Modernization of Legacy Software Tests to Model-Driven Testing

Nader Kesserwan(✉), Rachida Dssouli, and Jamal Bentahar

Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Canada
n_kesse@encs.concordia.ca,
rachida.dssouli@concordia.ca,
bentahar@ciise.concordia.ca

Abstract. Software has become ubiquitous in healthcare applications, as is evident from its prevalent use for controlling medical devices, maintaining electronic patient health data, and enabling healthcare information technology (HIT) systems. As the software functionality becomes more intricate, concerns arise regarding quality, safety and testing effort. It thus becomes imperative to adopt an approach or methodology based on best engineering practices to ensure that the testing effort is affordable. Automation in testing software can increase product quality, leading to lower field service, product support and liability cost. It can provide types of testing that simply cannot be performed by people.

Keywords: Automation · Model-driven testing · Human factors · Reverse-Engineering · Testing effort

1 Introduction

Proprietary frameworks for software tests are common in industry. The resulting test cases and procedures can be very large, difficult to maintain and hard to compose into complex scenarios involving parallelism. Migration to a standards-based and more efficient software testing environment is appealing to organizations seeking to reduce costs, and to benefit from the continuing advancements in technology. In this paper, we propose to modernize legacy software tests to a model-driven testing methodology, based on formalized test cases. The legacy test procedures are initially translated to Testing and Test Control Notation (TTCN-3) and then abstracted to test cases in the Test Description Language (TDL). The goal here is to study model-driven test procedure generation from TDL, and to evaluate TDL as a formal language for expressing test cases. Software modification is a typical activity in a software system's life cycle and especially in the maintenance phase [19]. During maintenance, the challenges of integration and regression testing are to select a sufficient subset of existing tests to apply and to create new tests. Test automation is highly desirable for regression testing, an activity that is generally tedious and time consuming [8].

After many maintenance cycles, a legacy set of Test Cases and Procedures (TCs and TPs) may become increasingly difficult to adapt to change of the System Under Test (SUT) or difficult to improve using new functionalities related to test automation.

This limitation is sometimes exacerbated by the high-cost and the inherent complications of integration with an evolving SUT environment. At some point or another, an organization may wish to modernize its software testing framework. Such a migration is costly, but could bring many benefits: a modern infrastructure that allows better test management and a higher level of test automation [11]. Furthermore, the migrated set of TCs and TPs should be easier to change, enhancing the agility of the organization. Organizations need to be able to integrate additional functionalities seamlessly when new requirements arise. Then, reengineering of legacy TPs could address some of the issues. In this paper, we present a pilot project in reengineering of embedded software TCs and TPs towards a model-driven methodology, whereby executable TPs are automatically generated from TCs. The modernization process has two phases:

i. The first phase is tool-based; it starts with the automatic translation of legacy TPs to functionally-equivalent TPs (test implementation) in the TTCN-3 [9]. The TTCN-3 language was selected for its industrial strength, its ability to enable test automation, and its recognition as a standard.
ii. In a second phase, the TTCN-3 TPs are abstracted into TCs (test specification) written as models in TDL [10]. TDL is a standardized scenario-based approach; it expresses requirements as test objectives and connects them to scenarios that describe the interaction with the SUT. This approach is suitable for automated TPs as tests can be derived from the scenarios and automated.

We use the terminology[1] of the ETSI TDL and TTCN-3 standards, as defined in context, to describe various test artifacts and activities. In this paper, we discuss the modernization of software tests for testing software. The rest of this paper is organized as follows: Sect. 2 surveys related work; Sect. 3 discusses the modernization approach; and Sect. 4 concludes the paper.

2 Related Work

In general, system or software migration is performed in order to increase quality, to manage obsolescence, and/or to satisfy new business requirements. The migration of legacy software tests may involve the hardware platform, the software framework or both. A hardware migration implies switching the hardware platform to a modern one. On the other hand, software migration proposes to change the programing language, the operating system, and the data or the database. For example, data can be migrated from a file system to a database management system. In [4], a migration from a relational to an object-oriented database was implemented to benefit from object-oriented technology. TPs written in programming languages such as Perl, Python, C, C++, etc. are considered to be software programs. Therefore, the migration to a new language is handled as a specific language translation activity. In [15], massive amounts of TPs that were implemented as Excel and Word files were migrated

[1] **TDL:** Test Description, Test Objective, Test Configuration, Data Set
TTCN-3: testcase, Test Type, Test Data, Test Component, Test Behavior.

to a centralized test management tool. The authors aimed to achieve better test management of test artifacts— distributed in an ad-hoc manner— by centralizing them in one location. In [12], a medium size software system written in PL/IX was migrated to C++ to respond to new business requirements, such as lower maintenance costs, higher performance and better reliability. In [16], the authors address the potential and risks of migrating the software and the hardware platforms of massive software tests that may require significant computing resources and lengthy execution time to cloud computing. Several tools and strategies to assist in the migration of legacy systems are presented in De Lucia et al. [1, 2]. Reliable analysis of source code is essential for successful software migration. Selecting and comparing analysis tools is usually based on a defined set of criteria. Analysis tools can be used to support a migration, for example, by identifying lines in the test code, analyzing the control flow, and providing information about test data [20]. Our work differs by migrating the software tests to be used in model-driven testing methodology.

3 Reverse-Engineering of Legacy Software Tests

Most companies that provide solutions to organizations in healthcare use Natural Language (NL) to express software requirements. In the Requirements Engineering Management Findings Report [13] several surveys of industry practice are conducted: *"…when asked in what format requirements are being captured, the overwhelming majority of the survey respondents indicated that requirements are being captured as English text, shall statements, or as tables and diagrams"*.

In our pilot project, the original TPs were written in a proprietary test language based on Eclipse Ant/XML [3] software is a PC based verification tool used to execute automated test in order to verify the software. Based on user feedback, manually creating TPs in Ant/XML can be qualified as labor-intensive. The legacy TPs are the starting point of the migration as they convey information about test behavior, test components and test data. In this paper, we propose an approach that starts with the code migration of these TPs to the TTCN-3 language, which in turns will be reverse-engineered into TCs in TDL. Once the reengineering of the software tests is completed, new TCs can be captured directly in TDL, and these TCs can be used to generate TPs in TTCN-3 or in any other desired scripting language. Furthermore, when new requirements emerge to demand the evolution of the software tests, this software evolution can take place at the model level. Figure 1 shows the modernization process of the legacy software tests.

There are some difficulties with the legacy process shown in Fig. 1. The test engineer spends a lot of time transforming TC into TP. There is a large gap in abstraction level between the TC and the TP. The detail level is low in the TC, but very high in the TP. It is an error-prone, time-consuming task to bridge this gap manually. Because resources are always restricted, the software quality engineer has less time for a more intensive TC. For example, Patient Monitors (PM) are electronic medical devices for observing critically ill patients by monitoring their vital signs. The four

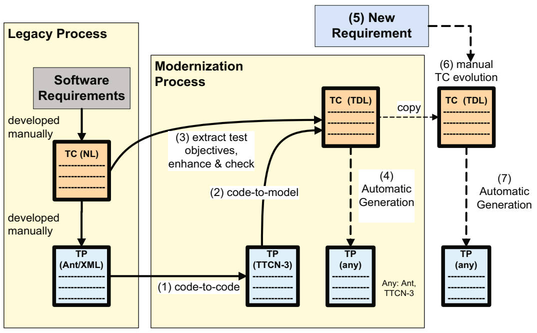
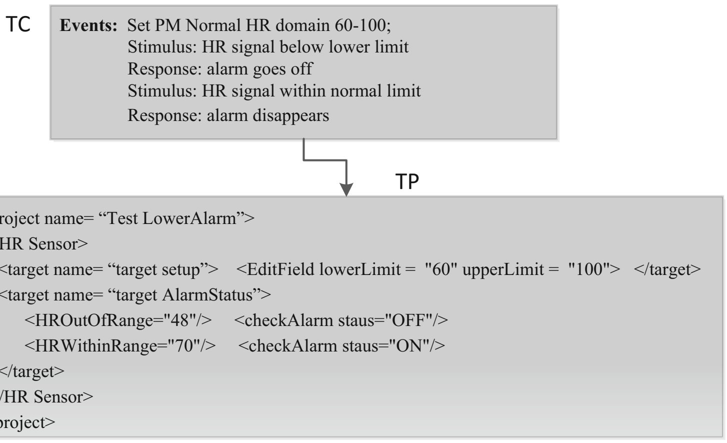


Fig. 1. Modernization of legacy software tests

most important vital signs are the pulse rate, oxygen level, body temperature, and electrocardiogram (ECG) activity. Doctors and nurses are informed by an immediate alarm when a vital sign of a patient such as heart rate falls below a given lower limit or exceeds a given higher limit, so they can provide timely and appropriate treatment. A malfunction of the PM may result in the death of a patient which makes this functionality safety-critical and it must be validated accordingly. Figure 2 shows an example that illustrates a legacy TC for Heart Rate (HR) alarm testing and the transformation to the appropriate TP.



Events: Set PM Normal HR domain 60-100;
Stimulus: HR signal below lower limit
Response: alarm goes off
Stimulus: HR signal within normal limit
Response: alarm disappears

```
<project name= “Test LowerAlarm”>
 <HR Sensor>
  <target name= “target setup”>   <EditField lowerLimit =  "60" upperLimit =  "100">   </target>
  <target name= “target AlarmStatus”>
      <HROutOfRange="48"/>    <checkAlarm staus="OFF"/>
      <HRWithinRange="70"/>    <checkAlarm staus="ON"/>
  </target>
 </HR Sensor>
</project>
```

Fig. 2. Develop a TP manually from a TC

The ultimate goal in the modernization process is to enable automatic generation of the TPs and to have them migrate to a more standard testing language to benefit from its important features. The next subsections explain the two activities enclosed in the modernization process, code-to-code migration and code-to-model.

3.1 Code-to-Code Migration

From a migration perspective, we consider a set of existing software tests as software source code. Typical migration approaches [5, 7, 14] fall into three categories:

- New development of the application. For large-scale systems, rewriting an application from scratch is difficult, error prone, expensive and time consuming [6].
- Modernizing the legacy code itself. Code modernization is an attractive approach to organizations as it has the highest probability of success among the three types of approaches [17]. The migration risks are identified early in the code modernization process.
- Automated conversion of the legacy application. Automated conversion is an attractive option when new technology or process improvement initiatives require legacy software tests to be converted to another language, for example to improve the test process itself or its environment. The current technology for legacy migration allows organizations to automate the migration process with lower risk and in a shorter time.

The need for better test management environments grows with the size of the testing effort. Organizing test artifacts and resources is a necessary part of test management and is facilitated in our reengineering process by dispatching elements of the legacy TPs into its new TTCN-3 implementation— in modular style. Our code-to-code migration activity is iterative as it requires a repetition of activities to migrate successfully the code. The process is composed of the following sub-activities:

- Analysis of the legacy TPs to gain understanding of the code to be translated to TTCN-3;
- Code translation according to transformation rules, from legacy to TTCN-3, defined early in the analysis activity; and
- Deployment and Verification of the migrated TPs to ensure their satisfaction of the legacy TCs, and equivalence to the legacy TPs.

These sub-activities are shown in Fig. 3 and will be explained as we progress after an overview of the TTCN-3 language below.

TTCN-3 is a standardized language used to write human and machine-readable test scripts; it is designed specifically for conformance testing. The semantics of TTCN-3 is clearly and precisely specified. A TTCN-3 test suite can be structured into several modules. A module is a top-level container for code which is composed of an optional control part and an arbitrary number of definitions. A TTCN-3 "testcase" is a behavior description of how to stimulate the SUT. Each TTCN-3 testcase runs on components that communicate with each other through ports. A TTCN-3 model is composed of an Abstract Layer and a Concrete Layer. The Abstract Layer can host a test suite that is composed of several modules to describe the test to be executed. The Concrete Layer is responsible for communicating with the SUT, coding and decoding the exchanged messages during the test execution.

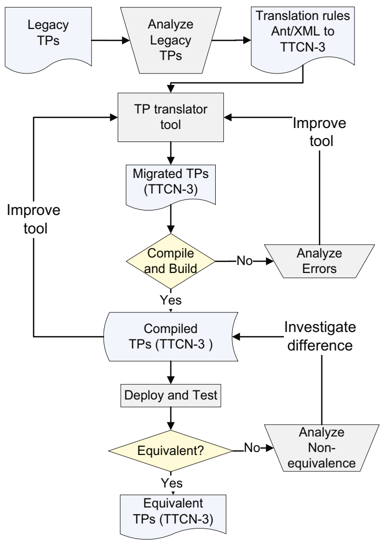


Fig. 3. Code-to-code migration sub-activities

3.1.1 Analysis of the Legacy Code

It is essential to the success of a migration to understand the functionality of the legacy software tests and to know how the SUT interacts via its various interfaces. A typical TP written in Ant/XML is composed of one or more targets where each target may control several interfaces via sending and verifying commands. For example, if a TP needs to sense some data, it specifies the interface to be queried, sends the required commands and reads the result for verification.

After analyzing the legacy TPs, we extracted valuable information that is located exclusively in the TPs; this information helped us to establish principles for code migration from Ant/XML to TTCN-3. We determined that a legacy TP contains instructions pertaining to three aspects: Test Verification, Test Component, and Test Action. This decomposition of a legacy TP can support the translation to TTCN-3 in a modular manner. In our translation scheme, the Test Verification, Test Component and Test Action are respectively translated to Test Data, Test Component and Test Behavior modules in TTCN-3. The transformation rules should convert legacy TP code while preserving the semantics. We found that core TTCN-3 contains equivalent semantics for most Ant/XML language elements, such as sending data, verifying responses, representing regular expression, defining interfaces and connections, etc.; however, since the legacy application did not use an XML schema, the data types of the legacy TP could not easily be rule-transformed. Thus, neither type checking nor value checking could be performed.

3.1.2 Code Migration

We developed a language translator tool to migrate the TPs automatically to three TTCN-3 modules. The resulting modules along with the Type module constitute an executable TTCN-3 TP that is equivalent to the Ant/XML TP. The Type module is

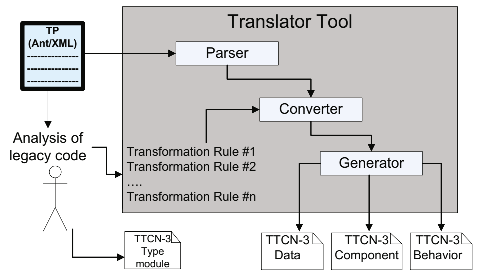


Fig. 4. Language translator tool

produced manually by analyzing the SUT inputs and outputs and the legacy TCs and TPs as shown in Fig. 4. The architecture of the translator tool combines the following elements:

- Transformation Rules: a number of rules to transform each Ant/XML construct to a one or more equivalent constructs in TTCN-3. (one-to-many transformations are possible)
- Parser: reads legacy TP in order to generate syntactic element tokens encountered in the TP.
- Converter: based on transformation rules, it transforms the syntactic element, returned by the Parser, to functionally-equivalent code to the generator.
- Generator: writes the generated TTCN-3 code, produced by the Converter, dispatched in each of the corresponding modules.

Table 1 shows the transformation rules. Third column describes how the TP legacy semantic is preserved using TTCN-3 syntax.

Having a Translator Tool is a key success factor in the migration process. It enables a high degree of automation and meets economic and timeframe objectives, such as lower cost, shorter time-to-market, consistent style, and good quality code. However, the migrated TTCN-3 modules define an abstract test suite, i.e. components residing in the TTCN-3 Abstract Layer. The migrated code lacks any concrete implementation-specific information, such as how messages are encoded or how communication with the SUT actually takes place [21]. In order to execute the TTCN-3 modules, Codecs and SUT Adapters must be provided. These parts reside in the Concrete Layer and allow tests to be encoded into a format understood by the SUT and to be executed by it.

3.1.3 Migrate Real-Time TPs

Real-time embedded applications require testing the functional aspect and the timing aspect of the requirements. The functional aspect of SUT interfaces is concerned with the sending and receiving of the messages; it is verified by checking the message values and their order. On the other hand, the real-time requirements require observing time at the communication ports and associating time with stimuli and responses. Testing real-time behavior remains a challenge as the test system needs to be time-deterministic [18], in order to verify accurately the response time, sending time, latency, delay, jitter, etc.

Table 1. Transformation rules to convert Ant/XML and TTCN-3 languages along with transformation rules.

Legacy code element	**Equivalent construct in TTCN-3**	**Transformation rules**

<project name = "str">	module <str_Template> { }
	module <str_Behavior> { }
	module <str_Configuration> { }
		Rule # 1: **project** element is translated to three **module** constructs which together compose a full TP in TTCN-3. The project name = str is used as a prefix with "Template", "Behavior" or "Configuration" to designate each TTCN-3 module. If the project name contains special characters such as dot or space, they are replaced by underscores.

<target name = "str">	testcase <target_str> runs on MTCType system SystemType	Rule # 2: **target** element is translated to a **testcase** construct, and the target name is prefixed with the string target_ The testcase will contain the action and verify constructs (stimulus and response)

<target name= "all" depends = "str_1, str_2, …, str_n"/>	control {
	 execute (target_str_1());
	 execute (target_str_2());
	 execute (target_str_n());
	}
		Rule # 3: **target** name = all is translated to a **control** construct, and the intermediate targets, str_1, str_2, … separated by commas, identified in depends are translated to a sequence of **execute** statements such as execute (target_str_1()); in the **control** construct.

interface port = "name"	type port interface_name message {
	 in sending_msg;
	 out receiving_msg; }
	type component interfaceType {
	 port interface_*x* interface; }
		Rule # 4: Every interface is mapped to a message-based port and attached to a component.
		
		The **interface** port = name is translated to a type port message-based construct and attached to a type component construct.

	function action (name, command, str_1, …, str_n) runs	Rule # 5: **action** elements are translated to **functions** and **function** calls constructs. The action parameters

<action key = "str_1", "str_2", …, str_n />	on componentType {
	 ….
	 portName.send(command, str_1, …, str_n);
	 ….
	 }
		command, name, str_1, …, str_n are passed as formal parameters to the function definition. The parameter name represents the interface name where command represents the input to send. Some actions take additional parameters to send the command, they can be represented by str_1, …, str_n. The parameter portName represents the port via which the input to SUT is sent. The action with its arguments in the legacy TP represent a stimulus to send to the SUT

< verify query = "str_1" value= "str_2" />	template component type verifyStep := { str_1 := **pattern** str_2 }
	 function matchResult(verify, portName) runs on componentType {
	 alt {
	 [] portName.receive(verify) {
	 setverdict(pass); }
	 [] portName.receive {
	 setverdict(fail); }
	 [] replyTimer.timeout {
	 setverdict(inconc, "No response from
	 SUT") } }
		Rule # 6: **verification** is translated to **template** construct named verify. One **template** can host several **verifications** for a given step. Then, the construct verify is translated to a function to handle the alternative sequences. In the legacy TP, a comparison between the expected value and returned one is performed: verify query = "str_1" value= "str_2"
		 The TTCN-3 TP migrates the expected values and store them in templates w.r.t to REGEXP used in the legacy. Then, the returned values are matched against the expected ones to issue a verdict.

< macrodef name = "MacroN" />
 action
 <MacroN interface = "interface_name, $para_1$, $para_2$, …, $para_n$" />
	function MacroN (interface_name, $para_1$, $para_2$, …, $para_n$) runs on componentType {
	 …}
	 MacroN(interface_name, $para_1$, $para_2$, …, $para_n$);
		Rule # 7: **macros** elements are translated to **functions** and **function** calls constructs. The macros parameters interface_name, $para_1$, $para_2$, …, $para_n$ are passed as formal parameters to the function definition. A macro may contain control statement such as looping, if, else. These statements are mapped to their equivalent in TTCN-3

Previously, we transformed Ant/XML TPs, which test the functional behavior of the SUT, to TTCN-3. Now, we discuss how Ant/XML and TTCN-3 handle the real-time. The Ant/XML test system, without any additional mediating devices, is unable to execute precise real-time tests scenarios. The wall clock time is measured at scripting level, i.e. not at the external test adapters. Ant/XML measures time via *tstamp* operation returning time with a millisecond resolution; it controls timing via *sleep* and *waitfor* operations.

Similarly, the core language of TTCN-3 was not originally conceived with real-time focus in mind. The first problem is the precision of time when it is recorded or checked by the test system or when it is associated with certain events. There is the semantic of timers that was not intended for suiting real-time properties, but conceived only for catching (typically long-term) timeouts [19].

A *tstamp* task is mapped to a TTCN-3 timer declaration followed by *start* timer operation. Ant/XML measures duration by computing time difference between two *tstamp* tasks. This can be mapped to TTCN-3 *start* timer and *read* timer operations.

TTCN-3 was extended with set of constructs for real-time testing RT-TTCN-3 [10] that introduces a mechanism to store the arrival time of messages, procedure calls at system adapter level and to control the timing for the stimulation. We have not yet attempted to use RT-TTCN-3. For now, only the core language is used in our pilot project.

3.1.4 Deployment and Testing of a Migrated TPs

As illustrated in Fig. 3, the code migration process is an iterative approach, as the Translator Tool needs improvement and corrections after an unsuccessful attempt to migrate the functionality. The migrated code may not compile or may fail to link to produce a build; in such cases an analysis is performed to improve the tool and fix the problem. It is essential that the legacy and migrated test suites are functionally equivalent and have full consistency in their verdicts that are produced by applying the same stimuli. In order to properly evaluate their conformance, the migrated code should not be manually modified during the process by diverging from the TC. Any enhancement should be added only after a successful migration has been declared. Accordingly, as shown in Fig. 5, testing the functional equivalence can be determined by comparing the original and migrated test verdicts' and the SUT's observable states — actions triggered by SUT in response to events sent by the TP.

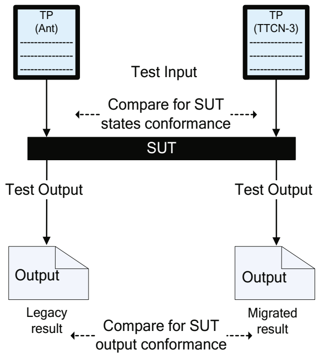


Fig. 5. Validation of legacy and migrated test suites equivalence

The correctness relationship holds when the migrated TPs, for the same test input, make SUT behave the same way as legacy TPs do. In other words, the legacy TPs are used as an oracle version, if the behavior of the SUT differs when stimulated by the migrated TPs, then the correctness relationship breaks and the tester needs to analyze the problem and investigate the difference. Annex 1 shows the TTCN-3 test suite migrated from the legacy TP in Fig. 2. After a successful code equivalence migration to TTCN-3, the second phase starts by reverse-engineering the migrated code in TTCN-3 to abstract models in TDL.

3.2 Code-to-Model

In the second phase of the reengineering process, we obtain the TCs by reverse-engineering the migrated TTCN-3 TPs. In most industrial domains, a test can be conceived at two levels of abstraction: a test specification (or test case) and a test implementation (a test script or procedure). Our goal is to abstract the latter to obtain the former. Here, the test implementation is the migrated TTCN-3 TPs containing concrete information. It is often considered useful to express TPs as stimulus-response scenarios. This is the path that we explore here using TDL.

Let's consider the modules of a TP.

- The *Test Behavior* module is composed of test events (stimuli and responses as interactions) that express the test behavior.
- The *Test Data* module contains information about the test input and the expected test output.
- The *Test Component* module consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface.

Next we consider a TC. It should use abstract types and instances to refer to test data, and should describe the system components and their actions and interactions with a minimum of details. In our project, to raise the level of test specification, we choose the TDL notation. It has the benefit of being complementary to TTCN-3. For a given test, a description is specified in TDL, whereas TTCN-3 is used to define a detailed implementation. An overview of the TDL concept follows.

TDL is a new language for the specification of test descriptions and the presentation of test execution results [10]. The introduction of TDL is being driven by industry to fill the gap between the high-level expression of what needs to be tested i.e., the test purposes, and the complex coding of the executable TP in TTCN-3 [18]. TDL is used primarily— but not exclusively— for functional testing, its major benefits include: high-quality testing process through scenario design of test cases (test descriptions) that are easy to review by non-testing experts. The TDL language was designed on three central concepts [10]: (1) a Meta-Modeling principle that expresses its abstract syntax, (2) a user-defined Concrete Syntax for different application domains, and (3) the TDL semantics that can be associated to the meta-model elements. Any minimal TDL specification consists of the following major elements:

- A set of Test Objectives that specify the reason for designing either a Test Description or a particular behavior of a Test Description. It can be written as a simple text in NL and it can be complemented with tables and diagrams;
- A Test Configuration, which is a set of interacting components (tester and SUT) and their interconnection;
- A set of Data Instances used in the interactions between components in a test description; and
- A set of Test Descriptions to describe one or more test scenarios based on the interactions of data exchanged between tester and SUT.

In order to obtain the TC (TDL specification) from the TP (TTCN-3 modules), we developed transformation rules to define TC elements from the TTCN-3 TPs'. These rules are meant for human processing; they are based on the equivalence between elements of both languages. The rules aim to remodel the TTCN-3 modules into more abstract TDL elements. The language-sensitive editor understands the concrete TDL syntax, based on the TDL meta-model.

Next, we show how each TDL element is derived from its corresponding TTCN-3 module by applying these rules. However, extracting the TDL Test Objectives cannot be rule-based since the TTCN-3 TPs do not have a concrete representation of the Test Objective. Nevertheless, the test objectives can be extracted from the legacy TCs and copied in TDL corresponding elements.

3.2.1 Remodel Test Data Sets

The concrete data definition, stored in the TTCN-3 *Test Data* module (TestData.ttcn3), is mapped to TDL Data Instances using TDL elements that link the data aspects between TDL and TTCN-3. These Data Instances are grouped in Data Sets and are considered as abstract representation of the corresponding concepts in a concrete type system.

3.2.2 Remodel Test Configuration

In a TDL specification, the Test Configuration element consists of a Tester, SUT components and a Gate. The corresponding TTCN-3 *Component* module contains equivalent objects with many more details. Specifically, it consists of a set of interconnected test components with well-defined communication ports and an explicit test system interface. TDL does not have a *receive* construct, instead it uses a *send* construct for the interaction between a Tester and the SUT. Therefore, the mapping of TDL Tester and SUT components is validated with the TTCN-3 interaction.

3.2.3 Remodel Test Description

The Test Description element in the TDL specification language defines the TC behavior. The enclosed scenario is mainly composed of actions and interactions between the Tester and the SUT components.

In the TTCN-3 *Test Behavior* module, the action is a function implementation or physical setup. The interaction is represented as a message being sent (from a source) or received (from the target). We remodeled the interaction and the action to their equivalent in TDL by applying the rules listed in Table 2. In the *Test Behavior* module, numerous sequences of events are possible due to the reception and handling of communication timer events. The possible events are expressed as a set of alternative behaviors and denoted by the TTCN-3 *alt* statement. Each TTCN-3 object in the *Test Behavior* is remodeled to an equivalent TDL construct by applying the transformation rules. In our experimentation, we used a TDL Editor to edit and validate the syntax of the TDL specifications.

Table 2. Transformation rules from TTCN-3 to TDL based on the proposed concrete syntax.

TDL Meta-model elements (abstract syntax)	TTCN-3 statements	Our proposed TDL concrete syntax	Description of transformation from TTCN-3 to TDL
TestConfiguration	module <tc_name> { }	Test Configuration <tc_name>	Map to a Test Configuration statement with the name < td_name >
GateType	type port <port_type> message { }	Gate Type <port_type> accepts <Data_Set_name>	Map to a Gate Type statement with the name <port_type> that accepts Data Set elements
ComponentType	type component comp_type{ port <port_type> <port_name>; }	Component Type <comp_type> { gate types : <port_type>	Map to a Component Type statement with the name <comp_type> and associate a <port_type> to it.
		instantiate <comp_instance> as Tester of type <comp_type> having { gate <gate_name> of type <port_type> ; }	
ComponentType	type component system_comp_type{ port <port_type> <port_name>; }	Component Type <comp_type> { gate types : <port_type>	Map to a Component Type statement with the name <system_comp_type> and associate a <port_type> as a port of the test system interface to it.
		instantiate <system_comp_type> as SUT of type <comp_type> having { gate <gate_name> of type <port_type> ; }	
Connection	map (mtc: <comp_type>, system <system_comp_type >)	connect <comp_type> to <system_comp_type >	Map to a connect statement where a test component is connected to test system component.
TestDescription	module <td_name> { import from <dataproxy> all; import from <tc_name> all; }	Test Description(<dataproxy) <td_name> { use configuration: <tc_name>; { } }	Map to a Test Description statement with the name <td_name >. The <DataProxy> element passed as formal parameters (optional) is mapped from an import statement of the <DataProxy> to be used in the module. The import statement of the Test Configuration <tc_name> is mapped to use configuration

TDL Meta-model elements (abstract syntax)	**TTCN-3 statements**	**Our proposed TDL concrete syntax**	**Description of transformation from TTCN-3 to TDL**
			property that is associated with the 'TestDescription'
Alternative Behaviour	alt { }	alternatively { }	Map to alternatively statement
Interaction	<comp_name_source>.send(<concreteData>)	<comp_name_source> sends instance < data_name > to <comp_name_target>	Map to a sends instance statement with respect to the sending component
	<comp_name_source>.receive(<concreteData>)	<system_comp_name_source> sends instance < data_name > to <comp_name_target>	Map to a sends instance statement when the sending source is SUT component
VerdictType	verdicttype <verdict_value>	Verdict <verdict_value>	Map <verdict_value> that contains the values: {inconclusive, pass, fail}to its corresponding value
TimeUnit	time_unit {1E-9,1E-6, 1E-3, 1E0, 6E1, 36E2	Time Unit <time_unit>	<time_unit> contains the following values: {tick,nanosecond,microsecond, miliisecond,second,minute,hour }
VerdictAssignment	setverdict (<verdict_value>)	set verdict to <verdict_value>	Map to a set verdict to statement
Action	function <action_name>()	perform action <action_name>	Map to perform action statement
Stop	stop	stop	Map to a stop statement within alternatively statement
Break	break	break	Map to a break statement within alternatively statement
TimerStart	<timer_name>.start(time_unit);	start <timer_name> for (time_unit)	Map to a start statement
TimerStop	<timer_name>.stop;	stop <timer_name>	Map to a stop statement
TimeOut	<timer_name>.timeout;	<timer_name> times out	Map to a times out statement
Quiescence/Wait	timer <timer_name> <timer_name>.start(time_unit); <timer_name>.timeout	is quite for (time_unit) waits for (time_unit)	Map to is quit for statement or to waits for
InterruptBehaviour	stop	interrupt	Map to interrupt statement
BoundedLoop Behaviour	repeat	repeat <number> times	Map to repeat statement. The repeat is used as the last statement in the alternatively behavior.
DataInstance	type_keyword <data_name>	Data Set <any_name> { instance <data_name> }	Map any <type_keyword> to an instance and group it in Data Set element

3.2.4 Model Real-Time in TDL

Previously, we mapped real-time elements enclosed in legacy TP to TTCN-3 timers' objects. TDL defines time package to express:

- A Time instance or time duration is expressed by a real positive value. The unit of Time instance is described by predefined instances for the TimeUnit. There are two Time Operation that can be applied on Tester components or on Tester gate:
  - Wait: defines the time duration that a Tester waits; and
  - Quiescence: defines the time duration during which a Tester shall expect no input from a SUT;
- A Time Constraint element resides within a test description; it is used to express timing requirements over two or more atomic behavior elements; and
- A Timer element defines a timer that is used by the following Timer Operation:
  - TimerStart: it sets the period property to define the duration of the timer from start to timeout. The Timer changes from idle to running state;
  - TimerStop: it stops a running Timer, the state of the Timer becomes idle; and
  - TimeOut: it specifies the occurrence of a timeout event when the period set by the TimeStart operation has elapsed. The Timer changes from running to idle state.

The TDL time package can be modeled from the TTCN-3 timer operations as shown in Table 2 that illustrates the transformation rules from TTCN-3 to TDL based on the versions in [9, 10].

4 Conclusion

The modernization of software tests to a new platform is often pressured by business requirements to reduce the cost and effort of testing. In this project, we automatically restructured legacy TPs, written as Ant/xml files into the TTCN-3 language that provides strong typing, structured constructs and for modular code. This migration enforced coding standards and offered a more readable, simple to modify and easy to understand test code. Next, we reengineered the code and data to a higher level of abstraction to obtain (model-driven) TPs. Our overarching goal is to support test automation, to reduce the effort involved in testing and to lower maintenance cost while meeting software tests' evolution requirements.

Annex 1

The migrated Heart Rate Test Procedure in TTCN-3

```
module HRDeployTest {
   type record HeartRateType {
     charstring heartRateSignal   }
   template HeartRateType TemplateType := {
      heartRateOutOfRangeSignal := "48"  }
   template HeartRateType TemplateType := {
      heartRateWithinRangeSignal := "70"  }
   type record AlarmType {
      charstring alarmOFFSignal
      charstring alarmONSignal   }
   template AlarmType TemplateType := {
      alarmOFFSignal := "OFF"
      alarmONSignal := "ON"  }
   type port defaultGT message {
     inout HeartRateType;
     inout AlarmType; }
   type component Patient {
     port defaultGT gPatient ;       }
   type component Monitor {
     port defaultGT gMonitor ;       }

   testcase _TC () runs on Patient {
      map (mtc:gPatient, system:gMonitor);
      gPatient.send(heartRateOutOfRangeSignal);
      alt {
            [] gPatient.receive(alarmOFFSignal)
               setverdict(pass);
            [] gPatient.receive
               setverdict(fail);
       }
      gPatient.send(heartRateWithinRangeSignal);
      alt {
            [] gPatient.receive(alarmONSignal)
               setverdict(pass);
            [] gPatient.receive
               setverdict(fail);
       }
  }
  control {
      execute(_TC());   }
 }
```

References

1. De Lucia, A., Francese, R., Scanniello, G., Tortora, G.: Developing legacy system migration methods and tools for technology transfer. Softw. Pr. Exp. **38**(13), 1333–1364 (2008)
2. Al-Azzoni, I., Zhang, L., Down, D.G.: Performance evaluation for software migration. In: ACM SIGSOFT Software Engineering Notes, vol. 36, no. 5, pp. 323–328. ACM, March 2011
3. Ant, A.: The Apache Software Foundation (2007). http://ant.apache.org/
4. Behm, A., Geppert, A., Dittrich, K.R.: On the migration of relational schemas and data to object-oriented database systems, pp. 13–33. Universität Zürich. Institut für Informatik (1997)
5. Bisbal, J., Lawless, D., Wu, B., Grimson, J.: Legacy information systems: Issues and directions. IEEE Softw. **5**, 103–111 (1999)

6. Brooks Jr., F.P.: The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, Noida (1995)
7. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Elsevier, New York (2002)
8. Wong, W.E., et al.: A study of effective regression testing in practice. In: The Eighth International Symposium on Software Reliability Engineering, 1997. Proceedings. IEEE (1997)
9. ETSI, E. 201 873-7 V3. 4.5.1 (2013-04): Methods for Testing and Specification (MTS). The Testing and Test Control Notation version, 3. http://www.etsi.org
10. ETSI ES 203 119 (stable draft): Methods for Testing and Specification (MTS); The Test Description Language (TDL), 25 Sept 2013. http://www.etsi.org
11. Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jézéquel, J.M.: Model-driven engineering for software migration in a large industrial context. In: Engels, G., Opdyke, B., Schmidt, D. C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 482–497. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75209-7_33
12. Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Müller, H., Mylopoulos, J.: Code migration through transformations: an experience report. In: CASCON First Decade High Impact Papers, pp. 201–213. IBM Corp, November 2010
13. Lempia, D.L., Miller, S.P.: Requirements engineering management handbook. National Technical Information Service (NTIS), 1 (2009)
14. Müller, B.: Reengineering: Eine Einführung. Springer, Heidelberg (2013)
15. Parveen, T., Tilley, S., Gonzalez, G.: A case study in test management. In: Proceedings of the 45th Annual Southeast Regional Conference. ACM (2007)
16. Parveen, T., Tilley, S.: When to migrate software testing to the cloud? In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW). IEEE (2010)
17. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley Professional, Boston (2003)
18. The Design of the Test Description Language (TDL). https://www.swe.informatik.uni-goettingen.de/research/design-test-description-language-tdl-stf-454
19. Wagner, C.: Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems. Springer, Wiesbaden (2014). doi:10.1007/978-3-658-05270-6
20. Wagner, C., Margaria, T., Pagendarm, H.-G.: Analysis and code model extraction for C/C++ source code. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009). IEEE Computer Society, Washington, DC (2009)
21. Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: TTCN-3 by Example. An Introduction to TTCN-3, Second Edition, pp. 7–24 (2011)