IPv6, but they follow the same design criteria, and the payload length position is well known after examining the first nibble of the message.

Length of structured types can be calculated as the sum of elements, provided that is possible to know their length. For simple types it is straightforward, as the type definition implies encoding size.

The following CoDec extension is introduced so as to provide direct dependencies between message fields that express the type and size. The selected tag and its syntax is shown in picture shown in figure 2.32.

```
@len_dep <orig> <dst> [<encoding_type>]
```

Figure 2.32: Simple message reception

The tag defines that the length of the variable `<dst>`, will be encoded into the variable `<orig>`. `<encoding_type>` is an optional argument which can be used to override standard encoding. Both `<dst>` and `<orig>` should be defined and addressable at compilation time, as the CoDec utility produces a CoDec for the static type definition.

This binding between data fields only expresses a relation in the type definition that makes decoding possible: when a message of this type is received, the length of the unknown field is determined from the length of the known one based on the inverse of the encoding function used for encoding the value. The relation can also be used for automatically filling fields in the CoDec, as we will see in the next subsection.

This clause only expresses a relation between fields, but does not particularly determine the relation function. Default codification (the number of bytes occupied by `<dst>` is encoded in 2-base in `<orig>`), can be replaced by another one, as shown in the next subsection.

Other possible situation is that the encoding schema is not based in a direct representation of the data length, but a reference to a well known standardized index, which is not represented in the protocol. This is the case of Ethernet type/length field, whose possible values are defined by providing a map that binds the actual value[2] with the type of payload. For these situations we define a different dependency, shown in figure 2.33.

```
@map_dep <orig> <dst> {(<value>,<length>,<type>)[(<value>,<length>,<type>)...]}
```

Figure 2.33: Simple message reception

Again the relation binds a couple of fields, specified by `<orig>` and `<dst>`, but in this case, the relation is expressed by the set of tuples containing the `<value>` found in the field `<orig>` and the `<type>` and `<length>` of the field `<dst>`. In many cases `<type>` and `<length>` are redundant, but in arrays they are not. A special keyword `_DEFAULT_` is reserved for matching previously unmatched entries. As we are not redefining the language, the type coding has to take care of this to avoid field name collision.

[2] `http://www.iana.org/assignments/ethernet-numbers`

2.5.2.5 Coding and decoding functions

We have introduced extensions that express relations between parts of the type definitions, and were prepended to type definition clauses. We will present further extensions that are intended to perform modifications in the values of the data to be transmitted and received, and basically aimed for template definition clauses.

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. It is possible to associate different templates to a single type definition. Thus, our methodology associates binding rules to types and encoding/decoding actions to templates. In this way we are able achieve flexibility of usage of rules based on the same type for different purposes. As an example, we would like to encode, in different variables of the same type one correct packet and one that would correspond to an erroneous checksum calculation so as to test some property of the IUT. If the action of encoding the checksum is declared at type definition level, all variables and templates declared after it would *inherit* the checksum calculation, forcing us to declare different types with different extensions to achieve our goal, at the expense of a non clean data modeling. With the possibility of defining actions at template definition level, the test designer can create templates based on the same type that best meets the test purposes.

There are basically two different moments where the functions can be invoked: before and after the conversion takes place. There are operations that are more suited to be performed in one moment and not in the other. As an example, encoding the length of a value in a certain field is an operation that has to be performed before codification takes place, because we have direct and easy access to the length of the TTCN-3 value. On the other hand, an operation like a checksum calculation has to be performed over the bitstring representation of the message. It would be inefficient to calculate it before the actual encoding takes place: we would need to perform it twice, one time for the checksum calculation and another time for the intended encode operation. The same applies for decoding.

Another important factor is the precedence of the operations. The operations should be performed in the same order that they are declared, and respecting the order of inclusions of templates into templates. In case of collision, only the most specialized definition will be applied.

```
@<codec_function> <field> := <function> [<argument>...]
```

Figure 2.34: Syntax of encoding/decoding functions

As we can see in figure 2.34, the general syntax is straightforward. The `@<codec _function>` tag precises the moment that the operation will be performed, and the possible values are the following: `@preencode`, `@predecode`, `@postencode` and `@postdecode`.

The `<field>` attribute addresses the element inside the type or template where the result of the functions is going to be stored. The function referenced by `<function>` are not TTCN-3 functions, as they are going to be executed by the generated CoDec. The

`<function>` belongs to proposed CoDec library that ought to be standardized. The reserved keyword is `auto`, which represents the automatic execution of the associated relation defined in the type. For example, if we define a `@len_dep` relation between `field_1` and `field_2`, `@preencode field_1:=auto` means that we want the length calculation to be performed and stored. `@predecode length(field_2):=auto` has a special meaning (only valid for `@dep_len` relations): the length of the field `field_2` will be taken from the encoded value bound with the `@dep_len` tag.

Other required functions are `length`, `valueof`, `checksum`. The optional presence and number of `[<argument>...]` is dependent of the function. The thorough definition of the library of required functions is beyond the scope of this work.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.35: ASCII representation of TCP header

2.5.2.6 Example: TCP simple type definition

In this brief example we intend to summarize the previous concepts. We do not intend to provide a complete definition for the TCP protocol here, but to show the concept behind the extension proposal.

TCP is standardized in the RFC793 [Pos81b], and the packet header format is shown in figure 2.35.

The TTCN-3 type with our extensions can be seen in figure 2.36. In this example we can see the meaning of the concept of "physical representation of the type". Field by field, all the parts of the field are mapped into the type, respecting the order in which they will be placed in the underlying bitstring representation. So far, the only field dependency expressed is regarding the size of the TCP header, which is expressed in a non-standard way. We will see later how to override the situation.

```
/*CD
     @len_dep dataOffset this
DC*/
type record TCPpacket {
              uint16 sourcePort,
              uint16 destinationPort,
              uint32 sequenceNumber,
              uint32 acknowledgeNumber,
               uint4 dataOffset,
               uint6 reserved,
               uint6 controlBits,
              uint16 window,
              uint16 checksum,
              uint16 urgentPointer,
        octectstring options,
        octectstring padding,
        octectstring data
}
```

Figure 2.36: Simple TTCN-3 type definition for a TCP packet with extensions

Let's now apply the encoding/decoding rules to a simple template. As we can see on figure 2.37, we can create a template for automatically filling missing information right before transmission.

This template can be parametrized to match other kind of information or used directly. We can see that `checksum` is zeroed, but we know that it will be post-encoded with the right value as it is specified by the corresponding rule. Another relevant factor to see is the way in which the default `@len_dep` relation is overridden.

2.5.3 Summary

This section presents a proposal for a methodology to ease CoDec development, not yet implemented. It is an alternative to the platform language dependent solution presented in 2.3.4.

Coding and decoding activities are not considered while test purposes and test design tasks are addressed. This methodology proposes some basic rules to apply when types are being defined and enables to express relations between fields.

This work fills a gap, where lack of standard solutions imposes time constraints or additional costs to TTCN-3 based testing. It also helps becoming independent of specific tool providers and their proprietary solutions. It is also platform language independent and does not require compiler modifications too.

The proposal has been validated to cope with identified requirements for IPv6 protocol handling. It is also possible to continue factorizing intelligence inside the CoDec, as it was suggested by being able to define properties between different messages and not only inside fields of a single one.

```
/*CD
     @preencode dataOffset := valueof(5+ceil(length(options)/4))
     @postencode checksum := ipchecksum(this)
DC*/
var template TCPpacket TCPpacket_template
                         (uint16 sport, uint16 dport, uint32 sn,
                          uint32 an, uint6 cb, uint16 win,
                          uint16 up, octectstring opt, octectstring uint8 d){
              sourcePort         := sport,
              destinationPort    := dport,
              sequenceNumber     := sn,
              acknowledgeNumber := an,
              dataOffset         := '0000'B,
              reserved           := '000000'B,
              controlBits        := cb,
              window             := win,
              checksum           := '0000'H,
              urgentPointer      := up,
              options            := opt,
              padding            := 0,
              data               := d
}
```

Figure 2.37: Simple TTCN-3 type definition for a TCP packet with extensions

2.6 Conclusion

TTCN-3 language is a unique tool for testing. It addresses the problem of providing an abstract language for test specification by removing the low level executable details from the specification. But low level details cannot be completely avoided, thus, a complex set of interfaces is defined. This set of interfaces communicate the TTCN- 3 executable with entities responsible of implementing the details left aside from the specification.

Despite the initial objective, a TTCN-3 ATS alone is not sufficient for specifying a Test System. Details must be provided. We contributed with a tool that addresses the automatic generation of this entities, currently implemented in C++ language. A second proposal is set, even though not yet implemented, to tackle the problem from a different approach: being platform language independent. We were able to show the methodological and test case design benefits of automating this activity too.

The main methodological lesson is that separating detailed executable aspects from test definition leads to specification abstraction, but it adds complexity to the process. We propose different ways to bring closer these two indivisible tasks in the process of executable test case generation. We showed that the methodological gains are worth and think that TTCN-3 must include them either in the standards or as sets of best practices or recommendations. This closes a little more the gap between abstract and executable test suites. The methodological discussion of CoDecs vs External Functions as a placeholder for test associated logic gives the test expert knowledge of when to apply different coding techniques.

Chapter 3

Automating interoperability testing execution

Program testing can be used to show the presence of bugs, but never to show their absence!
Edsger Dijkstra

This chapter deals with the work done on Interoperability testing in the field of IPv6. The first thing that should be pointed out is that this study addresses the work in the interoperability testing domain, not conformance. Existing solutions and requirements will be introduced to motivate the work. A general presentation of interoperability testing requirements and tools that were available at the time this work started is presented in Section 3.2. After the different needs are exposed, each of the building blocks used to assemble the solution are introduced in Section 3.3. The solution presented in section 3.3.1 was integrated to IRISA tool and now is a standard part of the testing process. The methodology presented in section 3.3.2 was published under the name *Plug once, test everything. Configuration management in IPv6 Interop Testing* [SV06] and presented during ATS 2006 in Fukuoka, Japan. The same methodology was also presented to the IPv6 Ready Logo Technical committee during the 9th. Tahi Event in may 2007.

The fully virtualized solution is presented in Section 3.4. Integration and configuration of building blocks is presented too. The characteristics of the tool and execution metrics are presented in Section 3.5. Some of the contents of this section are published in the article *Virtualized Interoperability Testing: Application to IPv6 Network Mobility* [SBBV07] during DSOM 2007 in San José de California, United States. No complete citation can be provided as the article was not yet published by the time of this writing.

A brief discussion on the scope and semantic of the usage of the different interfaces and operations during interoperability testing is presented in 3.6. The chapter concludes in Section 3.7.

3.1 Introduction

Interoperability testing is a pragmatical way of ascertaining that different implementations of a protocol can work together. The number of interoperability events worldwide is increasing in almost every field, from telecommunications to software. Interoperability statements or certifications in the field of telecommunications provide vendors and customers confidence that an heterogeneous group of devices will be able to operate in conjunction. It has demonstrated to be an important complement of the conformance testing discipline, which has been formalized. The acceptance and relevance of interoperability testing is growing, both in the academy and in the industry. It tackles the industrial requirement of ensuring that an heterogeneous group of implementations would work as expected. Several works like [BCKZ02, KOS$^+$00, HLSG04] address the generation of Interoperability test cases considering different factors and strategies, applied to different protocols. It is in the process of being considered a valid testing approach, and many researchers are working to formalize and automate it (see, for example [DK03, SKCK04, DV05, DV07]).

Interoperability testing requires different implementations to interwork. As a discipline, it requires the deployment of several pieces of equipment through one or more interconnected networks. The field practice is based on interconnecting implementations with the objective of ensuring that they interact properly, according to their specifications and provide the expected services. The way it addresses the verification of required properties in implementations is based on populating configurations with existing implementations and making them interwork. The observation of message exchanges and their compliance with the standards is the input used for determining if a set of implementations interoperate or not. This testing technique addresses the ultimate need of the customers: to plug components together and play. It has deep roots in Internet itself and in the Internet Community.

Interoperability testing considers different IUT during their interaction. Objectives pursued during interoperability testing are different, but in general, the first one is that it is required to validate that implementations communicate correctly. The second one is to check that they behave according to their specifications. The third one is that they provide their expected services. Among all different existing criteria for interoperability testing, the one followed by the Internet Community is to validate a non certified implementation against certified ones. In this case, when a test fails, the implementation that is considered non-interoperable (faulty) is the one being certified. Other scenarios or interoperability criteria exist, but are not being considered in this work. In spite of the criteria selected by the IPv6 Ready Logo, other definitions exist and the presented methodology and tools are applicable too.

Being able to do interoperability testing might be more complex and expensive than what initially it might look like. There is an implicit need to connect the different implementations together and to monitor all the message exchanges among them. First of all, we need to have together at the same time all the implementations that should be tested: the implementation under test and the reference ones required for verifying interoperability. Different test purposes targeting different network configurations require some physical or logical changes. It is also required to interconnect all the implementations in configurations, which are test case dependent. Configuration has to be changed efficiently and reliably during the execution of the test suite. Results of all the probes executed have to be collected and stored. State of the art solutions present automation or scalability issues.

There is a slow, but steadily ongoing process of migrating Internet from its current protocol (IPv4) [Pos81a] to the new Internet Protocol (IPv6) [DH98, NNS98, TN98]. There is an international initiative to globally develop test specifications for IPv6 new implementations. Testing requirements involve both classical Conformance and Interoperability testing. This chapter presents the work done on automating interoperability testing applied to the IPv6 domain.

Interoperability testing requirements involves the deployment of several pieces of equipment across different network configurations. When testing the IPv6 core protocols, up to three different collision domains are required. When addressing network mobility, up to five collision domains must be handled. Moreover, different hosts and routers are required during the test, each of them being a different implementation from each other. The deployment and configuration of all these elements is an error-prone activity, that requires a detailed and precise execution. Different errors exist due to the complexity of the task, errors that may bias the verdict. We can find synchronization errors, due to the execution of actions in the wrong order in some of the devices. Network configuration errors might be introduced too, due to the wrong manipulation of cables, faulty connectors or other reasons. Trickier errors can be found due to differences in the behavior of similar hardware. These last errors, which might include different cache erase policies or behavior of some commands are extremely difficult to find, and more common than expected when pieces of hardware are changed.

All forms of automation provide tangible benefits to the field. Another requirement is the ability to execute the test suites outside the laboratory, during interoperability events. Interoperability events occur several times a year in different parts of the globe. Laboratories in charge of offering testing services must be able to deploy their infrastructure reliably and run their services. When offering test services abroad, the full test system has to be deployed so as to execute interoperability test suites. The full platforms have to be transported.

The objective of the present work is to introduce a new methodological approach, based on the utilization of different virtualization technologies, to address existing problems in interoperability testing. Machine virtualization proved to be an adequate solution to converge implementations and communications services. Recently PC processor manufacturers added to their products extensions to enhance virtualization support. A new era of commodity components based virtualization is here now. The driving forces of the industry towards virtualization are server consolidation, business continuity, management and resource optimization. Machine and network virtualization together might give another meaning to consolidation.

The proposed virtualized testing platform solves several of the known problems in interoperability testing. The solution presented solves management problems, allowing us to deploy several configuration scenarios with fixed hardware configurations. The operation of the virtualized test platform not only solves technical issues that previously were only addressed with inaccurate physical manipulations, but saves resources and time. Complete testbeds involving up to seven devices and five networks, as those required for network mobility testing, can be virtualized into a single physical host. The whole platform can be converged to a single node that hosts all of the required implementations, thus offering a significative cost advantage over existing solutions. The solution also accepts hybrid configurations, with some components virtualized and some others not.

Research and results presented in this work have an inherent executable character. Practical aspects become particularly relevant when addressing interoperability test execution. Regardless the way the abstract test specification (ATS) is derived, the test has to become executable and be run. Tests cannot become executable if we do not bridge the gap existing between the ATS and the ETS.

This chapter is organized as follows. In Section 3.2 we describe general Interoperability characteristics and in particular, Internet Community relevant ones. Afterward, Section 3.3 introduces all the building blocks used for assembling the proposed solution. Section 3.4 presents and describes the solution and how it is built using previously described elements. Finally, Section 3.5 discusses the new characteristics of the solution, presents its methodological contributions and summarizes the main contributions.

3.2 A glimpse on IPv6 Interoperability Testing

In the IP field in general, and IPv6 in particular, the interoperability plays a key role, from the standardization process to industrial events. Well known events in the field are: **IPv6 PlugTests**, organized by the European Telecommunications and Standards Institute (ETSI); **IPv6 Interoperability Test Events**, organized by the TAHI group in Japan; **Connectathon** events, in the United States.

The increasing number of events addressing interoperability can be seen both in software and communication areas. Different IPv6 interoperability events takes place several times a year all over the world. They gather together telecommunication hardware manufacturers and research institutions (among others) for a few days inside big, cable-filled noisy rooms. During the interoperability sessions, different vendors connect together their switches, routers, appliances, etc. so as to determine if they can interwork together. The general case is that *ad-hoc* probes are executed. Vendors generally have their own checklist of probes for execution. They are generated by their protocol experts based on the complexity of protocol development or expected misbehaviors. Event though there is no general agreement on what tests to run, these events are a very good opportunity to interconnect implementations before they get to the market.

During Interoperability events it is also possible to find a different opportunity for performing more structured, standardized tests and apply for a certification. These tests and IPv6 certification initiative will be the subject of the next subsection.

3.2.1 Some IPv6 Ready Logo test requirements

The IPv6 Forum[1], who is committed to the promotion and development of IPv6 technologies, offers an *IPv6 certification programme.* Developing a globally unique certification programme helps to avoid confusion and transmit confidence. The objective of the IPv6 Logo programme, is to give confidence that IPv6 is available and ready to be used. The certified devices are allowed to show an “IPv6 Ready” label indicating its maturity, provided that they pass 100% of the tests executed.

The "IPv6 Ready" certification consists of series of test suites designed to address different parts of IPv6 protocol implementations. The technical requirements for current certification are publicly accessible on line in documents like [For05]. Tests are designed by experts trying to address relevant aspects that have to be fulfilled by any implementation. Tests address IPv6 Core Protocols, Network Mobility, IPsec, IKE, amongst others.

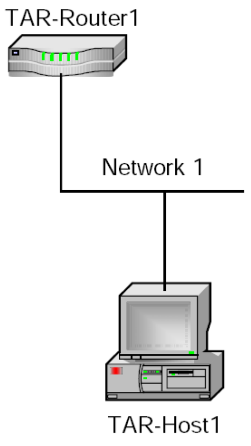


Figure 3.1: Single collision domain, two node test scenario.

To obtain the “IPv6 ready” certification, the vendor must submit a set of documents regarding the result of the execution of the tests. Tests can be performed by the vendor or with the aid of a test laboratory. The role of the test laboratory is not only to ascertain the required transparency in the certification process, but to provide the technical infrastructure and means for testing. A testing laboratory should posses not only know-how on the subject, but provide the required testbed and methodology for the test execution. All the pieces of equipment should be present, information should be gathered and collected in a way that is suitable for submitting it to the certification entity. Most of all, it is required to provide a reliable process, methodology and bullet-proof results. The test laboratory must transmit confidence when it states that an implementation passes, moreover, when it fails the developer of the device must

[1]http://www.ipv6forum.org/

comprehend and agree on the results. Maturity, technology and automation of the solution have direct impact on these subjective impressions.

3.2.1.1 General testing requirements

The requirements for IPv6 certification are varied, nevertheless some of them will be presented here. We will focus on those requirements that are addressed by this work and those required to have a good understanding of the discipline. As we are dealing with interoperability testing, a consequence is that existing implementations have to be used during the test execution. We cannot test interoperability until different implementations exist. This is different from conformance, where executable test cases can be built even before implementations exist. Moreover, we need at least two different implementations plus the IUT for each test. The implementations that are being tested or used as central elements of the probes are named *target*, while the implementations used to generate the test scenarios are named *reference*. For each test, there is a preamble in which each node (target and reference) is configured to meet test requirements. Afterward, commands are issued in every node to perform the specification required operations. All transmitted messages over the network are to be observed and recorded. During the postamble particular test configurations are undone, leaving the nodes ready for the next test of the suite. Issuing the right commands, in the right moment for every node is an intricate and error-prone task. Even though many networking commands are somehow standardized, the detailed level of configuration required makes the test execution itself a difficult task.

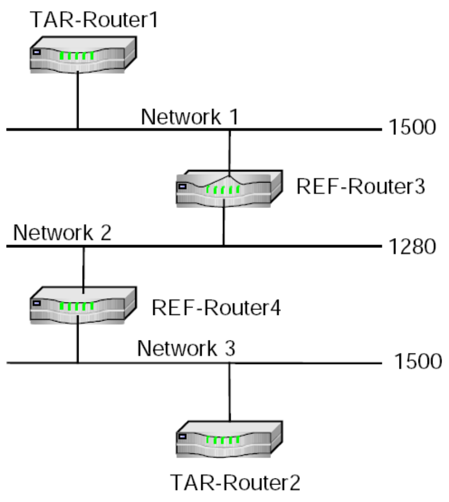


Figure 3.2: Three collision domains, four node test scenario.

Each test has an associated network topology, and elements conforming the test configuration. The tests addresses different protocol behaviors under different network topologies. Some tests require two nodes on a single Local Area Network (LAN), see Figure 3.1, to several LAN interconnected by routers, see Figure 3.2. Figures 3.1 and 3.2 were taken from [For05], and are part of the abstract test specification of some test cases corresponding to IPv6 core protocols. Other interoperability protocols have different requirements. As an example of higher complexity NEMO testing requires up to five networks and seven pieces of equipment to be handled, as seen in Figure 3.3. It is important that we devise a solution that is general enough to address all these cases.

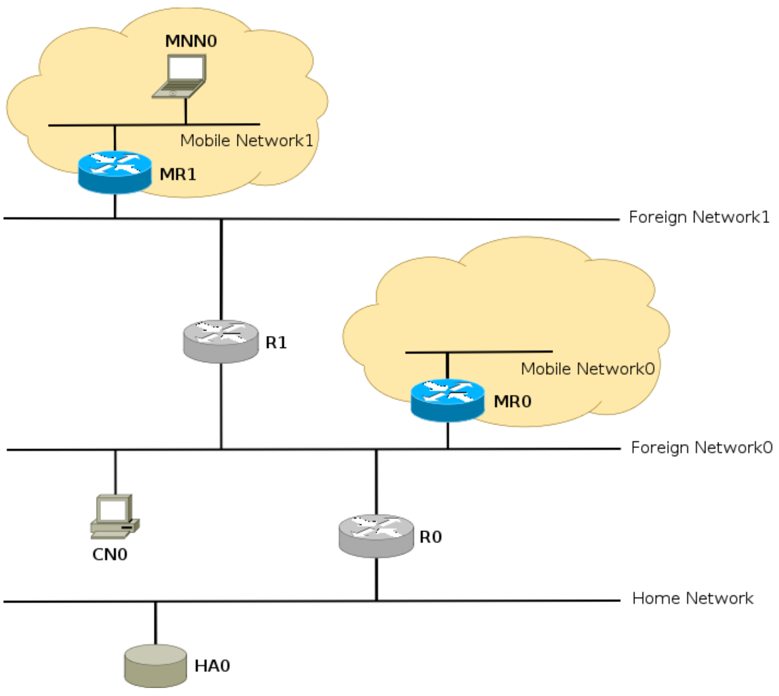


Figure 3.3: NEMO initial test network topology

Additionally, each test has to be run against at least two different implementations, thus, the number of network configurations that have to be deployed doubles the number of test layouts. This is one of the facts that turn configuration management into an important issue. All through the test suite execution, network configuration changes have to be made, together with corresponding changes in the involved nodes.

Apart from these configuration management problems, we have to take into consideration all the reference implementations required. Depending on the protocols that are being tested, it is required to use 2 to 8 reference and helper nodes. Reference nodes can take the role of hosts and/or routers and can be of any kind, from general purpose operating systems to specialized devices. Due to the large number of operations that have to be performed in each of the reference nodes, it is required to have reliable and well known hardware, together with equivalent requirements on the operating system software. Inside the laboratory, it is fairly simple to acquire required knowledge over available systems, but it becomes a hard task using unknown hardware. When participating in abroad interoperability events, it is not always possible to travel with the laboratory equipment, and the tests have to be executed over unknown combinations of hardware and software. This fact adds additional complexity during interoperability events. Ascertaining a non interoperable verdict with an unreliable platform is a very big challenge and must be avoided. Deploying a reliable test environment is not an easy task.

A part of the documentation that has to be delivered to apply for the IPv6 Ready Logo certification is a traffic capture of all message exchange performed during each test. This imposes some constraints to the way in which tests are executed and the technologies that can be used. The work [SV06] is the first publication we made for handling configuration management in this environment. Detailed attention has to be paid to traffic capturing and recording so as to gather all the required information when applying for the IPv6 Ready Logo. As the number of networks and hosts connected change, it becomes a non-trivial task too.

State of the art solutions for implementing collision domains in the field of testing use hubs, not switches. Hubs provide observability properties required for testing: packets transmitted by any node can be observed in any port allowing full observability. The roles of routers and nodes are implemented in standalone devices, usually using PC due to their capability of multiple booting. Unfortunately these legacy devices -network hubs- are harder to purchase year after year, and old hardware is stopping to work. Moreover, new pieces of hardware like NIC present unexpected behaviors under these legacy operation conditions (10 Mbps, half-duplex). Networking platform needs to be upgraded.

It is beyond the scope of this work to present all the details regarding test execution. A detailed explanation of the execution of TAHI's automated tool can be found in [End05].

3.2.2 IPv6 Mobility and Network Mobility Basic Support

Mobile IP provides an efficient roaming mechanism within the Internet. Using it, nodes can change their points of attachment to the Internet without changing their IP address. This allows to maintain transport and upper layer connections on the go. Mobile IPv6 protocol is defined by the RFC 3775 [DJA04].

The IPv6 Network Mobility (NEMO) Basic Support protocol specification can be found in the Request For Comments (RFC) 3963 [V. 05]. It is an extension to the Mobile IPv6 protocol and enables the support for the network mobility. This extension allows session continuity and reachability for every node in the Mobile Network as the network moves, not just a node moving. The protocol is designed so that network mobility is transparent to the nodes inside the Mobile Network.

The IPv6 Ready Logo [For] provides a worldwide unique certification program for NEMO Basic Support. We should concentrate on the description of the requirements for Network Mobility, and platform requirements for IP Mobility will be implicitly included. Test specification describes network topologies and test procedures to verify the correct interaction between components, Home Agent (HA) and Mobile Router (MR), as defined in the NEMO standard.

To introduce the state of the art in IPv6 NEMO Testing, a real test case from IPv6 Ready Logo Phase 2 NEMO test specifications is presented. It concerns Priority A1 Architecture of MR [IPv07], the initial test network topology is shown in Figure 3.3.

Topologically, the configuration consists of five collision domains and seven nodes. Different test cases of the suite may require other configurations. Let's assume that we are testing the Mobile Router0 (MR0). Tests are organized through scenarios, this one starts by setting up the MR0 under a Foreign Network0. After observing that the registration (Binding Update/Binding Acknowledgment) with the HA0 is well performed, MR1 and its associated Mobile Network moves to Mobile Network0. The next stage is to wait to observe the re-registration with the HA0 after the expiration of the MR0 lifetime. Then the Correspondent Node0 (CN0) must be able to ping Mobile Network Node0 (MNN0). The final steps consists of moving the resulting Nested Network (aggregation of Mobile Network) connected by MR0 from Foreign Network0 to Foreign Network1 and test again the connectivity between CN0 and MNN0 with ping command.

As in the general case, when applying for the IPv6 Ready Logo Program, traffic capture files containing all messages exchanged during each step have to be delivered to the IPv6 Ready experts. In this case, the number of networks, thus, capture files, grows. Again, each test has to be executed twice against different implementations.

3.2.2.1 Configuration considerations

The tests addresses different protocol behaviors. Sometimes from test to test, functionalities have to be enabled or disabled in the Mobile Router or Home Agent configuration file, in order to modify security restrictions or change the mode to obtain the Mobile Network Prefix. The NEMO Basic support uses IPsec to protect signaling between the Home Agent and Mobile Router. The security considerations are defined in the RFC3776 [2] [JAD04]. According to the standard, Security Policy Database (SPD) and Security Association Database (SAD) entries are defined to protect BU/BA, MPS/MPA and Payload packets between HA and MR. IPsec SA configurations between HA and MR is performed manually and can change between two tests. Automation of this task is required to provide a reliable test environment.

3.2.2.2 Classical, non-virtualized, mobility testbed description

Mobility is often associated with wireless technology. In the case of IPv6 mobility, it is a network layer mobility solution. Being a network layer solution, it is independent of the data link layer, whether it is wireless or cabled. In the general IPv6 testing field, the de-facto standard is the classical Ethernet technology. WiFi technology presents transmission errors not negligible. There are, at least, 3 orders of magnitude more of BER (Bit Error Rate) errors in WiFi connections than in Ethernet connections. Using classical Ethernet it is considered that a packet seen in an observation point also means, that it will be received by its target. When using WiFi, observing a packet does not

[2]April 2007: RFC4877 [V. 07] updates RFC3776

directly imply that the receiver of the packet will receive it. Observing a packet in a hub provides a higher confidence that the receiver will receive it too. Ethernet is used to minimize the bias in the verdict.

There are different solutions for performing mobility events during the test execution. Some solutions implement mobility events manually, by un-plugging and plugging Ethernet wires. Another solution is to connect a wireless access point to each network and change the desired communication network (i.e. ESSID) from the Mobile Components.

With a classical testbed, the aforementioned test case requires several pieces of hardware: five hubs, six PC plus the Implementation Under Test (IUT) and all required network cables. Hardware requirements makes it difficult to attend to international event carrying, deploying, configuring and validating all these equipments.

Moreover, as the number of component needed during a test campaign grows, the number of problems or errors which might happen in different steps increases too (i.e. availability of passports). The nature of these errors are varied. Maybe they are due to physical handling of devices (i.e. shocks), wrong voltages (110/220VAC, 50/60Hz, etc.) to simple aging of components. It is important to have a reliable platform during the test execution.

3.2.3 Existing interoperability tools and solutions

There are not many interoperability testing tools available. We had access to two of them during this work, both originally intended for IPv6 core protocols interoperability testing. The two tools are IRISA's tool and TAHI's one. TAHI tool is available for download while IRISA's one is an internal research and development solution, not intended for distribution.

Both tools provide different approaches to the problem. TAHI addresses the full automation of test suite execution, while original IRISA tool provides an incomplete but extremely flexible solution. We briefly describe tool strengths and the way they address the main requirements of interoperability testing. As this work uses and modifies IRISA solution, a deeper description of it is given.

3.2.3.1 IRISA interoperability testing solution

The solution developed by IRISA laboratory (referred as IRISA solution in the following) is specialized on IPv6 interoperability testing, as defined by the IPv6 Ready Logo. It addresses the automatic generation of per-node test scripts. Each of these individual scripts implements one of the Parallel Test Components (PTC) of the test case. There is a general model that abstractly implements the test cases described in the IPv6 Ready Logo programme technical documents [For05]. The distributed Abstract Test Suite (ATS) is instantiated with particular platform Implementation eXtra Information for Testing (IXIT) to produce actual Executable Test Suites (ETS). The methodology is strongly influenced by ISO/IEC 9646 [ISO94] recommendation, while adapted for interoperability testing. The generated scripts (PTC) are distributed to the nodes in an initial *deploy* phase, where all the nodes are connected together in a same LAN. This stage is prior to the whole test campaign. Nodes are connected to their corresponding networks according to the test specification and test execution is run. In the initial solution hubs were used for handling configuration management.

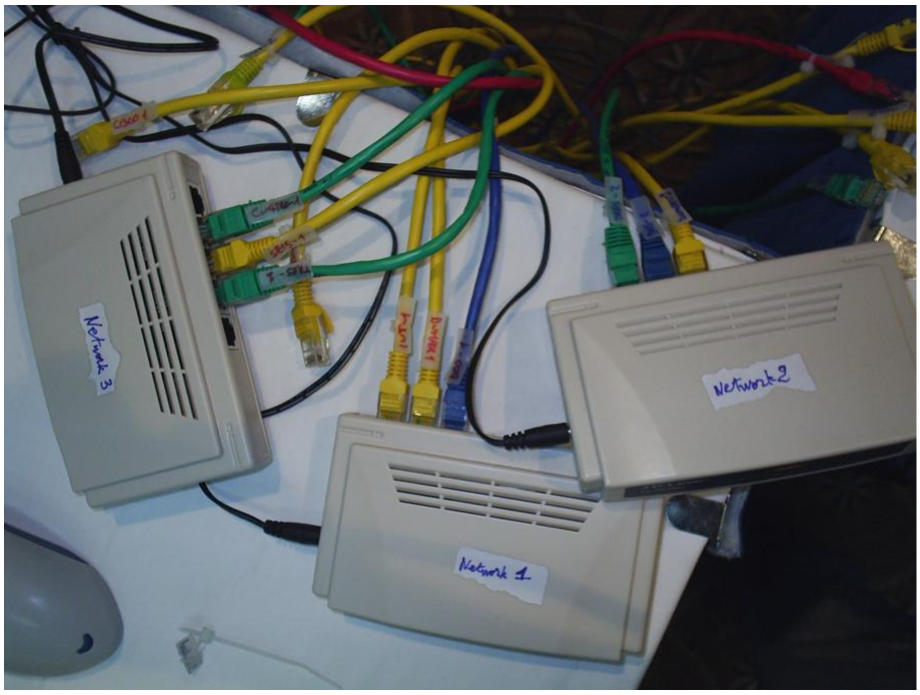


Figure 3.4: IPv6 Core Protocols, Interoperability suite Phase II, test 1.6.D network layout, FreeBSD router

Scripts are generated for each of the five nodes required for the test: two hosts, two routers and the node responsible of data gathering. The generated scripts automatizes completely all the task and the sequence in which they have to be done in every node. Internode synchronization of PTC was not addressed by the methodology, and is one of the subjects of the present work. Scripts are executed at the console in each node, starting from the initial configuration of the node. During the initialization, each node is configured as a host, as a router or as a data gathering node, according to its role, as defined in the IXIT files. The scripts are organized so as to allow simple manual dispatching of the commands. Each action is numbered, with the test identification and a sequence number inside the test. Each script waits before each action for a keystroke from the test expert and after the execution, it waits again so as to allow inspection of the observable results of the command execution through their standard and error outputs. The task of the expert is to check that all commands are executed without errors in each host, and to execute in order all the steps in each node, according to their respective sequence number.

This is a routinized task, manual task, which is indeed error prone, as it is always possible to make a mistake, executing some of the scripts out of order. This task is interleaved with the execution of manual changes to the test configuration. Task dispatching and PTC synchronization role is assigned to the test expert.

Even though node configuration and execution issues are completely automated, interconnection of nodes is still a manual task. During the preamble, test execution is paused so to allow physical intervention of the test expert. Nodes are connected manually to their corresponding networks according to the test specification and test execution is resumed. Hubs are used so as to allow traffic capturing of every transmission in every collision domain by *dumper* nodes.

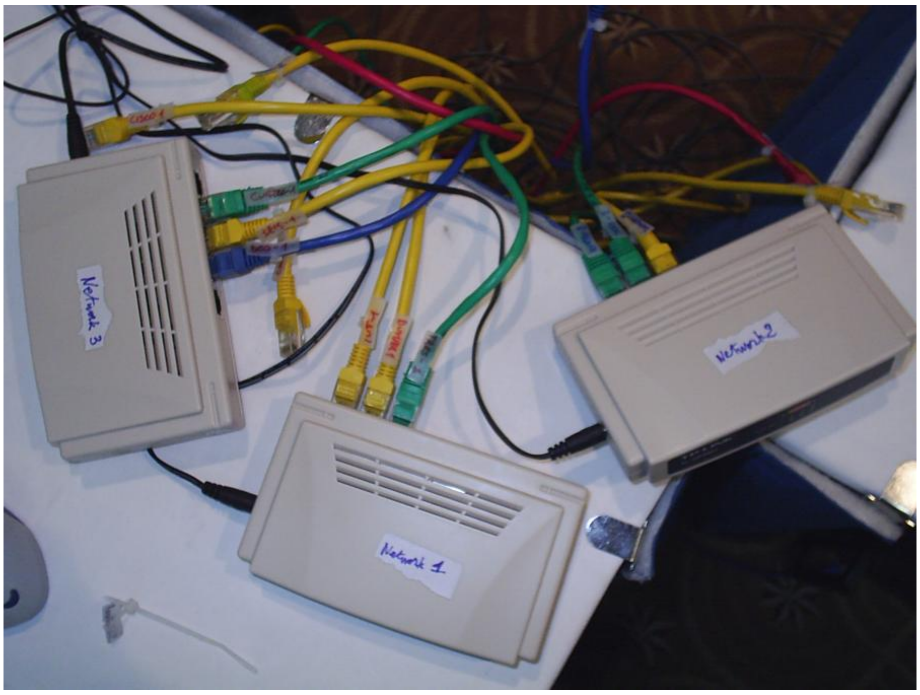


Figure 3.5: IPv6 Core Protocols, Interoperability suite Phase II, test 1.6.D network layout, GNU/Linux router

Despite our efforts to generate adequate schemas of what configuration management means in interoperability testing, we still find that the pictures shown as Figures 3.4 and 3.5 are the best way to show some of the technical problems. It is worth mentioning that the Figure 3.2 corresponds to the abstract specification of the test case shown in the Figures 3.4 and 3.5. Pictures show how three networks are interconnected for the same test in two different configurations. The selection of the pictures tries to transmit to the reader the difficulty of handling manual network configuration when just the color or tag of a cable separates right from wrong configuration. It is difficult to appreciate the subtle difference in the black-and-white printed version. Please refer to the color or

.pdf version of this document.

The test expert has to change the network topology, or configuration, between test cases during the execution of an abstract test suite. There is a inherent complexity with the handling of all required network cables to connect all different hosts. Moreover, there is always the problem of loose connectors broken jacks and other failures that directly impact the reliability of the test execution procedure. Whenever a misbehavior is found, and specially those where traffic is not observed, it is required to multiple check the test platform before issuing a *fail* verdict. IPv6 Ready Logo defines 27 test cases to be executed for a router when running IPv6 Core protocols interoperability test suite. As each test has to be executed against two different implementations at least, no less than 54 test cases have to be executed.

It is clear why this is a error prone task, even if it would be the only thing that the test expert has to perform. Some mistakes may arise during the test suite execution. Several physical connection problems might occur and the operator may plug wrong patch-cords into wrong ports after hours of work. This produces wrong test verdicts. In case of a fail verdict, everything has to be double-checked before validating the result.

Another source of complexity is data collection. Monitoring several interfaces might also introduce some additional constraints during Interoperability events. Test experts execute tests from portable computers, which do not present several expansion options. Two network cards in a portable computer does not allow to gather all the information from a single station. In some test configurations, it is required to use two different hosts to be able to capture all the traffic generated in the required networks. This fact complicates even more test configuration and data gathering.

3.2.3.2 TAHI interoperability testing solution

A completely different approach is followed by the tool developed by the TAHI Project [tah98]. TAHI distributes a set of tools named Interoperability Test Tools and scripts [TAH06]. They are highly specialized for IPv6 testing and provide a great control of the test execution. Regarding the test execution, they perform a similar task to the one described for IRISA tools in 3.2.3.1: they provide a high-level layer abstraction for the execution of network oriented commands in the host operating system. The main difference is that the solution addresses automatic configuration management. The details of such execution are omitted.

We will concentrate on tester configuration issues. The approach followed is more ambitious than the one shown in 3.2.3.1, as it also addresses the automation of configuration management. The underlying philosophy is to construct a superset of all the possible networks and enable or disable required links on a test-by-test basis. The Figure 3.6 shows the deployment of the solution. It requires a greater number of hosts (eight plus the Implementation Under Test), with several network interfaces each (up to four). This approach requires a very complex wiring scheme, making it difficult to deploy. The main drawback of the approach is that it is difficult to scale for bigger networks and host configuration details are sometimes complex.

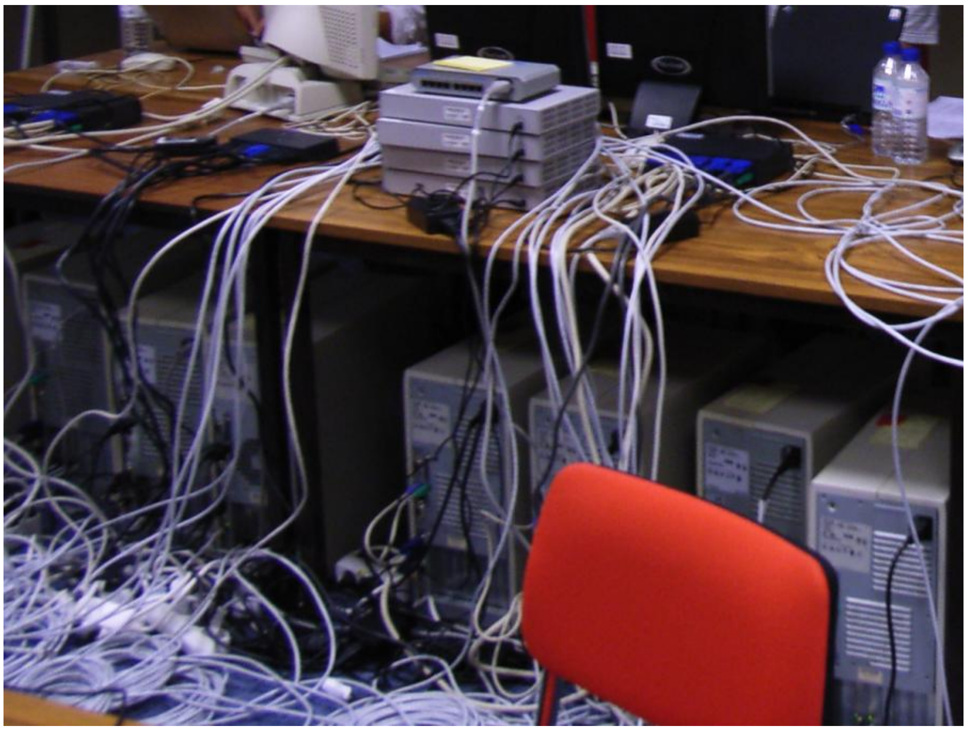


Figure 3.6: Cabling required for TAHI's solution deployment

3.2.4 Highlights

We reviewed details of the state of the art in interoperability testing in the IPv6 community. Existing solutions produce good results and level of automation regarding the execution of commands on nodes, but configuration management is addressed with different approaches. Original IRISA solution did not address configuration management automation and relays it as a manual task, while TAHI's approach is to reach full automation regardless initial deployment complexity. Existing testing methodology, and thus, the Abstract Test Specification does not derive from a formal approach, but from expert experience. This fact is criticized, but it is undeniable that it has proven results, and works, as this methodology built existing Internet.

It must be able to reliably and efficiently perform test configuration changes. It should also transmit it's reliability and maturity. Manual operations must be avoided to the maximum extent to avoid human errors and to release the test expert from routinized tasks and allow him concentrate on the implementation manipulation. Another non trivial requirement is to be able to minimize the deployment complexity to simplify its transportability, specially for international Interoperability events. Execution synchronization, configuration management and ease of deployment are the key elements addressed in our solution.

3.3 Building blocks

In this section we present all the elements required for building the new interoperability solution. Changes and their individual description are presented. The combined solution description is presented in Section 3.4, where all the building blocks are combined in a single solution.

3.3.1 Management network

The lack of an automatic test coordination procedures imposes constraints in the level of automation of the solution. The version of IRISA solution we started working with, did not address the synchronization of the different tasks across the different nodes. The operator of the test system was responsible of dispatching the actions in the right order over the nodes. The tool already addressed the automatic generation of scripts for each platform and each test, so the required step is to be able of dispatching and executing existing code automatically.

There are different choices for remotely executing commands in a host. Remote execution tools, specialized for networking include the execution of commands through serial interface. Initial IRISA solution uses this technology from TAHI tool, and could have been an option. This option is intended for routers, which provide a RS-232c configuration interface. The problem of extending this solution to address all the nodes is that it does not scale well on the number of serial ports required. That kind of hardware is already deprecated and becoming less and less common.

3.3.1.1 Network configuration: test and management networks

The modification introduced consists of adding an additional NIC to each node that allows remote execution of commands. These NICs are used to create a *management network*. This network is physically independent from all other networks used for test execution as specified on the test suite specifications. The management network must connect all the nodes of the test all the time, but must have no side effects on the results of the test execution, keep the verdict unbiased. It is fairly simple and inexpensive to add an additional NIC to hosts using USB devices, and avoiding expansion slot outage issues, specially on portable computers.

All new NIC are interconnected in an additional hub or switch. As there are no observability constraints on this network, any communication mechanism is adequate.

3.3.1.2 Remote command execution

All the modifications performed to the nodes in order to allow automation must not introduce any bias to the verdicts. Side effects introduced by the fact of enabling a new communication devices in the nodes must be avoided. To avoid conflicts of any kind with IPv6 test execution we used IPv4 addresses and disabled IPv6 services related to this management only network. Even though n-to-n connection is achieved, we will only be concerned about 1-to-n communication, from the master node, responsible of task synchronization, to each of the other nodes, the ones who take part in the test execution.

After achieving connectivity, we should provide means for remote execution of generated scripts. Tools like `rexec`, `rsh` and `ssh` are available for most platforms and provide adequate means for remote execution of commands. We concentrated and based our solution on `ssh`. OpenSSH server daemon is deployed in all nodes used for IRISA solution. To allow remote execution of commands without password-prompting we generated and distributed master node's RSA authentication keys. The master node can simply execute the command `<cmd>` in the host `<host>` simply by issuing `ssh <host> <cmd>` through the management network. This allows the complete execution of the IPv6 Ready Logo Interoperability Test Suite to be coordinated and executed from the master node.

Moreover, as there is no requirement that indicates that actions have to be dispatched simultaneously, the whole test campaign becomes a single (long) script that executes commands in all test nodes, using OpenSSH, over the management network. The way these new features were added to the existent solution managed to respect the abstraction of the tool and allows easy extension of the whole solution. A new abstract concept was introduced in the model, the *master node*, the one that controls the whole execution. The instantiation of the tool, with all required extra information for testing (IXIT) parameters produces a single meta-executable that is run on the master node. It consists of the code that should be executed locally to control the execution, and also, the remote code that must be executed in the nodes during each of the test cases.

3.3.1.3 Full controllability of involved nodes

Usage of a parallel network for test control purposes is not new, and it is used for several test methodologies. ISO9646 [ISO94] proposes a classification of them for conformance test procedures. As there is a coordination that has to follow specific procedures, this method is called a coordinated test method, a particular case of a distributed one.

Collapsing together the *master node* and the dumper (host that records all the traffic in all collision domains during the test execution) allows us to have a single system that controls the execution and gathers all relevant data. The level of controllability achieved by this methodology allows the complete remote access to all test nodes. Moreover, we achieve full observability, as we can monitor all generated network traffic in all the collision domains and complete results of command execution performed in each node. As all the activities are controlled in a single system, we avoid all synchronization problems: there is only one clock. All packets are timestamped according to the same clock.

`ssh` method for remote execution allows us to redirect standard and error output of remote commands locally. With this feature we can address specific requirements of the IPv6 Ready Logo certification procedure. As part of the documentation, it is required provide the output of command execution in the nodes involved in the test. The output redirection allows the automation of command execution output gathering too.

3.3.2 Network Virtualization

Virtualization is an old and vague term widely used since the 60'. It has been applied to many different aspects and scopes in computer science, from entire systems to services. It is used to refer the abstraction of computer resources, including making a single physical resource appear to function as multiple logical resources. The common factor of the different virtualization technologies and techniques is the hiding of technical detail through encapsulation. It allows the creation of an external interface that hides the underlying implementation. In our scope, we will concentrate on machine virtualization and network virtualization. As node is a generic term for hosts or routers in IP terminology we will use node virtualization or machine virtualization indifferently. Broader usage of virtualization will not be addressed in this work. This section deals with the usage of network virtualization to automate interoperability testing operations while Section 3.3.3 explains how we applied machine virtualization to interoperability testing.

3.3.2.1 IEEE 802.1Q: tagged VLANs

The IEEE Std 802.1Q-1998 [IEE98] is a part of a family of standards for local and metropolitan area networks. They address Physical and Data Link Layers and extend definitions and applicability of widespread Ethernet networks amongst others. Classical usage of VLANs simplifies traffic engineering, deployment and security of network infrastructure.

This work uses VLAN technology to multiplex traffic into a GNU/Linux host. We will present aspects of the standard that are to be used.

The same objective of making a single physical resource appear as multiple logical ones holds here. The IEEE 802.1Q standard [IEE98] was designed as an extension of classical Ethernet to support multiple bridged networks to share the same physical network.

3.3.2.2 VLAN design objectives

The standardization of Ethernet compatible VLANs aims to offer several benefits to this widespread ed and commodity networking technology. VLANs facilitate the easy administration of logical groups of nodes that can communicate as if they were connected in the same collision domain or link. VLANs also facilitate field activities that change the members of those groups. Traffic between VLANs is controlled, only internal VLAN traffic is forwarded automatically by bridges.

The VLANs extensions defined in 802.1Q are compatible over all IEEE 802 protocols, both on shared and point-to-point LAN. They should maintain compatibility with existing equipment as far as possible, communicating nodes in logical groups as if they were connected in the same collision domain or link.

3.3.2.3 VLAN implementation details

Virtual LANs are required to span over different Layer 2 network devices. Layer 2 frames had to be extended to carry an identification of their originating VLAN, the VLANID, which is inserted after the source address in the packet header. Additional information, like priority, is also added to the frame header. In the Figure 3.7, the format of a standard IEEE 802.3 Ethernet II frame is shown and its corresponding VLAN-tagged version. The additional VLAN information is inserted after the source address.

In order to maintain compatibility with existing equipment, VLAN-enabled switches and bridges must be able to accept standard Ethernet packets and new VLAN packet format. We will distinguish equipment that can handle VLAN-tagged frames referring to them as VLAN-aware in contrast to VLAN-unaware devices, which cannot handle tagged traffic.

In our proposal, we use a VLAN-aware network switch. As said before, it must be capable to handle both tagged and untagged traffic. The links used for connecting legacy unaware devices are known as Access Links. An Access Link is a LAN segment used to multiplex one or more VLAN-unaware devices into a port of a VLAN Bridge. All frames on an Access Link carry no VLANID.

The standard also defines Trunk Links as LAN segments used for multiplexing VLANs between VLAN Bridges. All the devices that connect to a Trunk Link must be VLAN-aware.

Untagged traffic of an Access Link that enters the switch is treated according to the VLAN membership rules defined in the device. Even though implementation is vendor dependent, it can be thought that an incoming untagged-frame will be tagged with its corresponding VLAN-tag and afterward will be treated as a tagged frame. The

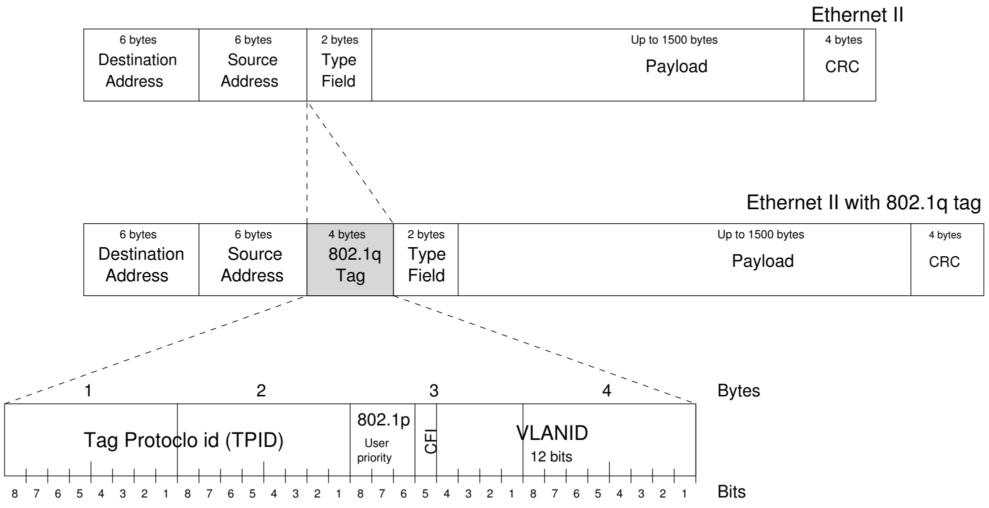


Figure 3.7: 802.1Q Tagged VLAN frame format

opposite happens when a tagged frame is to be transmitted on an Access Link: the tag is stripped, the frame converted into an untagged one and then it is transmitted.

Vendors provide different set of policies for tagging Access Link traffic. VLAN membership can be defined based on network addresses, network protocols, MAC addresses, etc. We will only use port based VLANs on Access Links.

3.3.2.4 GNU/Linux support for VLANs and bridges

VLAN support was added to GNU/Linux soon after IEEE 802.1Q Std. was published. The initial work was done by Alex Zeffertt, which was ported to current kernel versions and maintained by Ben Greear. Detailed information about their effort and GNU/Linux VLAN capabilities can be found in [Gre05].

At the beginning Linux kernel had to be patched. Since kernel 2.4.14 was released in March 2002, VLAN support entered mainstream Linux distribution. Current Linux vanilla kernels are VLAN-aware, if configured properly. There exist some MTU problems for specific Network Interface Cards (NIC), but as VLAN support is independent from the actual NIC, it is possible to use it with almost any hardware configuration.

The kernel module required for handling VLANs is called `8021q` and can be inserted simply by issuing a `modprobe 8021q` command, provided the required permissions. The userspace command to handle Linux VLAN capabilities is `vconfig`. As an example, the command to add the VLAN with the VLANID 10 to the interface eth0 is `vconfig add eth0 10`. The Figure 3.8 shows the list of interfaces available to the GNU/Linux host after virtual interfaces on VLANs 10 and 11 were added.

The relation between the physical interface (eth0) and the virtual one that uses it is clear as they share the same name prefix name and MAC address. There is a single

```
odie:~ # ip link list
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:15:c5:c0:c3:07 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:18:de:94:b9:cb brd ff:ff:ff:ff:ff:ff
4: sit0: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
5: eth0.10: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 00:15:c5:c0:c3:07 brd ff:ff:ff:ff:ff:ff
6: eth0.11: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 00:15:c5:c0:c3:07 brd ff:ff:ff:ff:ff:ff
```

Figure 3.8: Standard and VLAN interfaces in GNU/Linux

transmission queue attached to the physical interface, but apart from these details, the virtual interface can be handled as a regular interface.

GNU/Linux networking options do not only cover host or router related issues, but it can address lower layer operations, like bridges. Before Ethernet switches become commodity components, PC based software bridges were deployed to segment networks. This functionality is part of the Linux kernel since the very beginning of Linux networking. Bridge capabilities were kept updated and support bridging even with VLAN interfaces too, not only physical ones. This feature allows us to communicate different VLAN using the GNU/Linux kernel capabilities and still having full controllability on the traffic.

To have bridge support it is required to load the Kernel module `bridge`. The userspace command to handle bridge operations is `brctl`. Arbitrary number of software bridges can be created on a Linux host and can get any interface associated to them. Figure 3.9 shows how to create a bridge named `Network1` and associate the VLANs with VLANID 10 and 11 to it. Nodes on VLAN 10 and 11 now can get communicated through the software bridge.

```
odie:~ # brctl addbr Network1
odie:~ # brctl addif Network1 eth0.10
odie:~ # brctl addif Network1 eth0.11
```

Figure 3.9: Software bridge creation and handling

Access to the bridge interface from Linux is indistinguishable from other interfaces, thus, traffic can be captured and observed in a standard way. We can observe all the traffic that traverses the bridge as we do when sniffing a hub. We achieve full observability and automation of bridge configuration.

3.3.3 Machine virtualization

The main goal behind machine virtualization is to abstract physical hardware implementation from its physical implementation. It allows to deploy a virtual machine over more than one physical system for high availability purposes, or, to collapse a set of virtual machines into a single physical computer. The driving forces for current virtualization deployments are based on the fact that most servers operate at less than 15 percent capacity. Virtualization allows consolidation of workloads, making a single physical system host multiple virtual ones. It allows decoupling software needs from hardware needs, as it is possible to deploy new servers and services without the purchase of hardware, reusing available resources. We can see in Figure 3.10 a schematic representation of several virtual machines running over a virtual platform on a single system. Figure 3.15 shows a screen-shot of the implemented platform, consisting of several virtual operating systems running on virtual hardware, over a physical system. Our objective is to use virtualization to cut the explosion of nodes required for interoperability testing.

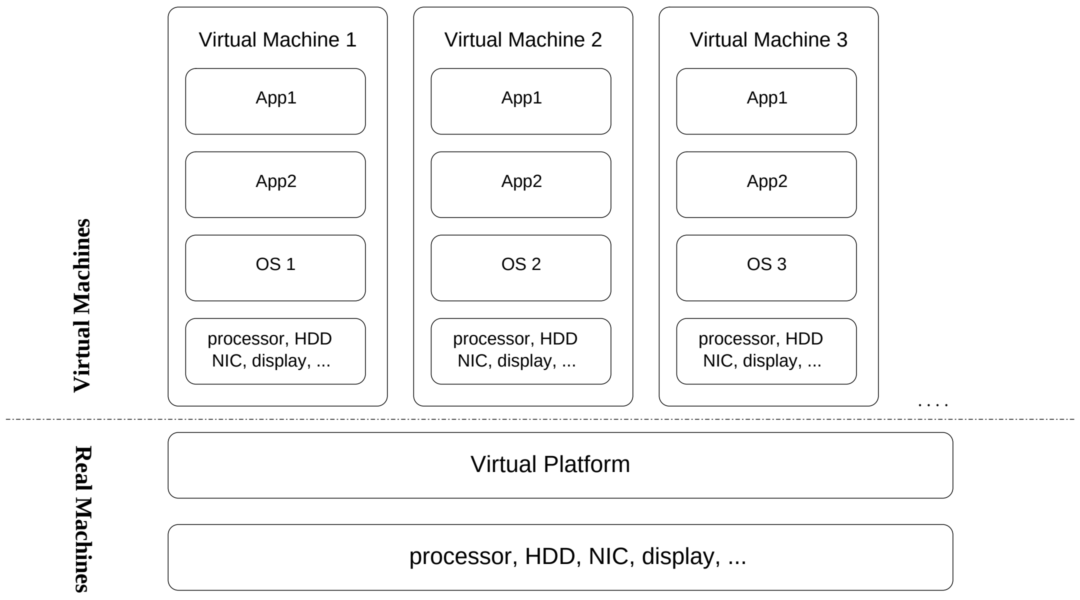


Figure 3.10: General virtualized platform

Virtualization took a new momentum in the industry with the latest developments in PC microprocessor industry. Even though the IA-32 processor architecture does not fully meet Popek-Goldberg requirements for virtualization [PG74] several software packages overcome the limitation and managed to provide virtualization solutions on the PC. Implemented solutions include techniques like dynamic recompilation of privileged code that makes use of processor's unprivileged instructions. Both major manufacturers of PC processors, AMD and Intel, have included hardware support for virtualization. AMD's AMD-V technology [Zei06] and Intel's IVT [NSL$^+$06], even though not compatible, provide required hardware support to run an unmodified guest operating system without the need to emulate significant parts of the hardware. Even though virtualization products exists for i32 architecture since 1999, it is these new changes in hardware capabilities what enables powerful virtualization to take place.

It is important to distinguish the **host** system and the **guest** one. The host system is responsible to create a simulated computer environment for the guest system. Host virtualization techniques address different levels of abstraction of the underlying hardware, and get different specialized names. What could be considered the lowest level of virtualization considered for our purpose is called *paravirtualization*. It consists of a host system that does not simulate hardware, but offers a special API that can only be used by modifying the guest OS. As we intend to replace only hardware, this is not a feasible option. It was considered due to availability of freely available solutions, like Xen [3]. The other relevant issue that prevent us to adopt this technology is that we need to use four different OS implementations, and current Xen only addresses Linux as a guest OS.

The other extreme is the *emulation* of a complete hardware platform, including even the emulation of the CPU. This solution indeed allows an unmodified guest OS run on a host. Moreover, it allows even mix-and-match of architectures and system types, as it is possible to emulate a little endian guest architecture on a big endian host or vice versa. Emulators like Bochs [4], that run on Windows, Linux, AIX, etc. supports unmodified systems like Windows, xBSD, Linux, etc. The principal drawback is the performance.

A sort of intermediate solution is called *native virtualization*, and is the one used in this work. Native virtualization addresses the same challenge as the emulation but without emulating the CPU. The rest of the hardware is emulated, but it uses the host CPU. The native virtualization solution selected is the VMWare Server [Inc07], which is distributed closed source and free of charge. Due to legal issues, parts of the platform that directly link to the Linux Kernel, even though some parts of the code is open. The technology delivers processor performance levels similar to the same OS running on the raw hardware.

Our goal applying virtualization is to collapse m virtual hosts into n physical ones without introducing any bias in the verdict. We expect $m > n$, and preferably, $n = 1$. This allows very simple deploy of test configurations and high levels of automation in configuration management issues.

3.3.3.1 Virtualization and networking

Despite all the shine of virtualization, we need a solution that is flexible, observable and unbiased from the networking point of view. As we are testing network protocols, we must be very cautious about network virtualization and the possible consequences on the test verdict. We concentrate our discussion on VMware, the virtualization solution used during this study.

[3]`http://www.xensource.com`

[4]`http://bochs.sourceforge.net`

VMware offers an adequate deal of options regarding networking. It allows up to four virtual network interfaces to be defined on a per-virtual machine basis. The way the network interfaces are made visible to the guest system is based on the emulation of a well known AMD PCnet32 (lance) network specification. For Linux, Solaris and BSD operating systems, the standard unmodified driver is used. In the case of Windows XP, a special VMware driver is used, even though, distributed in the standard Windows XP driver suite.

The way networking options are incorporated to the virtual machine is at a very low level, physical emulation. Same kernel modules (drivers) and networking in general are used in a virtual machine as in real PC using a lance NIC. No modifications are performed on the kernel or special drivers are required. We are running on a virtualized environment the same software that we would run on raw hardware. This fact allows us to rely (as much as we would do over real hardware) on the subject of our test.

Off-the-shelf networking options fall in three different categories: NAT, used to share host's IP address; Host-only where a private network is shared with the host; Bridged, connected directly to the physical network. At a first glance, none of this options are adequate for testing. If we use NAT, we are modifying the packets, something we do not want to happen. Host only networks are internal to the host where the VMware server is running, thus, we cannot communicate them to external implementations. The most suitable option is bridged networking.

The main problem arises due to the availability of NICs. If we virtualize two routers, we would need 4 different NICs for the working interfaces, plus two more for the management network. As we keep on adding hosts, we also grow on the number of NICs required. We have to use network virtualization too.

One fact that has to be considered is that there are some limitations on the way and moment that networking is done. Networking options can only be changed at virtual server configuration time. Changing the binding of a virtual NIC to a physical NIC can only be done with all virtual machines powered off. Some changes even require to rebuild (re-compile) kernel modules, removing them and inserting them back in the host OS. The solution must be kept as simple and practical as possible, trying to avoid these manipulations.

3.3.3.2 Virtualized machines using VLANs

The logical conclusion for avoiding physical interfaces is to use virtual ones. As shown in 3.3.2.4, we can create as much virtual interfaces as we need over a single physical interface. The issue behind this idea is that VMware does not allow virtual interfaces to be specified for networking. If virtual interfaces are specified, no communication occurs after the guest OS boots. Inspection of log files shows an error specifying that virtual interfaces cannot be used for bridging.

Another solution would be to define virtual interfaces inside the guest OS. This was tested using also GNU/Linux as guest and host OS without problems, but the option is not available for all required operating systems. As an example, on Windows XP, tagged VLAN support is a driver dependent and not an OS dependent issue. The

VMware provided driver does not support VLAN tagging. This would lead us to non standard solutions.

Additionally, this approach presents methodological implications. If our solution was based on handle VLANs inside the guest OS, the one we are either testing or using as a reference implementation, we would changing the data link layer used. It must be kept in mind that the solution presented in Section 3.3.2 is transparent to all nodes. VLAN tagging and tag removal is done either by the switch or the GNU/Linux host, but transparently to the nodes that take part of the test. For them, untagged packets are transmitted and received. They cannot differentiate the original hub-based solution from the one using network virtualization.

Before giving up, we decided to analyze the reason why VMware fails using virtual interfaces for bridging. Due to GPL/LGPL license issues, VMware distribution for GNU/Linux is not completely closed source. Kernel modules must be distributed open source too, thus, we have access to the source code of the bridge networking module. After analyzing the problem we found out that there is no particular reason for virtual interfaces cannot to work. From our understanding of the source code, they try to bridge Ethernet traffic into Ethernet physical interfaces. We think that the reason is to avoid users to send Ethernet frames over PPP links or tun devices, but the case of virtual interfaces is not considered.

After several approaches and modifications, we designed a patch that accepts standard Ethernet and 802.1Q frame formats. The patch is shown in Figure 3.11. Initial versions of the patch addressed handling of VLAN information through different places of the bridge module, but we finally realized that we can skip all processing and simply accept header lengths corresponding to 802.1Q headers. What the patch does is to include VLAN headers in the include section of the source. Afterward, in the checking of the device associated header size, we keep the checking for standard Ethernet header size (12 bytes represented by the `ETH_LEN` constant). We replaced the part of the code that rejected other possible values for a conditional clause that checks and accepts header lengths corresponding to VLAN tagged packets (comparing to the constant `VLAN_ETH_HLEN`). The result is in the style of minimalism and simple. With the application of this patch to the corresponding bridge source, unmodified virtualized Ethernet interfaces can be mapped to virtual Ethernet interfaces on the GNU/Linux host OS. The tagging process occurs automatically, transparently to the guest OS.

It seems that this is an unforeseen combined usage of these virtualization techniques. Stability of the bridge code through the last versions of the VMware Server shows that this code is not evolving, and we would like to see off-the-shelf VLAN usage on virtualization solutions.

```
--- ../../vmware-config2/vmnet-only/bridge.c    2006-08-10 00:59:13.000000000 +0200
+++ bridge.c    2006-12-17 13:40:50.000000000 +0100
@@ -18,6 +18,7 @@

 #include <linux/netdevice.h>
 #include <linux/etherdevice.h>
+#include <linux/if_vlan.h>
 #include <linux/mm.h>
 #include <linux/skbuff.h>
 #include <linux/sockios.h>
@@ -820,11 +821,16 @@
     */

    if (bridge->dev->hard_header_len != ETH_HLEN) {
-      LOG(1, (KERN_DEBUG "bridge-%s: can't bridge with %s, bad header length %d",
-             bridge->name, bridge->dev->name, bridge->dev->hard_header_len));
-      dev_unlock_list();
-      retval = -EINVAL;
-      goto out;
+       if (bridge->dev->hard_header_len == VLAN_ETH_HLEN) {
+             LOG(1, (KERN_DEBUG "bridge-%s: %s, assuming 801.1q device",
+                     bridge->name, bridge->dev->name));
+       } else {
+             LOG(1, (KERN_DEBUG "bridge-%s: can't bridge with %s, bad header length %d",
+                     bridge->name, bridge->dev->name, bridge->dev->hard_header_len));
+             dev_unlock_list();
+             retval = -EINVAL;
+             goto out;
+       }
    }

    /*
```

Figure 3.11: VMware patch to support virtual interfaces

3.4 Usage of virtualized nodes on the interoperability test platform

The building blocks used for the solution were already presented in the previous section. Chronologically, the first problem addressed in this work was the one related to configuration management for physically implemented nodes. The way this problem was solved was by replacing hub-based collision domains with a combination of VLANs and software bridges. This solution was implemented together with a virtualized version of the management network. The solution is described in the Subsection 3.4.1.

The next step was to address machine virtualization, as a way to cut the explosion of hosts and networks required for interoperability testing beyond core protocol testing. Mixing ant matching real and virtualized implementations provides a great deal of flexibility, easing not only deployment, but management of the test system. Moreover, if all implementations are virtualized, we only need the host system to have a full interoperability testing platform. We can even get rid of the VLAN aware switch. Virtualized solution is presented in 3.4.2. A description of the full solution summarizes current section.

3.4.1 VLAN based configuration management

The objective is to replace legacy Ethernet network devices (hubs) with VLAN-aware ones. The solution does not introduce any change into reference or target nodes used for interoperability testing, allowing us to completely reuse existing and validated methodology. We would like to find a solution that can handle simultaneously real and virtualized nodes during the test execution. This section describes the configuration and setup of Tester's network.

There are several reasons for replacing hubs. The first of them is that they are very old. Even though they are a good, solid and reliable technology, nobody uses them anymore. It is not only that nobody uses, but very few sell them. Each time it is more difficult to find a provider that still sells hubs, and we need to find a solution before all the second hand equipment we use finally wears out. Another important reason is that new hardware (i.e. network interface cards) shows unexpected delays or packet losses working in half-duplex mode. This observation has been discussed and confirmed with network administrators. We take this fact as an indication that hardware manufacturers stopped paying attention to the 10Mbps, half-duplex Ethernet network. We need to avoid bias in our testing tool. Last, but not least, the newest Ethernet technologies only work in switches, not hubs. The 100MB Ethernet collapsed the 2Km network radius of the 10Mbps one into 200m, but it was not the path followed by the designers of Gigabit Ethernet. Instead of shrinking the network to 20m, their decision was to avoid collision domains and build a fully switched network, without the option of hubs. Clearly, 10GigabitEthernet will not collapse the network to 2 meters. Collision domains are not available in some physical mediums, like fiber optics.

It is clear that we need to find an updated solution. We describe the solution, which is based on tagged VLANs and a GNU/Linux host.

3.4.1.1 Switch configuration

In general, switches are not used for network testing in the fields of conformance and interoperability testing. The reason is that traffic is relayed only to the intended destination, making it difficult to see *all* the traffic that is being generated. The initial naïve solution to use one VLAN per test network and changing the configuration from the switch holds that problem. The solution introduces a problem of observability, as it is not possible to monitor all the traffic of the collision domain in a standard way. Some vendors provide *mirror* or *monitor* ports. Commands for configuring the VLANs are not standard, and have to be adapted to different switch vendors. To avoid that problem a different approach is followed: VLANs are statically allocated during all test execution.

We will plug every reference and target implementation of the test directly to the switch. Hosts (reference or target) will use only one port, as they only have one network connection. Routers (reference or target) will use two ports on the switch. Ports that connect implementations will be configured as Access Links and each port will receive a different VLANID. As we connect every implementation directly to the switch, there is only one node interface per link. We are using the switch for tagging packets from different interfaces of different implementations with distinct and known tags. Identifying the tag, we can know the traffic origin.

All the port based VLANs must have a “way out” of the switch to allow analysis and interconnection. We will use a Trunk Link for this purpose, all previous VLANs will be combined into a Trunk Link. We perform this task using the switch so as to avoid making modifications on reference and target implementations used. It may be possible to tag traffic from the source and use Trunk Links instead of Access Links, but we would have changed the configuration of the nodes. The objective of this configuration is to only replace the intercommunication hardware, but to leave untouched the implementations.

A set of ports are kept together in another VLAN reserved for management. We should use this VLAN to implement the management network described in Subsection 3.3.1. The management VLAN does not hold any particular requirement regarding observability, just to provide connection. The VLANID must be reserved and not used for any other purpose.

Most of VLAN-aware switches provide several value added features. It is important to disable all features that would introduce traffic like Spanning Tree Protocol on the links. We want to avoid noise in our test scenarios to keep capture files clean and easier to analyze. Another features that should be disabled is priority or QoS tagging that might affect transmitted traffic.

For our purposes, the switch configuration is static all the time. In this way we avoid introducing in our scripts proprietary command sequences. We can use any 802.1Q compliant switch, provided that it is configured according to our needs. The solution does not introduce any requirement on the configuration options of the switch. Web-based configuration, console or command-line are suitable for our solution. We use the switch as a *traffic tagger* and to multiplex all the traffic into a Trunk Link. The solution was successfully deployed using diverse VLAN aware switches. We used 3Com, Foundry

Networks and Hewlett Packard ones. Despite the diversity of the configuration options, the solution is straightforward to configure and deploy.

3.4.1.2 GNU/Linux configuration

The GNU/Linux host (Lh) introduced is not part of the test, and has to be thought as part of the network infrastructure. The Lh is connected to the Trunk Link, therefore, it has to be configured to be VLAN-aware. As shown in 3.3.2.4, it is required to load the `8021q` kernel module to support IEEE 802.1Q traffic. The way to access the traffic of any certain VLAN from the Lh is to configure a *virtual* interface inside that VLAN. Afterward, the interface can be handled as a standard network interface in GNU/Linux. It is possible to attach a packet sniffer, add a network interface or perform any (software) operation on the virtual interface.

Traffic generated from the Lh and transmitted into the Trunk Link contains the corresponding tag. The switch receives the frame and determines the physical port to be forwarded according to the information provided by the VLANID and the destination MAC address. As the switch is configured to have only two ports in each VLAN (the Access and the Trunk), and the traffic came from the Trunk, it will be forwarded through the other port. The tag is stripped and it is transmitted as a regular Ethernet packet.

A virtual interface must be placed in the management VLANID. This would allow to implement all the remote command issuing on the nodes.

We are able to plug (virtually) as many network interfaces as required to our GNU/Linux host and connect directly each interface of the reference/target nodes to them. We have emulated a Linux host with up to 4094 Ethernet ports and the reference/target nodes directly connected to it.

3.4.1.3 Configuration management

What follows is done by software inside the GNU/Linux host, which can be fully automated. According to the test being executed, the nodes and network topology is known beforehand. The Lh will not consider any of the virtual interfaces that are not being used during the test execution. As all network interfaces used during the test are assigned to different VLANID, none of them is capable of communicating with each other only with the described configuration. Communication takes place due to the definition of software bridges in the GNU/Linux host.

The Lh will bridge the traffic of the hosts directly connected into a same network according to the test purposes, and will emulate as many networks as required. We have full control of the traffic that reaches every host, as they cannot communicate to each other through the switch, but only through the Lh. This gives us an excellent place where to capture and store all the traffic required for documentation purposes. As we are emulating the physical links (collision domains) inside the Lh, we can also monitor and log all the traffic.

All required changes during the test execution can be performed without any kind of physical interaction. We only control how the packets are bridged between VLANs.

Configuration management becomes a VLAN management issue, adding and removing VLANs to them. All the dynamic operations of physically connecting different ports can be mapped to software configuration operations, and automated inside the GNU/Linux host. Connecting two implementations consists of enabling bridging operations between their corresponding VLAN, regardless of the fact that the implementations run on physical hardware connected through the switch, or, that they are virtualized inside the GNU/Linux host.

3.4.1.4 Mobility events

Mobility events can be treated in the same way as configuration changes described in 3.4.1.3. Configuration management operations, instead of happening only during pre or postamble, are interleaved during test execution. Test cases will include operations that will be executed on the GNU/Linux host and correspond to mobility events. The VLAN corresponding to the network interface of a Mobile Node can be changed from one software bridge to another. Equivalently, one network interface of a Mobile Router can be changed from one software bridge to another, implementing the mobile event.

3.4.2 Virtualized addition of nodes

The need for machine virtualization did not appear in the laboratory, but traveling to interoperability events and providing test services abroad. In the laboratory there is enough time to know the hardware, master the solution and debug detailed hardware/software problems. When the testbed has to be moved to another country problems start. You have to carry as little equipment as possible.

3.4.2.1 Need for virtualized nodes

The first option is to ask the responsible of the organization of the event to provide you with the required hardware. Despite the quality of the hardware provided, there are always problems. Maybe the network cards do not have the same revision as those you use and the behavior of the protocol is different as the one expected. We found several of these problems related to buffers, caches and internal tables that affect protocol observable behavior. During some tests you are required to flush caches to force neighbor discovery solicitation, and a minor change in the hardware makes you face an unknown problem during a test execution, with a customer.

Another option is that you ask the organizer 4 network hubs, and thinking that they are making you a favor, they provide you 4 network switches. Or even worse, 10/100 hubs. 10/100 hubs, even though they are called hubs, implement a bridge between the 10Mbps collision domain and the 100Mbps one. If due to link speed negotiation machines get connected to different collision domains, the tester will get into observability problems. You will see multicast traffic, but maybe not unicast responses, filtered by the bridge. This might introduce bias in your verdict unless you suspect problems in your platform. But you must rely on the platform to be able to run tests successfully.

Assuming that you would carry a complete platform with you -as we did several times- you will only carry what is minimally required. Five hubs, 15 NICs and 7 computers is enough. It is difficult to explain this at the customs and it is very risky to damage (or forget) something. After you get to the place, deploy the platform and are ready to test, you are informed about a last minute request from an implementation that you did not consider. Let's say that people from Kernel.org want to test their new IPv6 support. Then, that computer you carried with Linux, is tested, you know how it works and you rely on becomes useless and your platform needs a new implementation.

Assuming that the number of requests for interoperability testing grows and you do not have enough time to test them all with your platform, then you need a second platform to run tests in parallel. Either you carry two platforms or you return to the problems discussed before.

Virtualization of machines comes as a way of bringing more, carrying less.

3.4.2.2 Configuring a virtualized machine

Here we present how to configure a virtual machine to be integrated in the platform. We will not stop to present all virtualization capabilities and benefits in this work, but those relevant to interoperability testing. We concentrated our work on VMware due to its maturity, capabilities, prior experience and for being a proven virtualization solution.

As presented in 3.3.3.2, standard VMware Server distribution does not support bridging traffic over virtual Linux interfaces. This is a critical need for us, as the configuration management solution we developed is based on the usage of VLAN traffic. We need to integrate virtualized hosts seamlessly to our solution. With the patch presented in Figure 3.11 we enabled bridging into virtual interfaces.

Now that we have a solution that can communicate over VLANs, everything we developed in the IRISA tool can be applied to virtual machines.

For virtualized nodes, each virtual NIC is bridged to its corresponding VLAN inside the GNU/Linux host. It is important to precise that the mappings must be performed over the same physical interface of the GNU/Linux host where the VLAN aware switch is connected. In this way, mapping of VLANs is consistent across physical and virtualized implementations. This fact, and the ability to mix and match physical and virtualized implementations enhances the applicability of this solution. A virtual machine can be communicated transparently with any other machine (virtual or physical) that is connected to that VLAN or through the software bridges configured.

Separate Virtual Machines are configured for each of the hosts required according to the test specification. Different operating systems or network stacks must be used to meet test case requirements. We configured virtual machines with Linux, Solaris, FreeBSD, OpenBSD, Kame, etc. To have a comfortable working environment, each virtual machine is allocated 256 MB of RAM. It allows the execution of state of the art *nix environments with adequate speed for IPv6 related configuration and command execution. Deploying 5 virtual machines on a single node is possibly with commodity PCs.

There are few requirements regarding the software that needs to be installed in each host, basically, IPv6 networking tools. Each of the virtual machines is configured with the required number of virtual network interfaces, and each interface is mapped to a different VLAN in the Linux host OS. This mapping is static, and cannot be modified during the test execution. Being more precise, it is required to stop VMware server and recompile it to change the mapping of a network device (`/dev/vmnet`) to a particular virtual interface. All virtual machines are configured to have enough interfaces to play any configuration in any test, either as a host or router. They are allocated 2 networking interfaces and a management one. Unused interfaces remain disabled in test cases that do not use them.

3.5 Characteristics of the virtualized interoperability tool

This section describes and presents the main results found during the implementation and usage of the virtualized solution. Presented results validate practically the results of field application. Moreover, it shows the usefulness of providing higher level of abstraction to concrete, network level, configuration management operations.

3.5.1 Implementation and execution

All of the different aspects of the tool were implemented and combined in the IRISA laboratories. Different stages of the evolution of the tool, and the complete one, were also presented during recent Plugtest and IPv6 Ready Logo Interoperability events. Some parts of the presented methodology are already accepted and applied by the Internet Community, while some others are yet to be approved for certification purposes.

Figure 3.12 shows a deployment of the solution, addressing automation of the test suite execution, but not taking profit of any virtualization technology. The deployment corresponds to a platform corresponding to IPv6 core protocols interoperability testing.

The solution consists of one node (on the right) that manages the execution and performs all traffic capturing. It is connected to all networks: it issues all the commands to the remote nodes through the management network; it observes all the traffic on the testing networks. The three test networks are implemented using 3 hubs, where nodes are plugged manually. The fourth network is the one used for management. It does not require any kind of handling during the test suite execution. The four nodes on the left are the target and reference ones used during test execution. No manipulation needs to be performed on their consoles, as all the commands are automated except on the IUT, which remains unknown.

Figure 3.13 shows a complete solution deployment, based on physical nodes while implementing network virtualization.

The master node is not directly observable on the picture, but counts just one physical cable that connects it to the switch (under the screen). The cable transmits 9 different VLAN into the GNU/Linux host, where software bridges are used for automating configuration management. The cable density is higher, as all required cables enter a single device, but conceptually it is the same number of cables from the switch to the platform. Usage of both solutions is the same. All commands are dispatched from

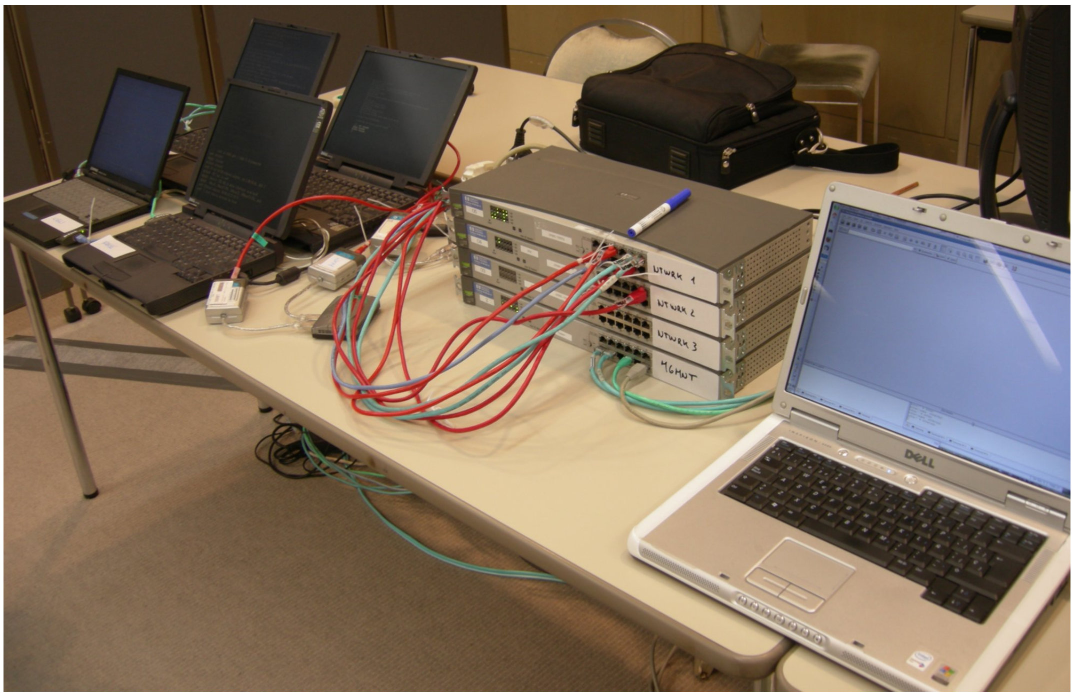


Figure 3.12: Fully deployed platform, addressing only automation

the master node through the management network (real or virtual) to the nodes. The difference are pre and postambles, where there is no need for manual manipulations. Cables are plugged at the beginning of the event in the platform and no further physical manipulation is required.

What remains is a fully virtualized solution. Figure 3.14 shows the physical aspect of the virtualized solution. The node on the right implements the complete platform, hosting the 9 required VLAN and 4 virtual machines of the platform. The VLAN aware switch is not required anymore, as the number of interface cards can be reached simply using USB NICs. The host on the right plays the role of the IUT.

The complete platform was virtualized into a single host, achieving the maximum level of virtualization we can think of. The consoles of the different virtual machines can be seen in Figure 3.15. A screen-shot shows the four consoles of the different systems deployed. Four implementations of different *nix systems with different IPv6 stacks are run: Linux, USAGI, Solaris and FreeBSD. No manipulation has to be performed on any of the systems through the execution, as it was the case with the two prior solutions. The test is run from the same console, with the same options and without any particular modification.

We can take even one step further. The platform also allows us to virtualize the IUT, something we also did. We successfully virtualized the IUT and managed to execute the platform and the IUT inside the same host. In that case, there is nothing tangible,

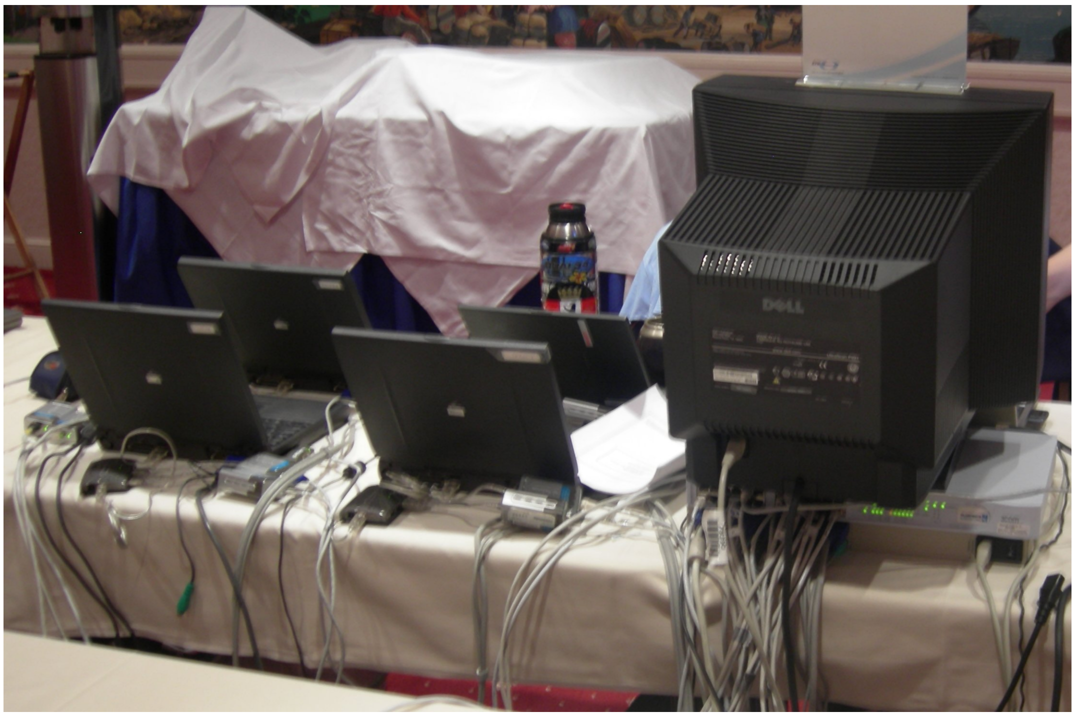


Figure 3.13: Fully deployed platform, using physical nodes and virtualized configuration management

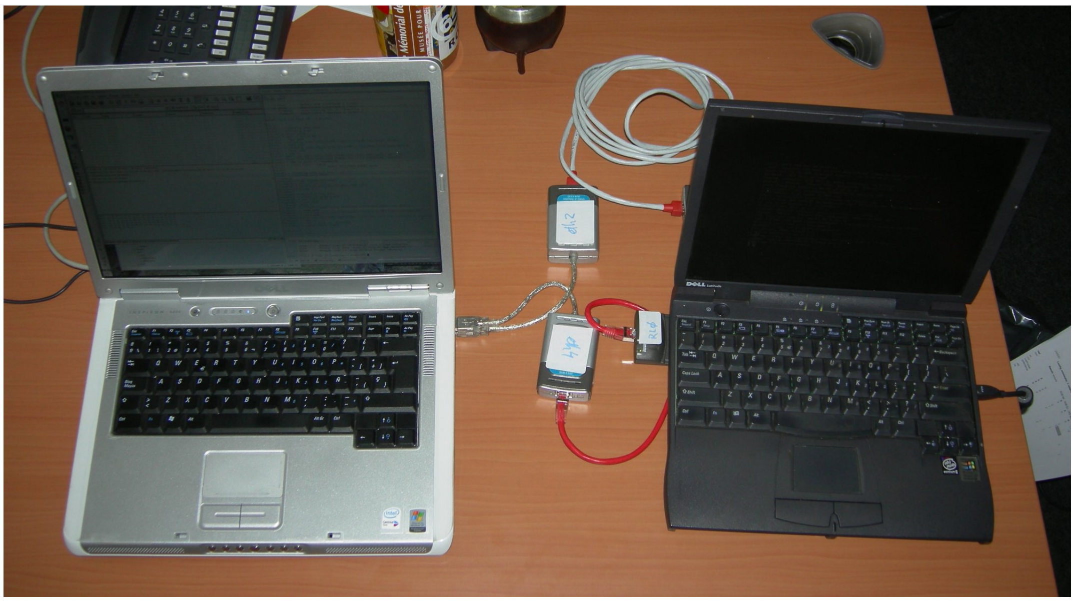
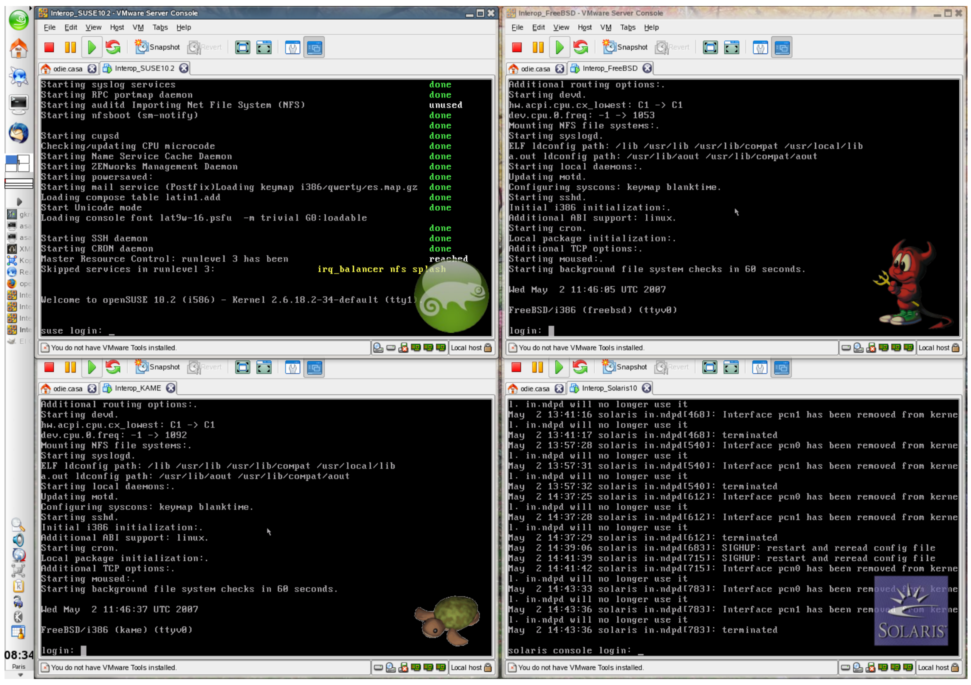


Figure 3.14: Fully deployed platform, network and machine virtualization

but the host system that runs the test platform, virtualizes network connections and all hosts.


Interop_SUSE10.2 - VMware Server Console
File Edit View Host VM Tabs Help
Snapshot Revert
odie.casa Interop_SUSE10.2
You do not have VMware Tools installed.
Local host
Interop_FreeBSD - VMware Server Console
Interop_FreeBSD
Interop_KAME
Interop_Solaris10
SOLARIS
08:34
Paris

```
Starting syslog services                                     done
Starting RPC portmap daemon                                  done
Starting auditd Importing Net File System (NFS)              unused
Starting nfsboot (sm-notify)                                 done
                                                             done
Starting cupsd                                               done
Checking/updating CPU microcode                              done
Starting Name Service Cache Daemon                           done
Starting ZENworks Management Daemon                          done
Starting powersaved:                                         done
Starting mail service (Postfix)Loading keymap i386/qwerty/es.map.gz done
Loading compose table latin1.add                             done
Start Unicode mode                                           done
Loading console font lat9w-16.psfu  -m trivial G0:loadable
                                                             done
Starting SSH daemon                                          done
Starting CRON daemon                                         done
Master Resource Control: runlevel 3 has been                 reached
Skipped services in runlevel 3:                  irq_balancer nfs splash


Welcome to openSUSE 10.2 (i586) - Kernel 2.6.18.2-34-default (tty1).


suse login: _
```

```
Additional routing options:.
Starting devd.
hw.acpi.cpu.cx_lowest: C1 -> C1
dev.cpu.0.freq: -1 -> 1053
Mounting NFS file systems:.
Starting syslogd.
ELF ldconfig path: /lib /usr/lib /usr/lib/compat /usr/local/lib
a.out ldconfig path: /usr/lib/aout /usr/lib/compat/aout
Starting local daemons:.
Updating motd.
Configuring syscons: keymap blanktime.
Starting sshd.
Initial i386 initialization:.
Additional ABI support: linux.
Starting cron.
Local package initialization:.
Additional TCP options:.
Starting moused:.
Starting background file system checks in 60 seconds.

Wed May  2 11:46:05 UTC 2007

FreeBSD/i386 (freebsd) (ttyv0)

login:
```

```
Additional routing options:.
Starting devd.
hw.acpi.cpu.cx_lowest: C1 -> C1
dev.cpu.0.freq: -1 -> 1092
Mounting NFS file systems:.
Starting syslogd.
ELF ldconfig path: /lib /usr/lib /usr/lib/compat /usr/local/lib
a.out ldconfig path: /usr/lib/aout /usr/lib/compat/aout
Starting local daemons:.
Updating motd.
Configuring syscons: keymap blanktime.
Starting sshd.
Initial i386 initialization:.
Additional ABI support: linux.
Starting cron.
Local package initialization:.
Additional TCP options:.
Starting moused:.
Starting background file system checks in 60 seconds.

Wed May  2 11:46:37 UTC 2007

FreeBSD/i386 (kame) (ttyv0)

login:
```

```
l. in.ndpd will no longer use it
May  2 13:41:16 solaris in.ndpd[468]: Interface pcn1 has been removed from kerne
l. in.ndpd will no longer use it
May  2 13:41:17 solaris in.ndpd[468]: terminated
May  2 13:57:28 solaris in.ndpd[540]: Interface pcn0 has been removed from kerne
l. in.ndpd will no longer use it
May  2 13:57:31 solaris in.ndpd[540]: Interface pcn1 has been removed from kerne
l. in.ndpd will no longer use it
May  2 13:57:32 solaris in.ndpd[540]: terminated
May  2 14:37:25 solaris in.ndpd[612]: Interface pcn0 has been removed from kerne
l. in.ndpd will no longer use it
May  2 14:37:28 solaris in.ndpd[612]: Interface pcn1 has been removed from kerne
l. in.ndpd will no longer use it
May  2 14:37:29 solaris in.ndpd[612]: terminated
May  2 14:39:06 solaris in.ndpd[683]: SIGHUP: restart and reread config file
May  2 14:41:39 solaris in.ndpd[715]: SIGHUP: restart and reread config file
May  2 14:41:42 solaris in.ndpd[715]: Interface pcn0 has been removed from kerne
l. in.ndpd will no longer use it
May  2 14:43:33 solaris in.ndpd[783]: Interface pcn0 has been removed from kerne
l. in.ndpd will no longer use it
May  2 14:43:36 solaris in.ndpd[783]: Interface pcn1 has been removed from kerne
l. in.ndpd will no longer use it
May  2 14:43:36 solaris in.ndpd[783]: terminated

solaris console login: _
```

Figure 3.15: Virtualized console view

All the forms of virtualization used during the different steps and stages were introduced transparently to the test platforms. In any of the previous cases, the IUT receives the same set of stimuli and the responses are processed in the same way. From the IUT point of view, all the changes introduced are transparent. From the testing point of view, all the procedures used to generate the message exchanges are the same.

Behavior of any platform and deploy option was consistent. Verdicts of the test execution on the same IUT are consistent too. We can validate each of the virtual machines using a classical platform. Afterward all the tested virtual machines can be used as an option for interoperability testing. We can use a completely tested platform running on a single system.

3.5.2 New platform capabilities

The described set of solutions and techniques enhances mix and match possibilities for building an interoperability test system. Presented solution provides permanent and full access to all nodes and full observability of all the collision domains. It allows automation of different test execution tasks like traffic recording or gathering of IXIT information automatically. The activity of the test expert has become simpler with this methodology, as the whole activity can be described as: "pressing enter and observing that expected traffic is generated in each network". Informally we call the master node *the enter machine*, because it is the only operation performed there all test long.

3.5.2.1 Solution convergence

As a result of the joint application of presented tools and methodology, we managed to collapse the complete control and observation activities to a single node. This fact not only provides means for automatization of the execution, but releases the test expert from performing unnecessary tasks and allows him to concentrate on important issues during the test execution. This diminishes the risk of having a biased verdict due to wrong order of actions during test execution.

As a proven methodology, the test expert takes from granted that network changes took place the way they should, the platform adjusted to the required configuration, and can concentrate on the IUT currently being tested. It is in the end, his ultimate responsibility there.

3.5.2.2 Observability spin-off: synchronization

Collapsing control and observation in a single system also solves synchronization issues, as all the control can be referenced to a single clock. There is more to be said on this issue.

Changes performed due to network management issues also introduce changes regarding observability. When collision domains were built based on hubs, it was possible that two different messages were transmitted simultaneously on different collision domains. Observation of different sources requires synchronization, which cannot be guaranteed. As hubs are replaced with VLAN aware switches, collision domains disappear. Every interface is connected directly to a full-duplex switch port. Thus, no collisions occur at physical layer. Even though two packets might get transmitted at the same time, they will be buffered, ordered and transmitted in a certain order. From that point on, what is observed at the Network Layer level is ordered and the order is preserved in the switch. Packet rate generated during test execution is very low, and is far from generating buffer overflows in state-of-the-art switches. This means that traffic generated will not be lost.

Every packet coming from every node is transmitted through a unique link to the master node. Even though it is possible that two nodes transmit packets simultaneously, they will be ordered in the switch and transmitted serially to the master node. Inside the master node there are no collisions as packets arrive serially. After serialized, they are bridged serially too and forwarded back to the switch. From the observability point of view, all packets will arrive sequentially to the bridges and they will be forwarded sequentially, preserving the incoming order. We have no control on the order that simultaneous packets will get serialized, but we can ensure that after that, there is no data loss and we can ensure serialized behavior without ambiguities from there on.

On the other hand, this generates a hardware limit to the data rate that can be handled with this technology. It can never be faster than the link that connects the master node to the switch. For state of the art interoperability tests in our field, it proved to be enough. For testing other properties like stress test or performance test, it might not be enough.

3.5.2.3 Host virtualization benefits

It is also worth mentioning that the usage of host virtualization is a significant benefit from several points of view. The fact that a single host can condense a complete interoperability test suite has the direct and obvious benefit of cost reduction.

Some of the key benefits can be found when platform commuting is considered. Attending to an international event carrying 5 computers and a VLAN aware switch is not simple. Having all nodes X-Rayed at airports is not always simple, and might lead to enhanced security checks. Another possibility to avoid the transportation is to ask the organizer to provide the required hardware. The option is not simple, as even slight changes in hardware configuration have direct impact on the platform and how the tests are executed. The behavior of the platform might change unexpectedly with minor hardware changes. Failures observed range from failures in cache operations (unable to clean Neighbor Discovery tables between tests) to complete node hangs, forcing not only to reboot, but to restart the test case. Failures like cache operations might introduce bias in the verdict, as observable results might lead to absence of expected messages. It is important to rely on the testbed while executing a test. Configuring a test system based on foreign hardware and unknown combinations does not lead to a platform you can blindly rely on. It takes several hours of work and some complete test suite execution to rely on a test system.

The capability to integrate physical and virtual hardware also provides a great deal of functionality. A solution that only counts with virtual node testing is adequate for operating system testing, but might not be adequate for testing arbitrary implementations. With the ability of connecting our implementations through VLAN during the test execution, we can apply this methodology also to VLAN aware implementations. It would be possible to test devices independently from the physical medium. Current limitations of the technology is 10/100 Half-Duplex, but we can move to anything that can be switched into Ethernet with VLAN tagging. We can test the capabilities of a router connecting a single cable to one of it ports and working over different VLAN, interoperating with virtual machines. Similar modifications as those we added to VMware can be performed on other emulators and move testing to the same prototyping stages before product implementation.

Other spin-off of virtual hardware is its homogeneity. The virtual machine abstraction offers the same services and behaves equally over any deploy. The abstraction provided isolates hardware details and allowed us to re-deploy virtual machines over different hardware, without finding biased results. The same virtual machine was moved from AMD to Intel processors, single to dual core without changes in the observed behavior.

Another interesting factor is the reduction of the time it requires to build new test platforms. It is as fast as copying the virtual machines from one host to the other and solving some minor identification details. Also backup and snapshot operations are simple and useful. Before performing maintenance on the virtual nodes, they can be backed up or snap-shoot-ed. If any problem is introduced, it is easily rolled back to the previous version.

3.5.3 Field application results

This section exposes figures from field experience together with laboratory results. Automation of interoperability execution is a must. It presents all the complex signaling required for network protocol testing, plus, distribution management requirements of distributed testing. It has been reported that manual execution of IPv6 Ready Logo Interoperability test suite requires more than a man-week.

Virtualization technology used delivers performance levels similar to the same OS running on the raw hardware. Even though it is reported that there is a noticeable performance impact on disk and network access on VMware Server, it proved adequate for our purposes. In our experience, Unix based operating systems, like FreeBSD, Solaris and Linux have very low CPU requirements after booted. An idle Unix running on VMware introduces almost no overhead on the host system, thus, very low side effects are propagated to other virtual machines sharing the same physical host. All of our experiments are consistent, and no bias was introduced in any test case by the usage of virtualization.

Our tests on bridged networking characteristics were successful. A virtualized OS bridged over a network interface is indistinguishable from a physically deployed one. Full conformance and interoperability IPv6 Ready Logo tests suites were passed by the virtualized hosts.

3.5.3.1 Field error analysis

Manual operations are still required, and might always be required when there is no possibility to automate the IUT. The platform used for testing consists of several hubs, in which the different nodes are plugged test after test. Using IPv6 Ready Logo accepted technologies and practices, we studied the source of errors encountered during test execution in the field.

Errors found were classified in five different categories. The first class, *synchronization* errors, counts the frequency of mistakes in the sequence of events, mainly between the tool and the IUT. Whenever out of order execution is performed, the test have to be re-executed. The class *network configuration* counts wrong connection of test elements to networks during test suite execution. *Non standard IUT configuration* counts the number of errors found during the execution due to configuration parameters of the implementation found during the execution. They are not errors, and can be tuned to meet test requirements, but forces test case re-execution. *Single errors of the IUT* counts observations of wrong behavior, but that dissappear in subsequent executions of the test. They do not correspond to tunable parameters and could lead to fail verdicts if misbehavior holds. The last class correspond to effective *IUT flaws* detected.

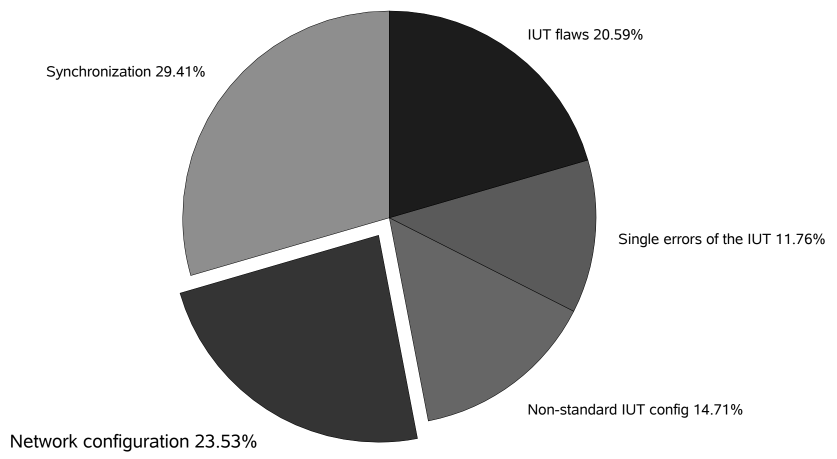


Figure 3.16: Distribution of execution errors

The Figure 3.16 shows graphically a comparison of the distribution of errors found during test execution. It is important to recall that the ideal situation would be that 100% of the errors encountered correspond to IUT flaws and no other errors might be present.

Presented methodology completely removes network configuration errors, which account for almost one quarter of the field errors. In our laboratory, where we also able to automate the execution script for the IUT, this solution fully solves synchronization task, disappearing the source of errors too.

3.5.3.2 Execution time

Interoperability testing is a time consuming activity, and requires the permanent participation of a test expert. Figures presented here were gathered during real interoperability events and in laboratory. The addition of the management network, solved the synchronization overhead and errors amongst the nodes of the test platform. The first solution was manually synchronized, even though automated and average times for test suite execution exceeded four hours -no time execution data was gathered with that tool-. With the addition of the parallel network, average execution times shrank to 2:50hs. In spite of that, some test executions might take up to 4:55hs. This is due to complexity of operation of the IUT or presence of unforseen problems. Due to this fact, when interoperability events are scheduled only two devices per day are scheduled, per test platform and test expert.

In laboratory executions, where conditions are better controlled, execution time averages 2:05hs.

3.5.3.3 Cost and reliability

State of the art solution and methodology accepted by the IPv6 Ready Logo involves up to 6 test nodes and the test manager node. Five different networks are required for test execution and a sixth for management, thus, 6 network hubs are required. Presented solution can be completely collapsed inside a single physical computer, avoiding all network complexity and using standard Ethernet interfaces to connect to the IUT. Without trying to put a monetary value to hardware involved, hubs dissappear and only a single physical computer is required. State of the art hardware can handle the whole workload, thus the saving ratios can be estimated between 5:1 to 10:1.

Reliability of the collapsed testbed is drastically enhanced. Let's consider the platform requirements for the testing platform used for addressing IPv6 Mobility. Initial solution requires that 6 nodes, 6 hubs, the test manager and lots of cables were 100% operational. Let's also assume that the probability of failure of the nodes (p_n) is the same amongst them. The same hypothesis is considered for the probability of failure of hubs (p_h). The probability of an operative physically deployed platform can be estimated with $(1-p_n)^7.(1-p_h)^6$.

Let's now consider the virtualized, collapsed platform that requires a single node. It is direct to see that the probability of having an operative platform is $(1-p_n)$, which is the probability that the single node required for running the collapsed platform is operative.

It is straightforward that $(1-p_n) \geq (1-p_n)^7.(1-p_h)^6$, what shows the enhanced reliability of the collapsed testing platform. If we consider physical operations with cables, reliability is even more degraded. Reliability becomes more relevant in international interoperability events, where the complete platform has to be commuted. Transporting seven notebooks might sometimes lead to a broken node, leading to a non operational platform. Care of a single host can be achieved more easily, and a second backup equipment can be transported, easily doubling the chances of having a single operational testbed.

3.5.3.4 Quantitative tool execution changes

We quantify the different contributions performed to the interoperability tool in table 3.17, particularly, the one used for IPv6 Core Protocol testing. By the time of this writing, equivalent results are being obtained for the mobility one, but not 100% of the test cases are implemented. The decision is to present definitive and mature results. The different metrics used are objective enhancements that can be used to distinguish the different aspects addressed at different stages. It is straightforward to see the different aspects addressed at each stage, what changed and the impact in the tool operation.

We measure the number of computers used, assuming that the deployed platform only uses operating systems deployed on computers that can be virtualized. If we use hardware-bound devices, then there is a limit for virtualization, but the rest of the analysis remains equivalent. The number of PCs count the number of real, physical, systems required to deploy the solution, regardless the fact if they conform the platform or if it is the management node. The number of cables just count the number of Ethernet cables required to cable the complete solution. Hubs/switch counts the number of network active devices required to communicate all the elements. Synchronization describes the means for ordering the distribution of the tasks according to the test specification. Connections count the optimal number of configuration management manipulations that have to be performed during a complete interoperability test suite.

	Original	Management network	Management + VLANS	Fully virtualized
PC	5 + IUT	5 + IUT	5 + IUT	1 + IUT
cables	12	18	15	2
Hubs/switches	3	4	1	0
Synchronization	manual	auto	auto	auto
Configuration operations	45	45	0	0

Figure 3.17: Metrics on the evolution of IRISA tool

We cross these inputs with four different platform deployments, those described in 3.5.1. The first one corresponds to the existing IRISA tool taken at the beginning of this work. The second column corresponds to the addition of the management network, that solves synchronization issues and automates the dispatch of commands across the nodes. The next column corresponds to the solution built using physical nodes but replacing classical networking infrastructure with a VLAN based infrastructure. The last column corresponds to the fully virtualized solution, where only the IUT is physically implemented. We assume that we are testing a router for counting the number of configuration operations.

We can see that the platform shrunk, manual operations disappeared and networking devices become unnecessary too.

3.6 What do we test when we test interoperability, short discussion on definition limits

After several years of seeing the application of testing on the field, subtle, but yet important questions arise. Different notions of interoperability testing exist. Despite the fact of which one we select, a common characteristic is that we do not have control over the lower layer, we can only observe messages exchanged there, without controlling what is being transmitted. Figure 3.18 shows architectural components of a test system and possible points of control and observation on a two implementation interoperability scenario.

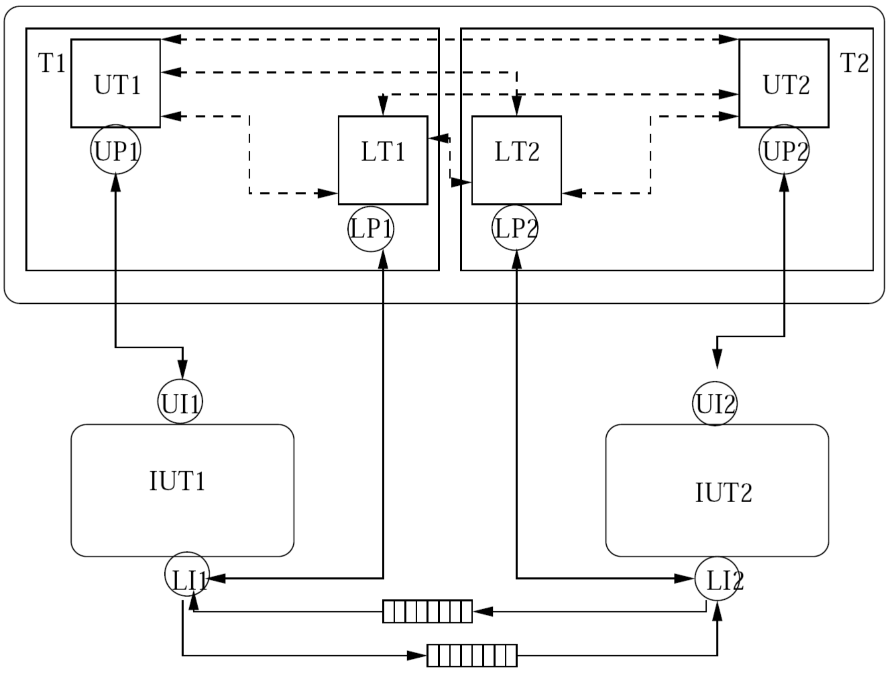


Figure 3.18: Two IUT interoperability architecture

The test system is conceptually divided into two different parts, T1 who controls and observes IUT1 and T2 who is responsible for IUT2. Every tester T contains entities that control the upper and lower layers of the IUT being tested. The entities may or may not be coordinated, as shown by the dashed-lines. Each IUT is controlled through their upper interfaces and observed on their lower ones. This architecture can be extended to multiple IUT following the same principles: lower interfaces can only be observed, not controlled and upper interfaces are those we use for controlling the implementations. Based on this philosophy interoperability test suites are designed and defined.

There is a problem regarding the abstraction of the concept of "upper interface". What do we mean by "upper interface"? Who, using which privileges, should be the user of that interface? At first glance it does not seem very important, but we should take a closer approach to this issue.

General interoperability test requirements concentrate on the lower interface traffic, and do not specify the upper interface requirements. It is a reasonable and logical requirement, as it is the interface where peer entities interoperate, and where the protocol we are testing is specified. Most of the times the upper interface, specially when it is not providing a standardized service, is not standard, completely implementation dependent. Moreover, new devices developed over an embedded general purpose operating system do require a better definition of what services of the upper interfaces could be used to control the implementation and which ones could not.

Different privileges exist for different users of the device. In some cases, the vendor can bypass the upper interface where the implementation customer is confined. Backdoors or simply extended command line access might allow the vendor to execute and have access to commands and options which are not available to the final user. Is it right to ascertain a *interoperable-pass* verdict when the customer is not able to repeat the test? This scenario is pretty common on embedded systems, where an application is running on top of general purpose operating systems. Let's say that the application offers a web user interface and it is still possible to get shell access to the host operating system. Let's also assume that the host operating system is fully interoperable with other implementations, according to certain criteria, and that all required operations could be performed through the command line interface provided through the shell access. Which one should be the verdict if some required operations were not published through the web interface, but could be accessed -by the vendor- through the command line interface?

We think that upper interfaces used for interoperability testing must be those effectively exported to the end user of the system. In this way we can ascertain that whichever verdict issue, it could be repeated by the ultimate user of the implementation under test. It is important that in the end the customer is able to execute all the tests.

Another possible problem is regarding the semantic of the operations allowed on the upper interface of the implementation. In the solution presented during this chapter all operations performed on the hosts consist of execution of network configuration commands on each host. As we have full access to general purpose operating systems, we could do even more. We could remotely order the node to just transmit a packet that we crafted (in a conformance-like approach), without using the protocol implemented by the device. It corresponds to the functional characteristics represented on Figure 3.18 but not to the concept behind interoperability testing: we want to test real implementations to interoperate, no artificially crafted actions.

Interoperability definitions must address these facts to be able to reflect more accurately the concept addressed.

3.7 Conclusions

We have presented methodological, technical and implementation achievements in interoperability testing execution. Interoperability testing automation simplifies and reduces the bias due to configuration management manipulation faults and manual synchronization of tasks in the tester. Usage of VLAN based switching allows the testing to get rid of legacy hub-based technologies and scale the number of hosts that can be handled one or two orders of magnitude. This fact would allow us to even go beyond known formally derived test case requirements. The synchronization of the full test system that removes causal dependency bias solves even more problems that just automation addresses.

Experimental results support the applicability of the solution. Deployment, maintenance, transportation and cost of the tester were positively affected. From 25 to 50%

of execution errors are removed, making testing process smoother and more reliable.

The ultimate most extreme application of network and host virtualization allows to collapse a platform of several nodes and collision domains into a single system. With this level of automation and virtualization, we provide abstract operations over the full interoperability platform that can be used by formally derived test specifications. Formal interoperability testing has the tools required for mapping abstract operations into executable ones.

Chapter 4

Conclusions

Making predictions is hard, especially about the future.
Niels Bohr

This chapter concludes the thesis. As previous chapters already presented their specific conclusions, we will concentrate of what we consider the three core contributions of the thesis as a whole. Hints and ideas of future work are suggested too.

4.1 Contribution summary

In this thesis we address the problems found turning an abstract test specifications into an executable ones. This is a problem disregarded by many researchers, but that has a deep impact in the practice. As it is the usual case when there is a distance, it is possible to start working from both ends. We did so. In the Chapter 2 we presented our results taking TTCN-3 language as an abstract test specification language and solving field issues. Starting from the execution side we also provided solutions to existing problems, and the results were presented in Chapter 3. Even though relevant solutions were built on both sides there is more to be done. Moreover, both sides are not static, but they are moving. Executable requirements are growing, as protocols implemented by devices are more complex day after day. Abstract requirements are broader too, as the level of abstraction required is always higher, and execution details are removed from the specification.

TTCN-3 based methodology and framework

TTCN-3 is great for abstract specification of test suites, but more work has to be done on the language. The expressiveness and power of the language itself for precisely describing abstract test cases and behavior is not matched by platform adaptation and communication related entities. The language does not provide the developer required tools for agile test case development, making it difficult and a time consuming activity to apply it. TTCN-3 standard does not specify or place requirements except on the TTCN-3 Runtime Interface and TTCN-3 Control Interface. The result is a lack of portability of the test cases between tools and a difficult design decision for the test developer. Either you choose a tool that provides off-the-shelf components that ease the development, but you end up tied to a single provider, or you develop yourself the whole set of components you need, raising the amount of time required for the development of the solution. As shown in Chapter 2 we developed and distributed a tool to ease not only coding-decoding code generation but to achieve portability of test suites among TTCN-3 C++ tools. Automation in the coding-decoding code generation isolates the test expert from manual development issues. Automation provides a higher level environment, allowing faster development cycles. The methodology also provides independence of the platform. Adaptation tasks are automated in the tool.

The core contribution of this thesis work on the TTCN-3 field propose solutions to the problem of turning a TTCN-3 abstract test suites into executable test suites. Two different approaches were proposed. They are based on the fact that there is a tight correlation between abstract specification of the messages interchanged with the IUT and their codification. This relationship can be exploited and automated approaches can take profit of this fact and provide methodological benefits to the test specification lifecycle. Despite the high initial effort of developing the tools, the benefits of these techniques are worth the time invested, as shown by the metrics. The test expert does not need to manually develop coding-decoding extensions from scratch or use tool dependent, non standard, extensions of the language. It is only required to follow a methodology for type definition, that is augmented with platform language pieces of code named *CoDets*. Tool independence has been achieved. ATS, CoDet and external function reuse become simpler and more effective.

The results of this field of work were presented in different conferences as [SFRV05, SBFV05, SBV06, SBD$^+$06, SCV07]. Results seem to be interesting to the TTCN-3 user community, as a Best Presentation Award was obtained in 2005 and different works were accepted three years in a row in TTCN-3 User Conferences. The special issue on The Evolution of TTCN-3 of the Software Tools for Technology Transfer journal confirmed the acceptance of the article *Automatic CoDec generation to reduce test engineering cost*. The IPv6 Test Toolkit was registered with the number IDDN.FR.001.030006.001.S.A.2006.000.10600 by the Agence por la Protection des Programmes.

Internet Protocol testing

Results of our work were not only presented to TTCN-3 and academic testing community, but also to the Internet Community. Different vendors benefited from our testing services, using TTCN-3 developed test suites. We showed to the Internet Community that TTCN-3 is a viable option for testing Internet Protocols, despite the fact of being a general purpose testing tool.

The research done on the executable side, mostly on interoperability testing of Internet Protocols, is presented in Chapter 3. We produced and showed tangible, concrete results, as shown by indicators in Table 3.17. The complexity of assembly, deployment and operation of a full interoperability test bed collapsed into a single computer. It is difficult to say if complexity of the solution grew or shrank, but indeed, manual operations were minimized. Complete test case execution is automated and collapsed into a single system. From the physical point of view, interoperability testing now requires the same number of pieces of hardware than conformance testing. From the signals exchanged, conformance and interoperability are indistinguishable too. We can say that now there exists the methodology and technology to make state of the art interoperability test suites runnable from single systems. There is also availability of methodologies and techniques for test case generation, message assembly and transmission.

Independence from legacy networking technologies was achieved too. This methodological and technical advances not only allow to test devices over new physical mediums, but to ease testing platform requirements. Recently it has been hard to implement collision domains using newly purchased equipment. With the proposed tools, every sort of traffic that can be virtualized and transmitted to the management node can be included in an interoperability test campaign.

Tester control and management applying virtualization

Automation solutions based on virtualization provides new means for interoperability testing. Results are presented addressing Internet Protocols, and are presented in Chapter 3 too. Configuration management in interoperability testing is no longer a problem for automation. Either using physical or virtual implementations, the complete configuration management was automated. Test cases can be now designed with new levels of freedom. Interoperability testing can be automated together with regression testing or other automatic testing activities. We foresee that these interoperability management tools are capable of matching the requirements of automatic test suite derivation based on formal methods.

Two publications were done with the results of this line of work: [SV06, SBBV07]. Different parts were also presented during ETSI Plugtest events and Tahi events to the testing community.

4.2 Future work

Several lines of work were addressed during this work. Most of them reached to an end, while others still remain open. Moreover, other new ideas were born too. Here we present some ideas of future research aspects related to what was presented here. Some of them are just the required steps forward to continue the evolution of ideas a even further. Other were just conceived, but we had no opportunity to research on them.

TTCN-3 Open/Free compiler

This is ongoing work, but we firmly believe that we will only see the limits for the TTCN-3 language after we have a compiler to play with. During different parts of this work we would have benefited from digging our hands in the internals of the compiler or making proposals on language properties. The lack of an Open/Free compiler makes the proposals too abstract or unreal to be even thought about. We expect that after we have access to first generation of Open/Free compilers language evolution will occur faster and new proposals will be backed with prototype implementations. We foresee a bigger and wider application of TTCN-3 language in different test fields based on the universal availability of TTCN-3 tools and test suites based on the open/free development process.

Portable Library format

TTCN-3 ATS reusability is based on cut and paste operations over source code. The language specification does not standardize means for pre-compiled code distribution binary formats. These details are left aside of the implementations and considered tool vendor implementation decisions. We believe that it is necessary to provide means for pre-compiled ATS distribution, as it is the case with major programming languages. Library usage provides better reusability characteristics to the language and better tools for test experts to design ATS and develop ETS.

Automatic traffic inspection

Automation of the interoperability platform reached full observability from a single *master* node. Execution decision still relies on the test expert, who is observing the evolution of the test and exchanged messages among systems. It is possible now to extend the tool with additional control features that allow automatic traffic inspection. With the evaluation of the traffic it is possible to automate the decisions concerning evolution of the test case and even real-time verdict issuing. Clearly automatic inspection of the traffic can verify more details than those controlled by the test expert. Automatic traffic inspection features can turn centralized-manual test execution synchronization into fully-automated execution of the test cases. Only manual operations remaining would affect vendor provided IUT, for which we lack automation capabilities.

Formal derivation of interoperability test suites

Provided the degrees of automation makes real the possibility of executing automated interoperability test suites generated formally. One of the common drawbacks of automatic derivation of test suites is that the number of test cases generated is several orders of magnitude higher than those manually designed. Up to now it is not feasible to execute manually thousands of test cases in reasonable amounts of time. Combined techniques of configuration management automation, virtualization and traffic inspection enables full automatic execution of interoperability test suites. We want to see full execution of interoperability test suites formally derived. A more precise definition of interoperability testing should be found too, specially one that addresses the semantic problems discussed in Section 3.6.

Glossary

ASN : Abstract Syntax Notation
ATS : Abstract Test Suite
BER : Basic Encoding Rules
BER : Bit Error Rate
BSD : Berkeley Software Distribution
CBC : Cipher Block Chaining
CCITT : Comité Consultatif International Téléphonique et Télégraphique
CD : (External) Coding/Decoding
CH : Component Handling
CRC : Cyclic Redundancy Check
CSMA/CD : Carrier Sense Medium Access / Collision Avoidance
CTS : Clear To Send
DAD : Duplicated Address Detection
DCF : Distributed Coordination Function
DES : Data Encryption Standard
ECMA : European Computer Manufacturers Association
ESP : Encapsulating Security Payload
ETS : Executable Test Suite
ETSI : European Telecommunications Standards Institute
GNU : GNU's not Unix
GPL : General Public License
GUI : Graphical User Interface
HA : Home Agent
HG : Home Gateway
HMAC : key-Hashed Message Authentication Code
ICMP : Internet Control Message Protocol
IDE : Integrated Development Environment
IDL : Interface Definition Language
IEC : International Electrotechnical Commission
IEEE : Institute of Electric and Electronic Engineers
IETF : Internet Engineering Task Force
IKE : Internet Key Exchange
INRIA : Institut National de Recherche en Informatique et en Automatique

IP : Internet Protocol
IPng : Next Generation Internet Protocol
IPsec : Internet Protocol Security
IPv4 : Internet Protocol version 4
IPv6 : Internet Protocol version 6
IRISA : Institute de Recherche en Informatique et Systèmes Aléatoires
ISO : International Organization for Standardization
ITU : International Telecommunication Union
IUT : Implementation Under Test
IXIT : Implementation eXtra Information for Testing
LCoA : Local Care-of Address
LGPL : Lesser General Public License
MAC : Media Access Control
MAC : Message Authentication Code
MD : Message Digest
MDA : Model Driven Architecture
MSC : Message Sequence Charts
MTC : Main Test Component
MTS : Methods for Testing and Specification
ND : Neighbor Discovery
NEMO : Network Mobility
NIC : Network Interface Card
NIST : National Institute of Standards and Technology
OMG : Object Management Group
OS : Operating System
OSI : Open Systems Interconnection
OSS : Open Source Software
PA : Platform Adaptor
PCO : Point of Control and Observation
PDU : Packet Data Unit
PDML : Packet Details Markup Language
PER : Packed Encoding Rules
PO : Point of Observation
PPP : Point to Point Protocol
PTC : Parallel Test Component
QoS : Quality of Service
RFC : Request For Comments
RTE : Routing Table Entry
SA : SUT Adaptor
SHA : Secure Hash Algorithm
SA : Security Association
SN : Sequence Number
SPD : Security Policy Database
SPI : Security Parameters Index

SUT : System Under Test
STF : Specialist Task Force
TA : Tentative Address
TCI : TTCN-3 Control Interface
TCP : Test Control Procedure
TCP : Transmission Control Protocol
TE : TTCN-3 Executable
TL : Test Logging
TM : Test Management
TMC : Test Management and Control
TR : Technical Report
TRI : TTCN-3 Runtime Interface
TSI : Test System Interface
TTCN : Tree and Tabular Combined Notation
TTCN-2 : Tree and Tabular Combined Notation, version 2
TTCN-3 : Testing and Test Control Notation, version 3
U2TP : UML 2.0 Testing Profiles
UML : Unified Modeling Language
VM : Virtual Machine
VPN : Virtual Private Network

Bibliography

[BCKZ02] C. Besse, A. R. Cavalli, M. Kim, and F. Zaïdi. Automated generation of interoperability tests. In *TestCom*, volume 210 of *IFIP Conference Proceedings*, pages 169–183. Kluwer, 2002.

[BIS06] Bison. http://www.gnu.org/software/bison/, 2006. [Online; accesed 22-April-2006].

[cci92] CCITT Recommendation X.292, OSI conformance testing methodology and framework for protocol Recommendations for CCITT applications - The Tree and Tabular Combined Notation (TTCN), 1992.

[CR05] A. Cavalli and E. Rodrigues. A Formal Approach of Interoperability Test Cases Generation Applied to Real Time domain. In *IEEE I2TS 2005 - International Information Technology Symposium, Florianopolis, SC, Brasil*, 2005.

[DH98] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification. http://www.rfc-editor.org/rfc/rfc2460.txt, 1998.

[DJA04] C. Perkins D. Johnson and J. Arkko. RFC 3775: Mobility Support in IPv6. ftp://ftp.rfc-editor.org/in-notes/rfc3775.txt, June 2004.

[DK03] Sarolta Dibuz and Péter Krémer. Framework and Model for Automated Interoperability Test and Its Application to ROHC. In *TestCom*, volume 2644 of *Lecture Notes in Computer Science*, pages 243–257. Springer-Verlag, 2003.

[DV05] A. Desmoulin and C. Viho. Quiescence Management Improves Interoperability Testing. In Ferhat Khendek and Rachida Dssouli, editors, *(TestCom 2005) Testing of Communicating Systems: 17th IFIP TC6/WG 6.1 International Conference, Montreal, Canada, May 31 - June, 2005, ISBN 3-540-26054-4*, pages 365–379. Springer, 2005.

[DV07] A. Desmoulin and C. Viho. A New Method for Interoperability Test Generation. In *(TestCom 2007) Testing of Software and Communicating Systems: 19th IFIP TC6/WG 6.1 International Conference, Tallinn, Estonia, 26-29 June, 2007, ISBN 978-3-540-73066-8*, pages 58–73. Springer, 2007.

[Eas05] D. Eastlake. RFC 4305: Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). http://www.rfc-editor.org/rfc/rfc4305.txt, 2005.

[End05] H. Endo. Getting IPv6 Ready Logo Phase-2 Certification PART 6 Tips for Successful Interoperability Testing. http://www.ipv6style.jp /en/apps/20050822/ 20050822_p.shtml, August 2005.

[ETH06] Ethereal: A Network Protocol Analyzer. http://www.ethereal.com/, 2006. [Online; accesed 19-April-2006].

[ETS03] ETSI. ES 201 873-1 TTCN-3 Core Language, Version: 2.2.1. http://www.ttcn-3.org/Specificationsed2.htm, 2003.

[ETS05a] ETSI. ES 201 873-1 Part 1: TTCN-3 Core Language, Version: 3.1.1. http://webapp.etsi.org/exchangefolder/esi_20187301v030101p.pdf, 2005. [Online; accesed 19-April-2006].

[ETS05b] ETSI. ES 201 873-5 Part 5: TTCN-3 Runtime Interface (TRI), Version: 3.1.1. http://webapp.etsi.org/exchangefolder/esi_20187305v030101p.pdf, 2005. [Online; accesed 19-April-2006].

[ETS05c] ETSI. ES 201 873-6 Part 6: TTCN-3 Control Interface (TCI), Version: 3.1.1. http://webapp.etsi.org/exchangefolder/esi_20187306v030101p.pdf, 2005. [Online; accesed 19-April-2006].

[ETS07] ETSI. TC MTS-IPT: IPv6 Testing an eEurope Project. http://www.ipt.etsi.org/deliverable.htm, 2007. [Online; accesed 22-April-2007].

[FJJV97] J.C. Fernandez, C. Jard, T. Jerón, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. In *Science of Computer Programming - Special Issue on Industrial RElevant applications of Formal Analysis Techniques*, pages 123–146, 1997.

[FLE06] Flex. http://www.gnu.org/software/flex/, 2006. [Online; accesed 22-April-2006].

[For] IPv6 Forum. IPv6 Ready Logo. http://www.ipv6ready.org/.

[For05] IPv6 Forum. Phase II Test Interoperability Specification Core Protocols. IPv6Ready_PhaseII_Base_Interop_version_2_8_4.pdf, December 2005.

[GD03] Roland Gecse and Sarolta Dibuz. An Intuitive TTCN-3 Data Presentation Format. In Dieter Hogrefe and Anthony Wiles, editors, *(TestCom 2003) Testing of Communicating Systems In 15th IFIP Testing of Communicating Systems, Tools and Techniques, ISBN 3-540-40123-7*, pages 63–78. Springer, 2003.

[GH99] Jens Grabowski and Dieter Hogrefe. Towards the third edition of TTCN. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *(Test-Com 1999) Testing of Communicating Systems, Methods and Applications, ISBN 0-7923-8581-0*, pages 19–30. Kluwer Academic Publishers, 1999.

[Gle04] Glenford J. Myers. *The ART of SOFTWARE TESTING*. John Wiley & Sons, Inc., 2004.

[Gra94] Jens Grabowski. SDL and MSC Based Test Case Generation - An Overall View of the SAMSTAG Method. Technical report, Technical Report IAM-94-005, University of Berne, Institute for Informatics, Berne, Switzerland, May 1994, May 1994.

[Gre05] B. Greear. 802.1Q VLAN implementation for Linux. http://www.candelatech.com/ greear/vlan.html, September 2005.

[GWWH00] 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.

[HD98] R. Hinden and S. Deering. RFC 2373: IP Version 6 Addressing Architecture. http://www.rfc-editor.org/rfc/rfc2373.txt, 1998.

[HLSG04] Ruibing Hao, David Lee, Rakesh K. Sinha, and Nancy Griffeth. Integrated system interoperability testing with applications to voip. *IEEE/ACM Trans. Netw.*, 12(5):823–836, 2004.

[iee90] IEEE standard glossary of software engineering terminology E-ISBN: 0-7381-0391-8, 1990.

[IEE98] IEEE. IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks, IEEE Std. 802.1q, 1998.

[Inc07] VMware Inc. VMware: Virtualization, Virtual Machine and Virtual Server Consolidation. http://www.vmware.com/, April 2007.

[IPv07] IPv6 Ready Logo Phase 2. NEMO: Interoperability test scenario. http://www.ipv6ready.org/about_phase2_test.html, January 2007.

[ISO94] Information Technology - Open Systems Interconnection - Conformance testing methodology and framework. International Multipart Standard, ISO/IEC Std. 9646, 1994.

[itu98] IUT-T Recommendation X.292, OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications - The Tree and Tabular Combined Notation (TTCN), 1998.

[itu01a] ITU-T Recommendation Z.140, The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language,, 2001.

[itu01b] ITU-T Recommendation Z.141, The Tree and Tabular Combined Notation version 3 (TTCN-3): Tabular Presentation Format, 2001.

[JAD04] V. Devarapalli J. Arkko and F. Dupont. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. ftp://ftp.rfc-editor.org/in-notes/rfc3776.txt, June 2004.

[Kau05] C. Kaufman. RFC 4306: Internet Key Exchange. http://www.rfc-editor.org/rfc/rfc4306.txt, 2005.

[Ken05] S. Kent. RFC 4303: IP Encapsulating Security Payload (ESP). http://www.rfc-editor.org/rfc/rfc4303.txt, 2005.

[KOS$^+$00] T. Kato, T. Ogishi, H. Shinbo, Y. Miyake, A. Idoue, and K. Suzuki. Interoperability Testing System of TCP/IP Based Systems in Operational Environment. In *TestCom*, pages 143–156, 2000.

[LIB06a] tcpdump/libpcap. http://www.tcpdump.org/, 2006. [Online; accesed 27-April-2006].

[LIB06b] The Libnet Packet Construction library. http://www.packetfactory.net/libnet, 2006. [Online; accesed 27-April-2006].

[MM97] G. Malkin and R. Minnear. RFC 2080: RIPng for IPv6. http://www.rfc-editor.org/rfc/rfc2080.txt, 1997.

[NNS98] T. Narten, E. Nordmark, and W. Simpson. RFC 2461: Neighbor Discovery for IP Version 6 (IPv6). http://www.rfc-editor.org/rfc/rfc2461.txt, 1998.

[NSL$^+$06] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology. ftp://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf, August 2006.

[PDM06] PDML Specification. http://analyzer.polito.it/docs/dissectors/PDMLSpec.htm, 2006. [Online; accesed 19-April-2006].

[PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[Pos81a] J. Postel. RFC 791: Internet Protocol. http://www.rfc-editor.org/rfc/rfc791.txt, 1981.

[Pos81b] J. Postel. RFC 793: Transmission Control Protocol. http://www.rfc-editor.org/rfc/rfc793.txt, 1981.

[RC91] O. Rafiq and R. Castanet. From Conformance Testing to Interoperability Testing. In *Protocol TEst Systems, volume III*, pages 371–385. IFIP, Elsevier Sciences Publishers, 1991.

[SBBV07] A. Sabiguero, A. Baire, A. Boutet, and C. Viho. Virtualized Interoperability Testing: Application to IPv6 Network Mobility. In *Proceedings of the Eighteenth IFIP International Workshop on Distributed Systems: Operations and Management*, 2007.

[SBD$^+$06] A. Sabiguero, A. Baire, A. Desmoulin, F. Roudaut, A. Floch, and C. Viho. Towards and IP-oriented testing framework - The IPv6 Testing Toolkit. TTCN-3 User Conference 2006 - 30 may-8 June, Berlin, Germany (http://www.ttcn-3.org/), 2006.

[SBFV05] A. Sabiguero, A. Baire, A. Floch, and C. Viho. Using TTCN-3 in the Internet Community: an experiment with the RIPng protocol. TTCN-3 User Conference 2005 - 6-8 June, Sophia-Antipolis, France (http://www.ttcn-3.org/), 2005.

[SBV06] A. Sabiguero, A. Baire, and C. Viho. Embeding traffic capturing and analysis extensions into TTCN-3 System Adaptor. In Winfried Dulz and Wolfgang Schröder-Preikschat, editor, *MMB Workshop Proceedings: Model Based Testing and Non-Functional Properties of Embedded Systems, ISBN 978-3-8007-2956-2*, pages 27–35. VDE Verlag, 2006.

[SCV07] A. Sabiguero, M. Corti, and C. Viho. The new Internet Protocol security IPsec testing with TTCN-3. TTCN-3 User Conference 2007 - 29 may-1 June, Stockholm, Sweden (http://www.ttcn-3.org/), 2007.

[SFRV05] A. Sabiguero, A. Floch, F. Roudaut, and C. Viho. Some Lessons from an experiment using TTCN-3 for the RIPng testing. In Ferhat Khendek and Rachida Dssouli, editors, *(TestCom 2005) Testing of Communicating Systems: 17th IFIP TC6/WG 6.1 International Conference, Montreal, Canada, May 31 - June, 2005, ISBN 3-540-26054-4*, pages 318–332. Springer, 2005.

[SK05] K. Seo S. Kent. RFC 4301: Security Architecture for the Internet Protocol. http://www.rfc-editor.org/rfc/rfc4301.txt, 2005.

[SKCK04] Soonuk Seol, Myungchul Kim, Samuel Chanson, and Sungwon Kang. Interoperability Test Generation and Minimization for Communication Protocols Based on the Multiple Stimuli Principle. In *IEEE Journal on Selected Areas in Communications, Vol 22, No 10*, pages 2062–2074. IEEE, 2004.

[SV06] A. Sabiguero and C. Viho. Plug once, test everything. Configuration Management in IPv6 Interop Testing. In *Proceedings of the Fifteenth Asian Test Symposium*, pages 443–448. IEEE Computer Society Conference Publishing Services, 2006.

[SVG02] Stephan Schulz and Theofanis Vassiliou-Gioles. Implementation of TTCN-3 test systems using the TRI. In Ina Schieferdecker, Hartmut Köning, and Adam Wolisz, editors, *(TestCom 2002) Testing of Communicating Systems, Application to Internet Technologies and Services, ISBN 0-7923-7695-1*, pages 425–442. Kluwer Academic Publishers, 2002.

[SVG03] Ina Schieferdecker and Theofanis Vassiliou-Gioles. Realizing distributed TTCN-3 test systems with TCI. In Dieter Hogrefe and Anthony Wiles, editors, *(TestCom 2003) Testing of Communicating Systems In 15th IFIP Testing of Communicating Systems, Tools and Techniques, ISBN 3-540-40123-7*, pages 95–109. Springer, 2003.

[t3d07] T3DevKit. http://t3devkit.gforge.inria.fr/, 2007. [Online; accesed 22-April-2007].

[tah98] TAHI Project. whoami.html, 1998.

[TAH06] TAHI Project. TAHI IPv6 Interoperability Test Tools and scripbs, March 2006.

[TES07] IPv6 Ready Logo. Phase II Test Specification IPsec. http://www.ipv6ready.org/pdf/IPsec_1_8_0b3.pdf (last ckecked 22/04/2006), 2007.

[TN98] S. Thomson and T. Narten. RFC 2462: IPv6 Stateless Address Autoconfiguration. http://www.rfc-editor.org/rfc/rfc2462.txt, 1998.

[Tör99] M. Törö. Decision on tester configuration for multiparty testing. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *(TestCom 1999) Testing of Communicating Systems, Methods and Applications, ISBN 0-7923-8581-0*, pages 109–128. Kluwer Academic Publishers, 1999.

[Tre99] Ian Tretmans. Testing concurrent systems: A formal approach. In J.C.M. Baeten and S.Mauw, editors, *CONCUR'99 - Concurrency Theory: 10th International Conference, Eindhoven, The Netherlands, August 1999. Proceedings, ISSN 0302-9743 (Print) 1611-3349 (Online)*, page 779. Springer, 1999.

[ttc92] ISO/IEC 9646-3, Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN), 1992.

[ttc98] ISO/IEC 9646-3, Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN), 1998.

[ttc00] ETSI, "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; TTCN-3: Core Language", 2000.

[UKW99] Andreas Ulrich, Hartmut Kööning, and Thomas Walter. Architectures for testing distributed systems. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *(TestCom 1999) Testing of Communicating Systems, Methods and Applications, ISBN 0-7923-8581-0*, pages 93–108. Kluwer Academic Publishers, 1999.

[Ulr06] Ulrich Grude and Friedrich Wilhelm Schröer and Peter Enskonatus. TTCN-3 for .Net. In Ina Schieferdecker and Anthony Wiles, editors, *TTCN-3 User Conference 2006 - http://www.ttcn-3.org/TTCN3UC2006/FinalPresentations/P5.2_rennoch-presentation.pdf*, 2006.

[V. 05] V. Devarapalli, R. Wakikawa, A. Petrescu and P. Thubert. RFC 3963: Network Mobility (NEMO) Basic Support Protocol. ftp://ftp.rfc-editor.org/in-notes/rfc3963.txt, January 2005.

[V. 07] V. Devarapalli and F. Dupont. Mobile IPv6 Operation with IKEv2 and the revised IPsec Architecture. ftp://ftp.rfc-editor.org/in-notes/rfc4877.txt, April 2007.

[vdBRT04] M. van del Bijl, A. Rensink, and J. Tretmans. Component based testing with ioco. In A. Petrenko and A. Ulrich, editors, *FATES 2003 - Formal Approaches to Testing of Software, volume 2931 of Lecture Notes in Computer Science*, pages 86–100. Springer, 2004.

[VGSB+99] Theofanis Vassiliou-Gioles, Ina Schieferdecker, Marc Born, Mario Winkler, and Mang Li. Configuration and execution support for distributed tests. In Gyula Csopaki, Sarlota Dibuz, and Katalin Tarnay, editors, *12th IFIP International Workshop ont Testing of Communicating Systems, Testing of Communicating Systems - Methods and Applications, ISBN 0-7923-8581-0*, pages 61–76. Kluwer Acdemic Publishers, 1999.

[WDT+05] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3.* Wiley, 2005.

[WLY03] Jianping Wu, Whongjie Li, and Xia Yin. Towards Modeling and Testing of IP Routing Protocols. In Dieter Hogrefe and Anthony Wiles, editors, *Testing of Communicating Systems In 15th IFIP Testing of Communicating Systems, ISBN 3-540-40123-7*, pages 49–62. Springer, 2003.

[Zei06] A. Zeichick. Processor-Based Virtualization, AMD64 Style, Part I. http://developer.amd.com/articles.jsp?id=14&num=1, 2006.

List of Figures

Résumé

Dans le contexte des protocoles de communication, il existe deux approche fondamentales pour tester les implémentations et s'assurer qu'elles fonctionneront correctement ensemble: le test de conformité et d'interopérabilité. Le but des deux approches est de fournir des suites de test exécutables (ETS) qui sont exécutées contre les implémentations. De grands efforts ont été fournis pour définir des langages de spécification de suites de tests abstraits (ATS) et des environnements pour dériver les ETS. Les langages de spécification d'ATS doivent être le plus abstrait possible. D'un autre côté, les environnements utilisés pour exécuter ces ETS sont conçus pour être proche de ceux utilisés par les implémentations. Il en résulte un gap entre les ATS et les ETS. La dérivation d'ETS à partir d'ETS est complexe, difficile et souvent sujette à erreurs, obligeant les développeurs de test à écrire les ATS qui soient très proches des ETS. Ceci entraîne la perte de l'abstraction nécessaire pour les décrire les ATS, leur portabilité et les propriétés intéressantes de maintenance et de réutilisation. Pour être exécutés sur d'autres implémentations, les tests ainsi développés doivent être pour la plupart complètement réécrits.

Cette thèse tente de réduire le gap entre ATS et ETS aussi bien du côté des ATS que des ETS. Elle propose des solutions qui préservent l'abstraction des ATS, qui facilitent la dérivation des ETS en garantissant leur portabilité. Le langage TTCN-3 est utilisé comme langage de spécification des ATS et plusieurs protocoles Internet nouvelle génération sont utilisées pour expérimenter les solutions et méthodes proposées. La virtualisation est également proposée comme solution pour gérer les problèmes de configuration, d'exécution et de contrôle lors des exécutions des tests, rendant ainsi les systèmes de tests d'interopérabilité indépendants des aspects matériels.

Abstract

In the context of communication protocols, there are basically two approaches to testing implementations to ensure that they will work effectively together: conformance and interoperability testing. The goal of both approaches is to provide ETS (Executable Test Suites) which are executed against implementations. Big effort has been invested defining languages for specifying ATS (Abstract Test Suites) and environments for deriving ETS. Languages for specifying ATS need to be as abstract as possible. On the contrary, environments used to execute ETS are designed to be close to the implementations. There is a gap between ATS and ETS. The work of deriving ETS from ATS is complex, intricate and often error-prone, making testers trying to write directly ATS close to the ETS. Abstraction, portability and other life-cycle relevant properties are lost. To be executed on another implementations or in another environment, the so obtained tests have to be completely rewritten most of the times. The complexity of the ATS to ETS transformation is not understood at first glance. No matter how abstract you are on your ATS, all details must be present on your ETS. The problems addressed go further than just a one-time ETS development, but ATS-ETS lifecycle management.

This thesis tackles the problem of bridging the gap by working on both the abstract and executable sides. TTCN-3 language is taken as an abstract test specification language, and several Internet Protocols are used to apply different methodological solutions. Virtualization is used to automate several configuration and test management problems. Virtualized interoperability testing also allows test systems become independent of hardware components.