ETSI GR NFV-TST 011 V1.1.1 (2019-03)

Network Functions Virtualisation (NFV); Testing; Test Domain and Description Language Recommendations

Reference

DGR/NFV-TST011

Keywords

language, NFV, testing

Contents

Intellectual Property Rights

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV).

Modal verbs terminology

Executive summary

The present document proposes a model of the NFV test domain and recommends requirements for a test Domain Specific Language (DSL) to manipulate it. In the context of NFV, a network service is supplied by multiple vendors and each vendor has its own test technology, interfaces into the system under test, and test languages (usually GPLs like Java®, Ruby®, Python®, etc.) In order to create a common test language, the test cases follow a standardized test case model that the language can manipulate, and that can be implemented within individual test technologies. The model includes shareable and reusable artefacts tied to the test domain: execution flow, data, abstract resources, environment, etc.

Integration of multiple test technologies is only possible by a system that can accept contributions of test resources from multiple parties. These contributions may include lab resources, test APIs, test data, high-level function libraries, test execution platforms, etc. The test environment is then constructed dynamically from these contributions. To allow the dynamic nature of the test environment, it is necessary for the test case to be decoupled from specific resource contributions and express the test process in terms of resource abstractions. Mapping these abstractions to concrete resources is the job of a Dynamic Resource Management (DRM) system. This is done by creating an environment resource meta-model available to the test case developers at design time. The meta-model is then used for creation of specific environment instance models at runtime. Each environment instance includes dynamic resource contributions to which resource abstractions are mapped.

Introduction

With the advent of NFV, the industry is experiencing the following transformative challenges:

- Multiple contributions to a network service
- Open collaboration
- Shift away from dedicated resources (sharing of resources)
- Shift of integration responsibility from vendors to service providers (or their agents)
- Test cases/plans can be repeated multiple times, making reuse critical

To address these challenges and to encourage collaboration, the present document provides recommendations for NFV test domain modelling and a Test DSL that does not force vendors/participants to change their test technology/language and enables efficient utilization of resources. To enable reuse, the model also decouples test data and test environment from the test case and uses dynamic allocation of test resources.

1 Scope

The present document proposes a model of the NFV test domain and recommends requirements for a test Domain Specific Language (DSL) to manipulate it. The description includes an NFV test automation ecosystem that facilitates interaction among NFV suppliers and operators, based on the DevOps principles.

The NFV test domain contains:

- System Under Test (SUT): Network Functions (NF), Network Functions Virtualisation Infrastructure (NFVI) and network services.
- Test Resources: tools or instrumented NF's and NFVI elements that test cases can interface to manipulate the SUT.
- Test Execution Flow: controlled and uncontrolled state transitions.
- Test case configuration data and parameters: test-resource-specific and non-test-resource-specific.

The present document explores the following attributes to enable efficient multi-supplier NFV interaction:

- Reusability of test plans, test cases and test resources.
- Abstraction of test data.
- Decoupling of test case from the test environment.
- Use of test resource abstractions in place of concrete resources.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GS NFV-TST 001 (V1.1.1): "Network Functions Virtualisation (NFV); Pre-deployment Testing; Report on Validation of NFV Environments and Services".

[i.2] ETSI GS NFV-MAN 001 (V1.1.1): "Network Functions Virtualisation (NFV); Management and Orchestration".

[i.3] ETSI GS NFV-SOL 003 (V2.3.1): "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Or-Vnfm Reference Point".

[i.4] ETSI GS NFV-IFA 006: "Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Vi-Vnfm reference point - Interface and Information Model Specification".

[i.5] ETSI GS NFV 003: "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI GS NFV 003 [i.5] and the following apply:

execution engine: means by which a program can be executed on the target platform using the two approaches of interpretation and compilation

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI GS NFV 003 [i.5] and the following apply:

DSL	Domain Specific Language
GPL	General Purpose Language
HLF	High-Level Function
TEP	Test Execution Platform

4 Test Domain

4.1 Overview

The NFV test domain is a set of artefacts and systems for testing NFV-based solutions. NFV introduced the concept of "dynamically configurable and fully automated cloud environments" (https://www.etsi.org/technologies/nfv) for network functions. The present document models the NFV test domain as a set of abstractions so that the same level of flexibility is available in testing those network functions. Figure 1 is included to illustrate relationships among artefacts discussed in clause 4; the NFV Test Domain model is described in more detail in clause 6. In addition, the present document proposes requirements for a Domain-specific language (DSL) to manipulate that test domain. Using these models and recommended requirements, suppliers and service providers are able to leverage different test technologies, dynamically allocate test resources, and reuse test plans, test cases, and Test Execution Platforms (TEPs). In the present document, a test case always refers to a computer program that can be executed by a test automation system.

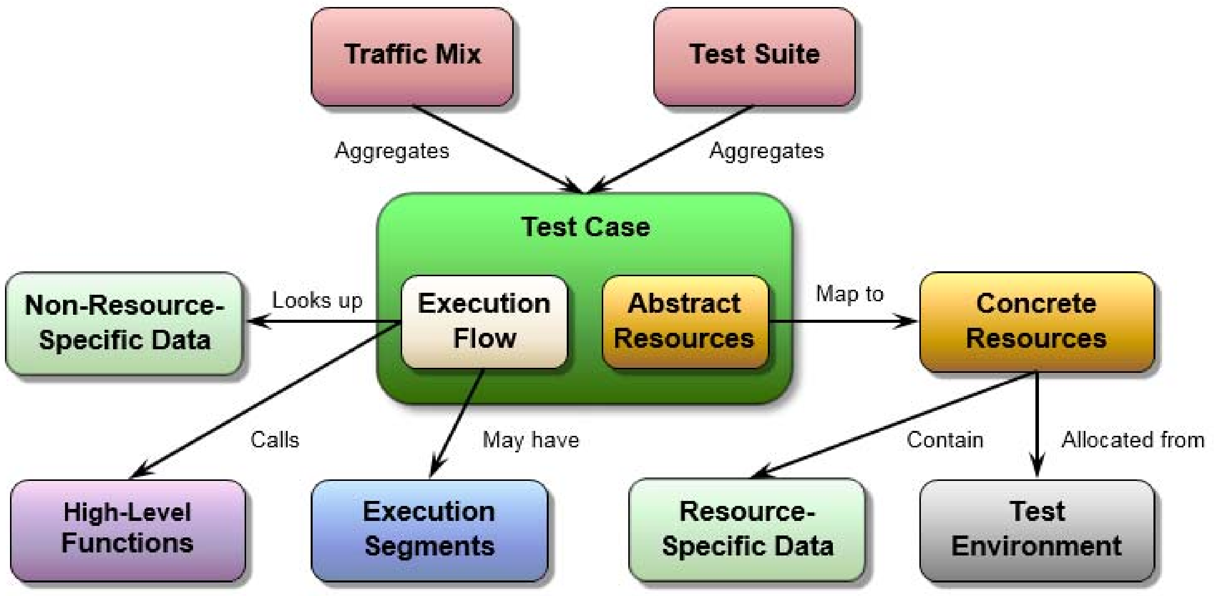


Figure 1: NFV Test Domain Artefact Relationships

The NFV test domain is comprised of:

- System Under Test (SUT) [i.1]: Network Functions (NF), Network Functions Virtualisation Infrastructure (NFVI), and network services
- Test Case Resources: tools or instrumented NF's and NFVI elements that test cases can interface to manipulate the SUT
- Execution Flow: controlled and uncontrolled state transitions
- High-Level Functions
- Test case configuration data and parameters: test-resource-specific and non-test-resource-specific
- Execution Segments
- Test Environment
- Test Suites & Traffic Mixes

4.2 Test Case Resources

In order to be tested, the SUT exposes a set of interfaces over which test interactions happen. These interfaces vary in the degree of complexity and may include entire protocol stacks. The test drivers communicate with the SUT by sending and receiving encoded messages over one or more of these interfaces. This means that the implementation of the interface is also present on the test side.

It is therefore necessary for the executing test case to create or otherwise acquire one or more objects that implement the SUT interfaces and use them to send and receive messages to and from the SUT. These objects in turn expose their own interfaces to the test case to allow the test to manipulate the message flow more easily. These objects will be referred to as abstract resources. In essence an abstract resource is an instance of an SUT interface that provides its own API to the test.

Abstract resources are test case-facing abstractions that utilize units of lab equipment, software, and/or data to do real work. These units will be referred to as concrete resources. In the context of a shared lab, concrete resources are shared among multiple users and are allocated to specific test cases for the duration of the test. This guarantees that the test case gets exclusive access to concrete resources required for its execution. Some examples of concrete resources include instances of instrumented or stubbed out MANO components, VNFs, VNFCs, etc. Availability of concrete resources is generally limited and concurrently executing test cases contend to gain access to them.

The relationship between abstract and concrete resources is typically many-to-many, and the mapping between them is described in clause 6.3. The test DSL is expected to enforce proper resource declaration, preclude any access to concrete resources outside the resource management system, and provide the user with an intuitive way to declare and manipulate abstract resources. Once an abstract resource is mapped to an allocated concrete resource they form a single entity used by the test case to interact with the SUT. This entity will be called test case resource as illustrated in Figure 2.

4.3 Test Execution Flow

As the test case executes, every resource goes through a sequence of states, reflecting the SUT functionality being tested. These sequences range in complexity from trivial to very complex. The test case may be interested in some but not all of these states. For example, if a resource is running a protocol stack, unless protocol conformance is being tested, most low-level messaging is of no interest to the test case. It may only be interested in the successful or unsuccessful outcome of such messaging. The degree to which the test case has visibility of the resource state can vary for the same type of resource depending on the test scenario. Consequently, it is necessary for a mechanism for different levels of control over the resource state to be present. The test case should have the ability to "take over" the resource state transition when necessary and let the resource run its own state machine at other times. The resource states controlled by the test case are referred to as controlled states.

Controlled states of all test case resources at any given time form the state of the test case. Test case initiates state transitions by sending or receiving and verifying messages to and from the SUT on any of the test case resources. For example, if the SUT is a VNFM implementation and an Instantiate VNF request is sent from the NFVO resource, the test case can verify receiving a Grant VNF Lifecycle Operation request by the NFVO resource and after acknowledging it with a Grant VNF Lifecycle Operation response, verify that the Allocate Resource request is received by the VIM resource. Running a single state machine per test case for controlled states while letting individual resources run their own state machines for uncontrolled states provides an intuitive and flexible framework for SUT interface interworking.

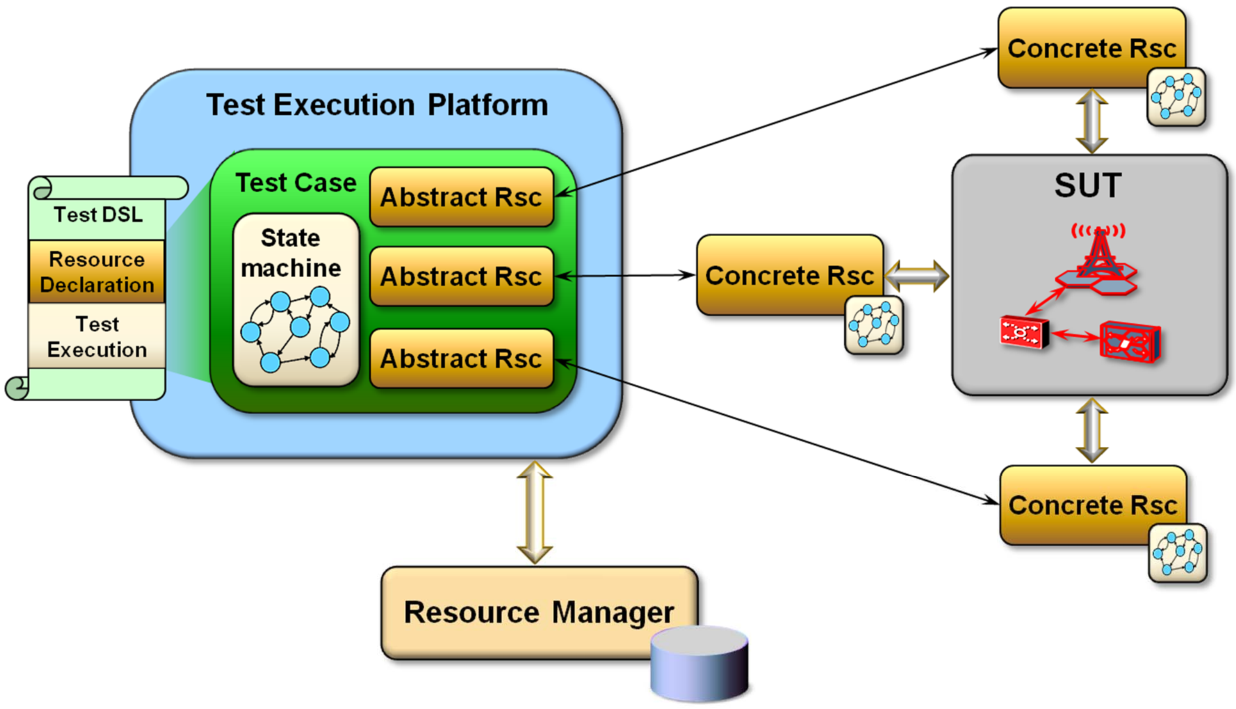


Figure 2: Test Case Resources and State Machine

Any controlled state can be passing or failing. Criteria for failing the test are evaluated by the test case at every controlled state. If failure criteria are satisfied the test fails immediately. Otherwise it continues until it reaches a final state at which point it passes. If a test case does not reach a final state it gets killed by the TEP once sufficient time has passed to conclude that the test behaviour is abnormal. Regardless of the type of test case termination (passed, killed, crashed, hung and killed by the TEP), the allocated concrete resources are returned into their initial state and released. Since no assumption can be made about normal vs. abnormal test case termination, the responsibility of resource clean-up and release cannot be assigned to the test case and should lie with individual resource implementations.

4.4 High-Level Functions

A test scenario may also require a series of controlled state transitions involving multiple resources whose details are outside the main focus of the test. Continuing with the previous example, if the main focus of the test is verification of VNFM configuring the newly created VNF with deployment-specific parameters, the details of the VNF instantiation are not important to the test scenario and can be grouped into a single action. These actions will be referred to as high-level functions. They capture frequently repeated state transition sequences and encapsulate them into invokable, parameterizable units of functionality. High-level functions can also call other high-level functions. This provides a flexible mechanism for control of granularity for different parts of the execution flow. High-level functions are reusable across multiple test cases and activities.

4.5 Test Case Data

Many test cases utilize data that are not explicitly defined in the body of the test case. Separation of the execution flow and the data that can parameterize the execution flow allows greater flexibility of test case design and reuse of the same data sets by multiple tests.

Some data are resource-specific and come from test case resources. Since the test case only manipulates abstract resources and has no prior knowledge of what concrete resource may be allocated to it, any attributes of the concrete resources used in the test case are acquired at run time. For example, in order to verify that the VNFM under test sends the correct VNF identifier in the alarm notification to the NFVO, it is necessary for the test case to know the VNF identifier of the VNF generating the alarm. Until a specific VNF test resource instance is allocated to the test case, its identifier is not known. The test case should therefore be able to obtain the identifier from the VNF test resource at runtime to verify that the correct one is sent to the NFVO test resource.

Some data are not resource-specific and come from elsewhere. For example, default values for a set of protocol messages can be specified outside the test case. This allows the test case writer to only provide non-default values for message Information Elements (IEs). This also allows reuse of this data across a potentially very large number of test cases using the same protocol.

Test case data should be easily customizable. In the protocol defaults example there may be global defaults, defaults that a particular user team is using for their purposes, or even defaults for a particular test activity that further specialize test case writers' defaults. This can be accomplished by creating progressively specialized test case data hierarchies with every new child defining a nested scope within its parent's scope. Data lookup then follows the regular scoping rules.

4.6 Execution Segments

In some cases the test execution flow can be separated into individually executable segments. For example, many control-plane test scenarios are designed to have segments of signalling separated by periods of inactivity. These periods of inactivity represent voice or data transmission and are commonly referred to as hold time. Hold time may be in orders of magnitude longer that the active segments of the test case and a very large number of test cases can be holding at the same time. It is therefore important to have a mechanism for suspending execution of the test case and releasing associated computing resources (not test case resources) for the duration of the hold time.

Unlike high-level functions execution segments are only defined within the scope of an individual test execution flow. They are not shareable artefacts and cannot be invoked by the test case. They only provide the ability to the TEP to execute parts of the test case individually. High-level functions of various degree of granularity can be invoked from within execution segments.

Segmentation also provides the ability to run parts of the execution flow in a different order. A separate model specifying desired combinations of the segments and their permutations can be defined and provided to the TEP. The TEP can then execute the segments in the specified order. This can provide substantial time and effort savings compared to writing a separate test case for every desired permutation.

4.7 Test Environment

Test Environment is defined in ETSI GS NFV-TST 001 [i.1], clause 4.2. The present document introduces the concepts of Abstract and Concrete test environments that enable reuse. Similarly to test case resources, environments can also be abstract and concrete. Abstract environments consist of abstract resources, and concrete environments consist of concrete resources. Abstract resources are mapped to concrete resources by the resource management system.

Before concrete resources can be used in a test case (or a group of test cases within the same test activity) they are allocated to that test activity and provisioned. In addition to specifying required abstract resources, abstract environment definition also specifies Provisioning Data that is applied to allocated concrete resources to complete concrete environment instantiation.

4.8 Test Suites & Traffic Mixes

In a typical test scenario test cases are not executed by themselves. They run as a part of either a test suite or a traffic mix.

Test suites aggregate test cases that verify a particular set of functional requirements. Test cases in a test suite are executed once and success and failure are considered on individual test case basis. Test suites impose sequential or parallel mode of test case execution, or define flows that combine both. Test suites can also set context (e.g. test data) for parts of such flows.

Traffic mixes aggregate test cases that verify non-functional requirements, such as performance, robustness, etc. Test cases in a traffic mix execute repeatedly, each at its own rate. Together, they emulate real network traffic where multiple activities happen at the same time. Success and failure are considered on the entire traffic mix execution and a certain number of individual test case failures is typically expected. For example, a five 9's availability metric allows one out 100 000 test cases to fail in a successful run.

Traffic mixes define relative frequencies of different network activities and the overall rate of network traffic. Both activity distribution and the overall rate can change over time.

5 Reuse Guidelines

5.1 Overview

NFV solutions are composed of contributions from multiple suppliers. As such, test cases or entire test plans may be repeated multiple times, making reuse critical. The NFV test domain model (as defined in the present document) enables reusability through decoupling, abstraction and modularization.

5.2 Environment Decoupling

Decoupling the test environment from test cases is achieved by strict separation of abstract and concrete resources and using a dynamic resource management system to map the abstract resource space to the concrete resource space.

The relationship between test cases and their environments is many-to-many, which means that multiple test cases with the same resource requirements can execute on the same test environment and that the same test case can execute on multiple environments.

The resource management system manages the available test resources and accepts contributions of resources from various resource contributors. From these contributions it builds a dynamic model of the concrete resource space. The model adapts dynamically to changes in resource contributions. This dynamic model is used by the resource management system to find a concrete resource for an abstract resource request from the resource consumer.

This dynamic model is an instance of a meta-model that specifies resource abstractions and their relationships. This meta-model is defined statically and describes a family of resource models. The resource abstractions specified by the meta-model are visible to resource consumers. A resource consumer can request any resource abstraction without the knowledge of the concrete resource space.

When a test activity needs a concrete environment to run on, an abstract environment gets designed to describe it. This abstract environment is defined in terms of specific resource abstractions provided by the meta-model. This abstract environment is a reusable artefact that multiple test activities can use to build concrete environments.

Environment meta-models and abstract environment definitions are carefully designed by solution and automation architects and managed by the test environment management service. The end user only deals with the namespace for resource abstractions (like a pool of VIMs) they use in the test case.

5.3 Resource API Decoupling

Test resources are contributed from multiple sources and, in the context of multi-organizational collaboration, different test technologies come from different suppliers. The resource APIs are therefore decoupled from any resource implementations and are defined outside of any specific test technology. Test cases written in a test DSL have no knowledge of the test resources but as long as the APIs are defined outside of any specific resource contribution, the test cases can be written against these APIs, compile-time error checking can be performed, and code assist/code completion can be provided to the user.

5.4 High-level Function Decoupling

High-level Functions (HLFs) provide a high degree of reuse and a single point of truth (written or fixed once - used everywhere). Most existing test technologies will have some form of HLFs already implemented that may be leveraged. Similarly to test resource contributions, this necessitates having an externally defined contract for using these HLFs with which all implementations comply. It is the responsibility of individual execution engines to compile HLF API calls into calls on specific HLF implementations.

5.5 Test Data Decoupling

As indicated in clause 4.5, the test domain definition includes a mechanism for separation of the test case execution flow from the data used to parameterize the test.

Resource-specific data come from the concrete resource allocated to the test case and is not known in advance to the test case designer. Non-resource-specific data is a part of the context for the test case execution and forms a hierarchy that can be easily customized. The customization may include progressive specialization from global set of values down to values for individual test activities. The data is dynamically looked up by the test case using fully qualified symbols as keys. Each fully qualified name represents a path from the root of the data tree to the node holding the value.

The symbol lookup functionality is provided by the TEP and can have its own implementation. Test data is provided to the TEP as metadata that can be converted into any TEP-specific format or integrated into already existing functionality.

In order to use test case data, it is necessary for the test case designer to provide names for individual data elements. In the case of resource-specific data the problem of checking validity of these names at compile time and providing code assist/code completion functionality to the user is easily solved by making getters for any data elements the resource exposes to the user a part of the resource API.

For non-resource-specific data this problem is more challenging. Since this type of data is a part of the context for the test case execution it is not known at design time. If a test case tries to look up a symbol that is not present in the data it will be a run-time error. In addition, the namespace can be very large and fully qualified symbol names could also be long and prone to spelling errors. The problem can be rectified if a schema for test data is made available to the test case at design time. Symbol names will be validated against this schema, auto-proposed, and auto-completed.

6 Recommended Models

6.1 Overview

In order to create a common test language, the test cases follow a standardized model that the language can manipulate, and that can be implemented within individual test technologies. The model includes shareable and reusable artefacts tied to the test domain:

- execution flow;
- data;
- abstract environment definitions;
- etc.

6.2 Test Case Model

The diagram in Figure 3 shows test case elements and their relationships. The test case has a test script and uses test data, high-level function libraries and segment ordering data. The test script in turn has resource declaration and execution flow definition parts. The resource declaration section aggregates a set of abstract resources that realize their respective APIs. The execution flow can call the abstract resource APIs. It can also invoke higher-level functions. Higher-level functions can invoke other higher-level functions and call abstract resource APIs. The execution flow uses the test data model for dynamic symbol lookup. The execution flow aggregates execution segments that are ordered according to the segment ordering data model.

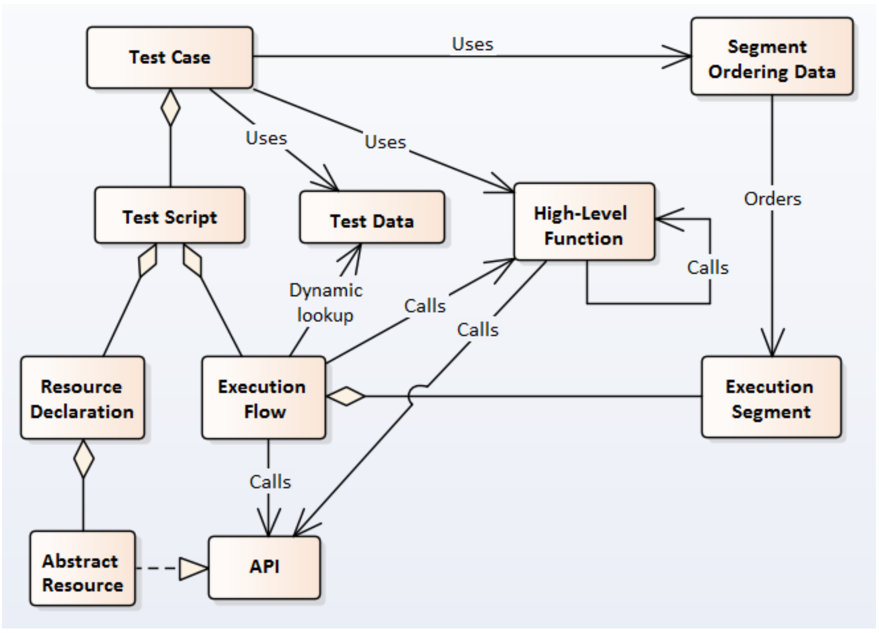


Figure 3: Test Case Elements

6.3 Test Environment Model

The diagram in Figure 4 shows how test environments relate to test cases. Abstract test environments aggregate abstract resources, which in turn are mapped to concrete resources. There are three major types of concrete resources: elements of the SUT, test tools used to interact with the SUT and reservable data. An example of reservable data could be a license key with a limited set of instances.

The link between the abstract and the concrete environments is provided by the Environment Meta-model. It models the concrete resource space as a set of reservable entities and provides a number of abstractions (e.g. pools) that can be referenced from the abstract environment definition. The meta-model captures domain knowledge about the concrete resource space and considerably simplifies abstract environment definition.

Before concrete resources can be used in a test case - or a group of test cases within the same test activity - they are provisioned (configured). Abstract environment definition specifies Provisioning Data that is applied to allocated concrete resources to complete concrete environment instantiation.

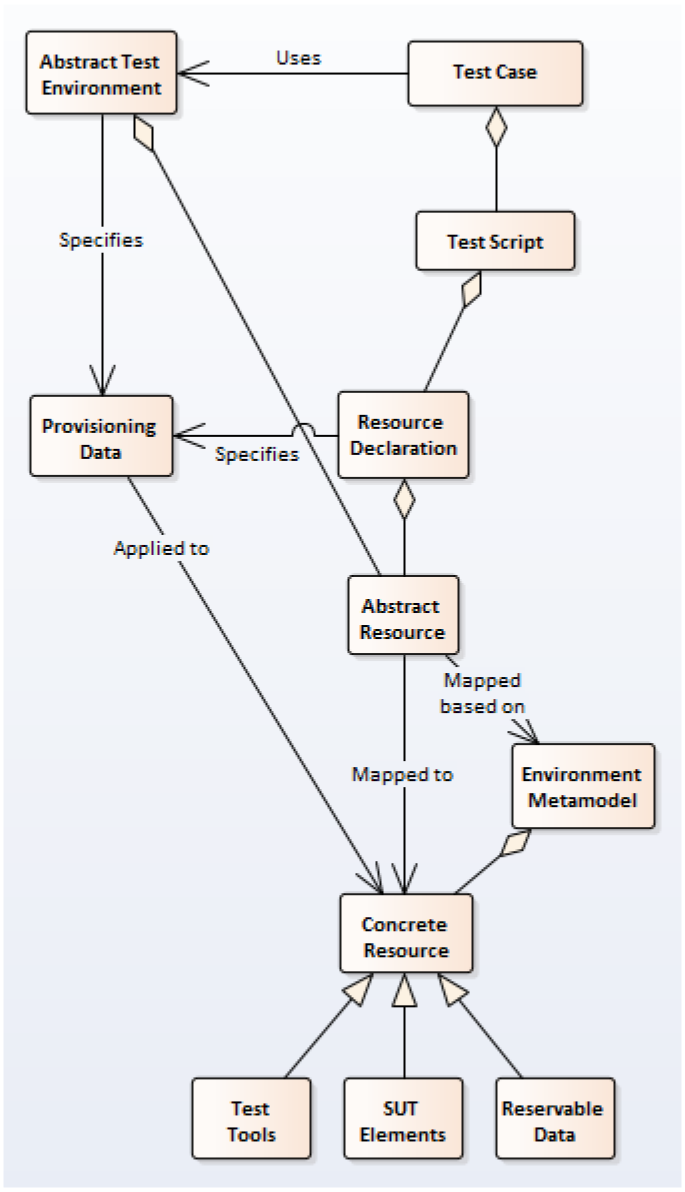


Figure 4: Test Environment in Relation to Test Cases

6.4 Test Scenarios

The diagram in Figure 5 shows test suites and traffic mixes in relation to the test cases. Test suites and traffic mixes aggregate test cases and verify functional and non-functional requirements respectively. All requirements define a set of attributes or keywords that the test cases are labelled with. Dynamic suites and traffic mixes are constructed from test cases selected based on these labels. A natural constraint is that all test cases within the same test suite or traffic mix are able to execute on the same test environment. Test cases with non-intersecting environment requirements cannot be part of the same test suite or traffic mix.

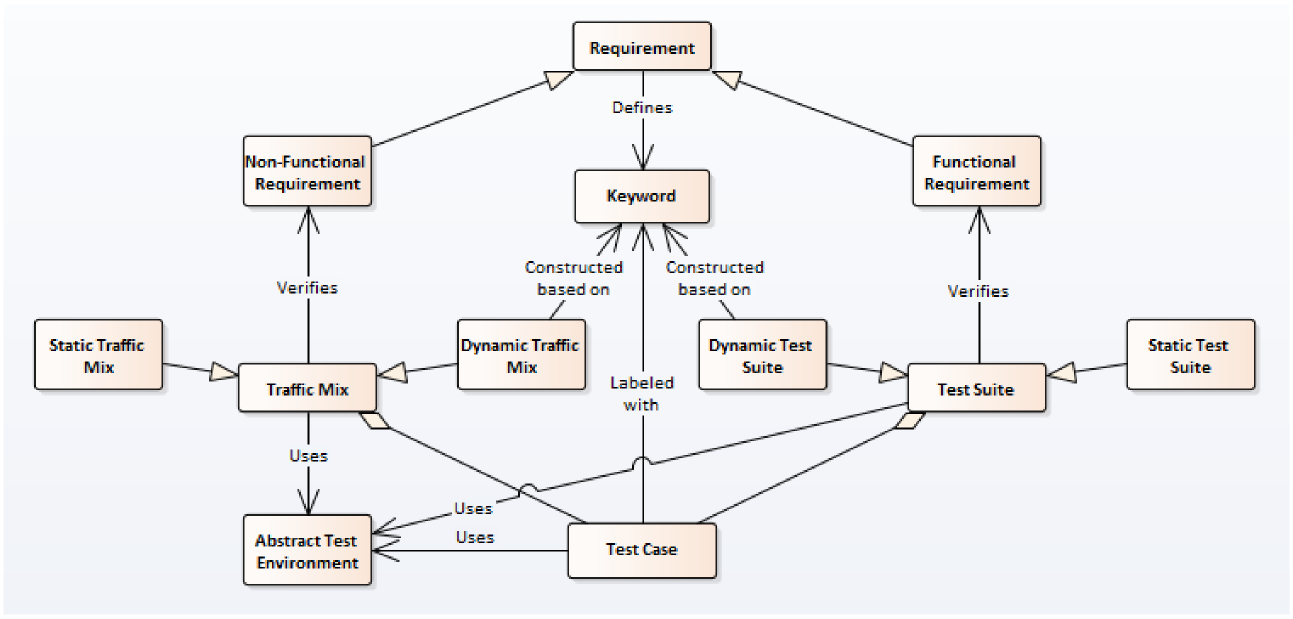


Figure 5: Statically and Dynamically Defined Test Suites and Traffic Mixes

6.5 Full Domain Model

Domain diagrams in Figures 3 through 5 combine to form a recommended test case domain model as shown on Figure 6.

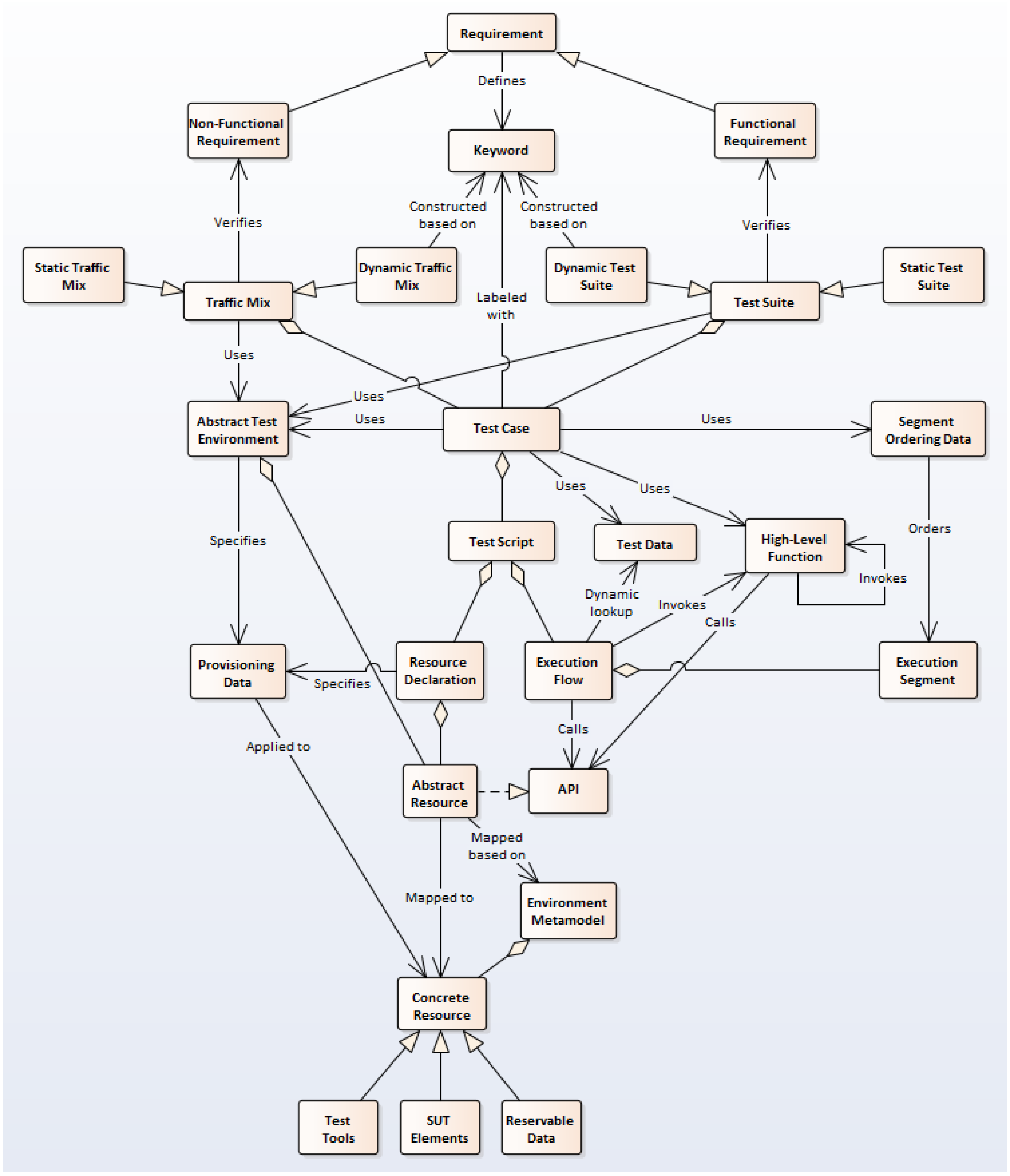


Figure 6: Recommended Test Case Domain Model

7 Test DSL

7.1 Overview

Vendors have their own test technologies, interfaces into the system under test, and test languages (usually GPLs like Java®, Ruby®, Python®, etc.). In order to create a common test language, it is necessary for the test cases to follow a standardized test case model that the language can manipulate, and that can be implemented within individual test technologies. As stated in clause 6, the model includes shareable and reusable artefacts tied to the test domain: execution flow, data, abstract resources, environment, etc.

Integration of multiple test technologies is only possible by a system that can accept contributions of test resources from multiple parties. These contributions may include lab resources, test APIs, test data, high-level function libraries, test execution platforms, etc. The test environment is then constructed dynamically from various contributions. To allow the dynamic nature of the test environment, the test case are decoupled from specific resource contributions and express the test process in terms of resource abstractions. Mapping these abstractions to concrete resources is the job of a dynamic resource management system. This is done by creating an environment resource meta-model available to the test case developers at design time. The meta-model is then used for creation of specific environment instance models at runtime. Each environment instance includes dynamic resource contributions to which resource abstractions are mapped.

7.2 Test DSL Concepts

A test case is a computer program. This program has syntax and semantics. The semantics reflect the meaning of what the program does and the syntax describes a particular representation of this meaning. Many different syntactic representations may result in the same program semantics. As a clarification of scope, any examples of such representation are for illustration purposes only and are not meant to suggest any particular concrete syntax.

The task of writing a test case in a DSL is a task of building the semantic model of the test using that DSL syntax. This semantic model is an object model that captures what the test case is supposed to do when it executes.

In clause 6, the test case domain model was recommended to have the following elements: test execution flow, test case resources, test data, high-level functions, and test segment model. The test case semantic model defines a specific instance of the execution flow and manipulate instances of other test case elements (resources, data, etc.) It is important to note that these instances can have different implementations and can be written in different programming languages (GPLs or DSLs). The semantic model together with other test element instances form the conceptual model of the test. In other words the test case domain model is the meta-model of which the conceptual model is an instance.

A conceptual model can be constructed in several different ways. One way is to create a grammar for a particular DSL concrete syntax and generate a parser for this grammar. The resulting text-based DSL will need an IDE providing common services like code assist, code completion, and design-time compile error checking. As mentioned above, there can be many different concrete syntax representations for the same conceptual model. Another way is to use a projectional modeller or a set of modellers that will manipulate the conceptual model directly from the modeller UI (as illustrated in Figure 7).

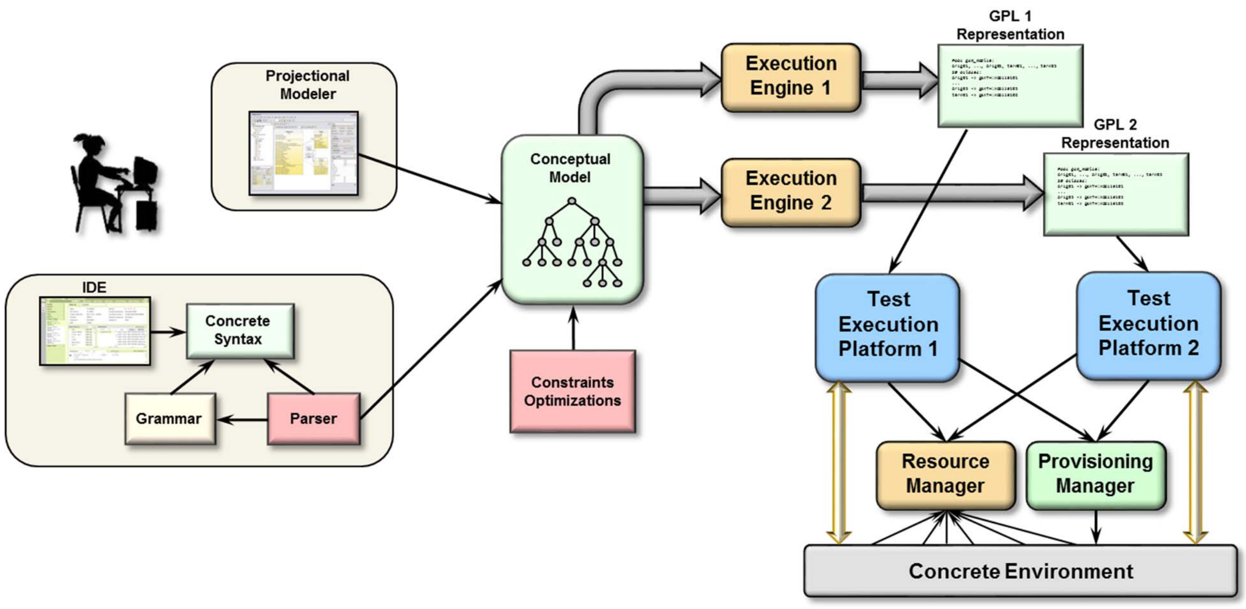


Figure 7: DSL Components

The same conceptual model can have multiple execution engines that generate GPL representations of the test for multiple target platforms. Figure 7 illustrates how this works: the same conceptual model uses two execution engines to generate programs in two different GPLs that execute on their respective test execution platforms sharing the same environment. The conceptual model provides clear separation of concerns between the language concrete syntax (or projectional modeller implementation), the resulting execution semantics, and code generation of various GPL representations.

7.3 Abstract Syntax meta-model

When a DSL test case is processed by the parser, a semantic model of the test case is generated. A different semantic model is created for each test case, depending on the meaning of the individual test program. All such semantic models have the same structural elements specified by the language grammar. In this sense they are all instances of a meta-model that models these structural elements and their relationships. This meta-model is the abstract syntax meta-model for the language.

For example, a semantic model may have any number of specific resource declaration elements, one for each declared resource. All such elements will have the same structure and will all be children of the resource declaration section element. The element structure and its relationship to its parent element is defined in the abstract syntax meta-model.

The relationships between the abstract syntax meta-model and test case semantic models are illustrated in Figure 8.

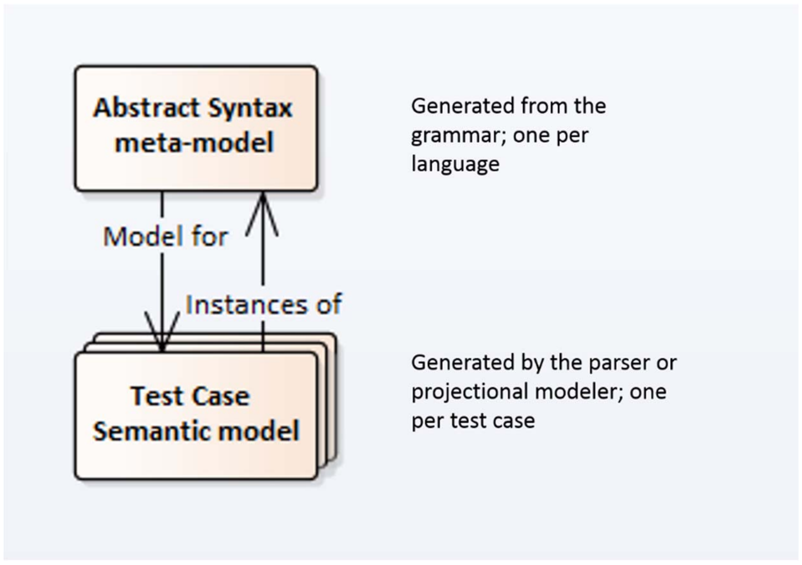

Abstract Syntax meta-model
Generated from the grammar; one per language
Model for
Instances of
Test Case Semantic model
Generated by the parser or projectional modeler; one per test case

Figure 8: Meta-model Relationships

7.4 Dynamically Loaded Constraints

Nodes of the abstract meta-model can have constraints that, in addition to syntax, determine the validity of the test case code. Test cases that are syntactically invalid or that contain code violating such constrains will produce compile-time errors. It is recommended that the test DSL takes advantage of this functionality to not only add statically defined constraints, but also validate the code against dynamically loaded artefacts. For example, in the resource declaration section of the test case, a URL can be specified that points to an environment meta-model against which the test case is written. The environment meta-model provides a namespace for abstract resources available from the test environment.

Once the environment meta-model is loaded and processed, the information of all available abstract resources and their APIs becomes a part of the test case's sematic model that imposes constraints on the resource declaration element of the abstract syntax meta-model. Thus, the check that the particular abstract resource is present in the test environment and that it realizes a specific API can be done at compile time. Violation of either of these two constraints should result in compile-time errors.

Likewise, the validity of API calls on abstract resources and high-level function calls based on resource APIs and high-level function contracts dynamically loaded into their respective modules of the test case semantic model can be verified at compile time. Violation of these constraints should also result in compile-time errors. It is important to note that even though the environment meta-model and the resource and high-level function contracts are loaded dynamically, they are statically defined artefacts.

7.5 Test Case Header

7.5.0 Introduction

The header describes the test case and its relationship to other artefacts the test case interacts with. It is recommended that the header has the elements described below.

7.5.1 Test Case Identifier

A test case identifier uniquely identifies the test case. Uniqueness cannot be enforced by the language and is a concern for test case management, which is outside the scope of the present document. However, since individual users are only in control of a specific namespace within their organization, separation of these namespaces are essential to avoid conflicts. It is recommended that such separation is achieved by the requirement that the test case identifier be a fully qualified name. In the service integration context, it is even more important to avoid name conflicts between different vendors.

Specifying the test case identifier in the test case header should be mandatory.

7.5.2 Test Case Description

The test case description element is a natural language description of the test case. A meaningful description simplifies maintenance of the test cases and can be utilized by maturing Natural Language Processing/Artificial Intelligence technologies.

Specifying the test case description in the test case header should be mandatory.

7.5.3 Custom Test Case Attributes

It is recommended that custom attribute elements be provided for test case maintenance purposes. There can be any number of custom attributes. Each attribute has a name that is a valid identifier and a value that can be any string.

Specifying custom test case attributes should be optional.

7.5.4 High-Level Functions

The high-level function element of the header declares high-level function APIs required for the test case to execute. Specific high-level function implementations can be contributed by different test technologies. As long as these implementations realize contracts declared in the test case header, the test case can execute. It is recommended that such declaration contain a list of URLs, each of which specifying an API for a library of high-level functions. These externally defined APIs are also used as constraints on high-level function calls in the execution flow section of the test case. Invalid high-level function API calls should result in compile-time errors.

Specifying high-level function APIs should be optional.

7.5.5 Test Case Data

Similarly to high-level functions, it is recommended that the test data element be a list of URLs. Each URL specifies a schema for test data required by the test case to execute. Specific test data instances are provided at runtime but the schema is known at design time to ensure validity of test data lookup.

The order in which the schemas appear in the list is insignificant. The order in which the symbols are looked up at runtime is determined by the design of the data instance rather than the order in which the schemas are specified in the test data element of the test case header. Test data schemas are only used as constraints on non-resource-specific data lookups at design time.

Specifying test case data schemas should be optional.

7.6 Resource Declaration

The resource declaration section specifies the environment meta-model against which the test case is written and the abstract resources the test case requires for its execution. The environment meta-model defines the available abstract resource namespace and the API for each resource abstraction in the namespace. The environment meta-models are described in more detail in clause 5.2 of the present document.

Resource declaration is a part of the test case script. It is recommended that it contains an environment meta-model URL, indicating that all required resources will come from a concrete environment that can be modelled by an instance of the specified meta-model.

The rest of the resource declaration section should be a list of resource allocation statements indicating the abstract resource name and the API the resource is expected to realize. A constraint on resource allocation statements is that a resource abstraction with the specified name is defined in the specified meta-model and that it realizes the specified API. Violations of this constraint should result in compile-time errors.

The relationship between test cases and their environments is many-to-many, which means that multiple test cases with the same resource requirements can execute on the same test environment and that the same test case can execute on multiple environments. In addition, concrete environments can, and in most cases should, be allocated dynamically. Statically defined environment meta-models allow the test case designer to write and validate (at design/compile time) test cases that use a yet unknown test environment.

7.7 Execution Flow

7.7.0 Introduction

The execution flow is a sequence of executable statements that manipulate abstract resources defined in the resource declaration section. As illustrated in the diagram in Figure 3, these statements call resource APIs and invoke high-level functions from high-level function libraries, APIs for which are specified in the test case header. Any API call may succeed or fail. If an API call fails, the test case fails immediately. If the required behaviour for the passing scenario is for the API call to fail, the language syntax should have a clause to indicate that if a particular API call succeeds, the test case is supposed to fail.

Control-flow statements are not recommended in the execution flow as they would compromise the notion of the test validity. Multiple possible paths through the test execution flow would result in uncertainty as to which specific path was executed at run time and hence which specific scenario was tested. This would render the test result essentially meaningless.

7.7.1 Getters for Resource-Specific Data

As mentioned in the clause 5.4, it is recommended that resource-specific data be retrieved by calling a getter defined in the resource API. The resource API is a contract, and as such, it makes no assumptions about the resource implementation or its properties. The getter is a method that has a contractual obligation to return a value with a particular meaning documented in the API. How the actual resource maintains, obtains, or computes this value is up to the resource implementation.

7.7.2 Symbol Lookup

Non-resource-specific test data come from the test case context and is looked up dynamically. It is recommended that symbols be fully qualified names to avoid ambiguity.

Similarly to the case of resource-specific data getters, the contract for symbol lookup does not imply any specific implementation of the test data or any specific data lookup mechanism. It can be implemented by the TEP or as a separate service. It can be represented as a search tree or stored in the object database. To the test case designer, this only means that the required datum value will be returned if it exists in the test data instance of the current test case context.

As described in clause 7.5.5, test data declaration in the test header contains a list of data schemas. This suggests that the data instance provided to the test case can potentially have multiple separate data sets, each corresponding to its respective schema. These datasets may have conflicts.

While it is possible to create different qualified names in different schemas to avoid ambiguity, it is not possible to enforce, especially if these data sets come from different organizations. Additionally, an argument can be made that using more generic names makes the test case more reusable.

If exactly the same qualified name exists in more than one data set, the first one found will be the one whose value is returned to the test case. The order in which the data hierarchy is searched depends on the specific data instance provided to the test case. Modelling of such data instances should include a mechanism to customize the data set with additional data sets that get searched first. This should be a general customization mechanism that can progressively specialize test data by providing a new set of values for an existing subset of qualified names and placing them first in the search order.

It is also recommended to have the ability to selectively specify which dataset to search first. An example of the use case that may need such functionality is running the same test with multiple sets of protocol defaults, specified by a single selector rather than replacing the whole dataset.

7.7.3 High-level Function Invocation

High-level function invocation is very similar to calling resource API methods. Similar to the resource API definition coming from the environment meta-model, the high level function API definitions come from URLs specified in the high-level function element of the test case header.

The difference from resource API calls is that test case resources are objects that have a state, while high-level functions are not called on a particular object and are therefore stateless. This means that any implementation of the high-level function API is a stateless component. This, of course, does not mean that the function cannot depend on, or modify, the state of any object passed to it as an argument.

Annex A: JADL Example

The following is an example test scenario represented in Joint Agile Delivery Language (JADL) that illustrates the concepts described in the present document. JADL is a DSL developed for the Test domain as described in clause 4, following the reuse guidelines in clause 5, and leveraging models in clause 6. It also closely follows the Test DSL design recommendations in clause 7. JADL can be found on GitHub: https://github.com/usrdcc/jadl.

In this example the system under test is a VNFM implementation, the required test resources are an NFVO component, an EM component, and a VIM component that can be either instrumented implementations assumed to work correctly or automation framework stubs. The test scenario is validation of the VNF instantiation procedure from NFVO ETSI GS NFV-MAN 001 [i.2], clause B.3.2.2. Since the focus of this example is on the Test domain and the Test DSL rather than specifics of the VNF instantiation, it is kept at a high level with many low-level details of the VNF instantiation procedure omitted.

A sequence diagram of the VNF instantiation from NFVO flow copied from [i.2], Figure B.11 is shown in Figure A.1.

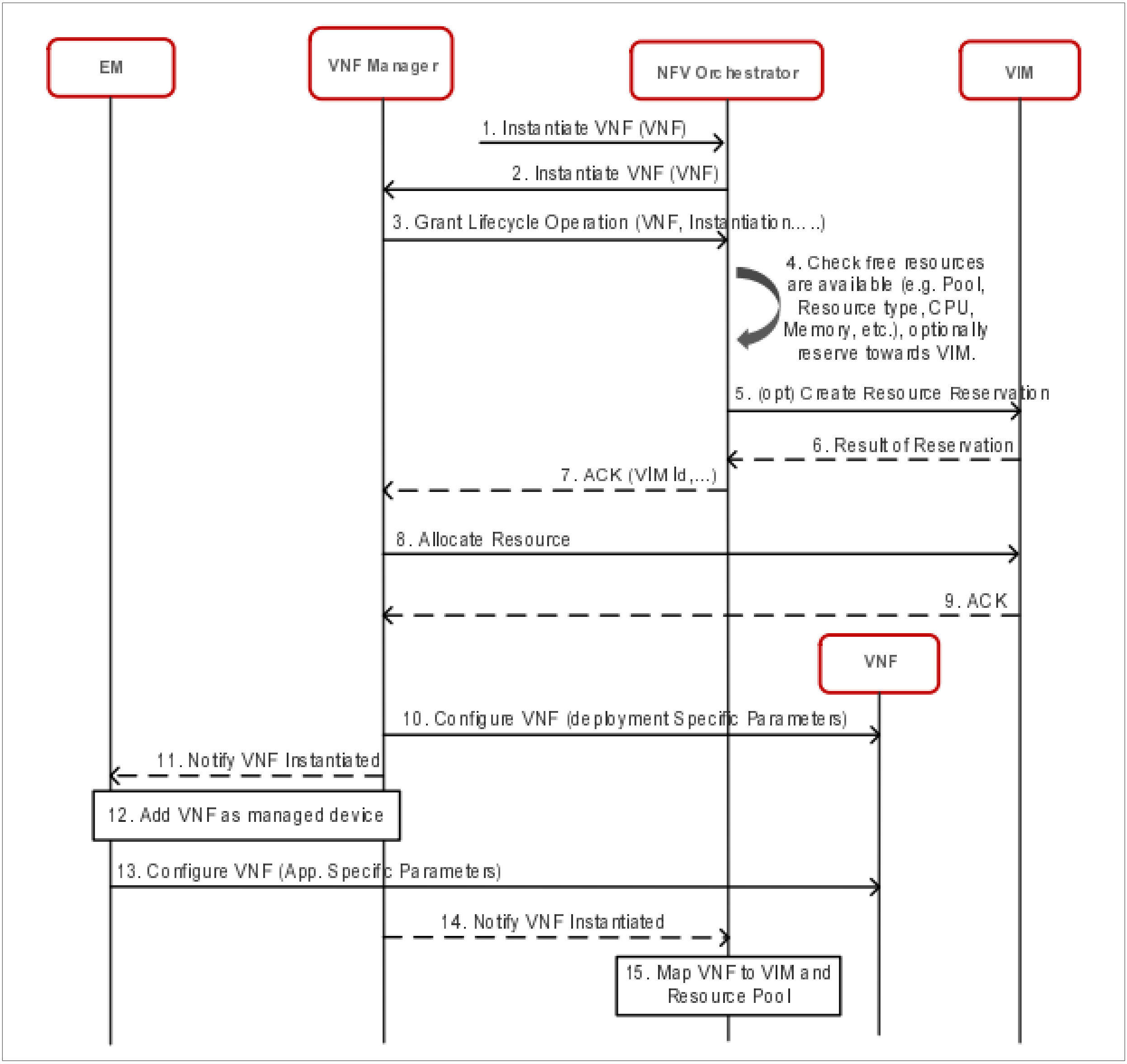
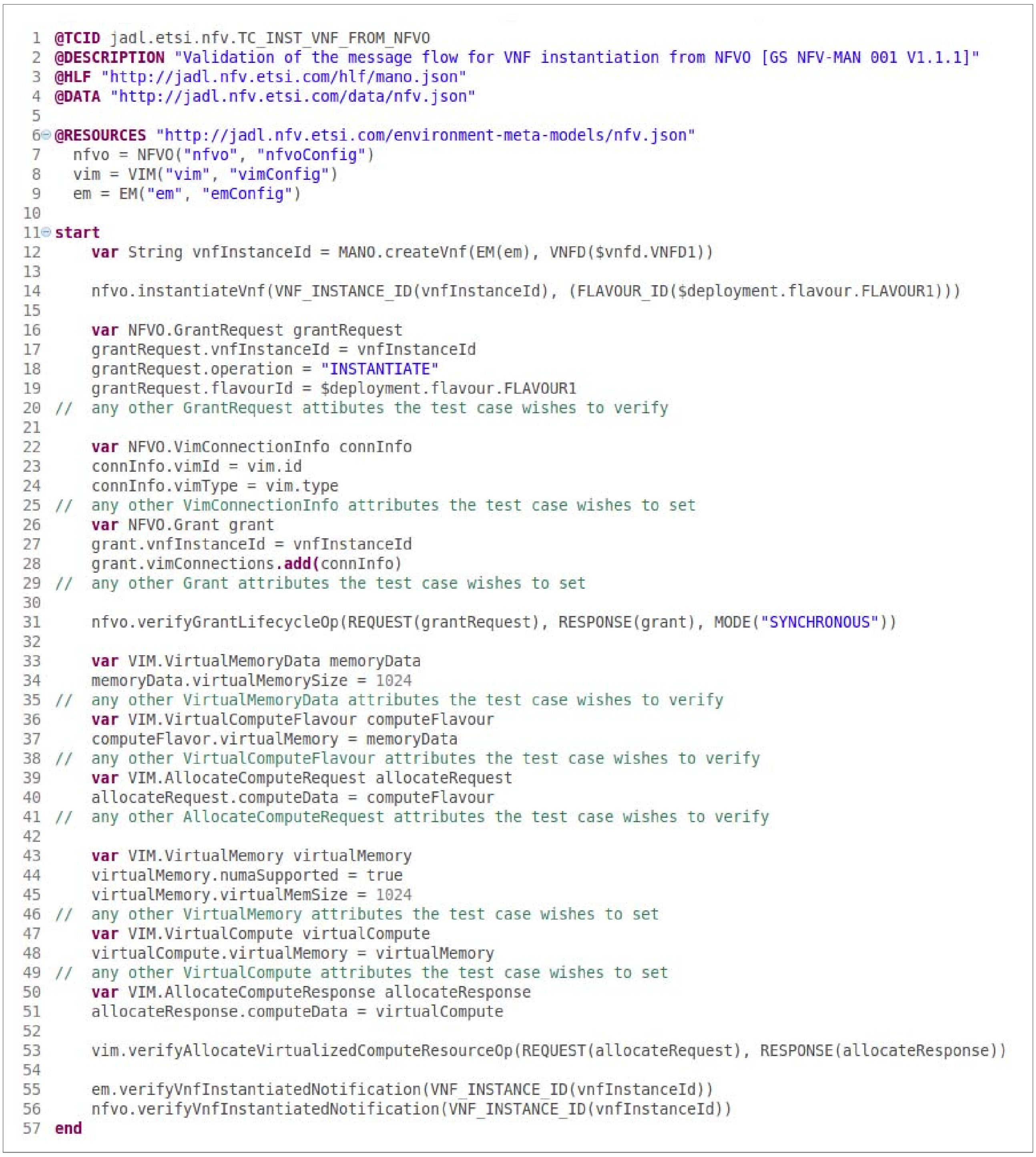

EM
VNF Manager
NFV Orchestrator
VIM
1. Instantiate VNF (VNF)
2. Instantiate VNF (VNF)
3. Grant Lifecycle Operation (VNF, Instantiation.....)
4. Check free resources are available (e.g. Pool, Resource type, CPU, Memory, etc.), optionally reserve towards VIM.
5. (opt) Create Resource Reservation
6. Result of Reservation
7. ACK (VIM Id,...)
8. Allocate Resource
9. ACK
VNF
10. Configure VNF (deployment Specific Parameters)
11. Notify VNF Instantiated
12. Add VNF as managed device
13. Configure VNF (App. Specific Parameters)
14. Notify VNF Instantiated
15. Map VNF to VIM and Resource Pool

Figure A.1: VNF instantiation from NFVO flow

In this diagram the VNF Manager timeline is the SUT for the test case and the EM, NFV Orchestrator, and VIM timelines are test resources.

An example JADL test case verifying the corresponding VNFM functionality is shown in Figure A.2.



```
 1 @TCID jadl.etsi.nfv.TC_INST_VNF_FROM_NFVO
 2 @DESCRIPTION "Validation of the message flow for VNF instantiation from NFVO [GS NFV-MAN 001 V1.1.1]"
 3 @HLF "http://jadl.nfv.etsi.com/hlf/mano.json"
 4 @DATA "http://jadl.nfv.etsi.com/data/nfv.json"
 5
 6 @RESOURCES "http://jadl.nfv.etsi.com/environment-meta-models/nfv.json"
 7   nfvo = NFVO("nfvo", "nfvoConfig")
 8   vim = VIM("vim", "vimConfig")
 9   em = EM("em", "emConfig")
10
11 start
12     var String vnfInstanceId = MANO.createVnf(EM(em), VNFD($vnfd.VNFD1))
13
14     nfvo.instantiateVnf(VNF_INSTANCE_ID(vnfInstanceId), (FLAVOUR_ID($deployment.flavour.FLAVOUR1)))
15
16     var NFVO.GrantRequest grantRequest
17     grantRequest.vnfInstanceId = vnfInstanceId
18     grantRequest.operation = "INSTANTIATE"
19     grantRequest.flavourId = $deployment.flavour.FLAVOUR1
20 //  any other GrantRequest attibutes the test case wishes to verify
21
22     var NFVO.VimConnectionInfo connInfo
23     connInfo.vimId = vim.id
24     connInfo.vimType = vim.type
25 //  any other VimConnectionInfo attributes the test case wishes to set
26     var NFVO.Grant grant
27     grant.vnfInstanceId = vnfInstanceId
28     grant.vimConnections.add(connInfo)
29 //  any other Grant attributes the test case wishes to set
30
31     nfvo.verifyGrantLifecycleOp(REQUEST(grantRequest), RESPONSE(grant), MODE("SYNCHRONOUS"))
32
33     var VIM.VirtualMemoryData memoryData
34     memoryData.virtualMemorySize = 1024
35 //  any other VirtualMemoryData attributes the test case wishes to verify
36     var VIM.VirtualComputeFlavour computeFlavour
37     computeFlavor.virtualMemory = memoryData
38 //  any other VirtualComputeFlavour attributes the test case wishes to verify
39     var VIM.AllocateComputeRequest allocateRequest
40     allocateRequest.computeData = computeFlavour
41 //  any other AllocateComputeRequest attributes the test case wishes to verify
42
43     var VIM.VirtualMemory virtualMemory
44     virtualMemory.numaSupported = true
45     virtualMemory.virtualMemSize = 1024
46 //  any other VirtualMemory attributes the test case wishes to set
47     var VIM.VirtualCompute virtualCompute
48     virtualCompute.virtualMemory = virtualMemory
49 //  any other VirtualCompute attributes the test case wishes to set
50     var VIM.AllocateComputeResponse allocateResponse
51     allocateResponse.computeData = virtualCompute
52
53     vim.verifyAllocateVirtualizedComputeResourceOp(REQUEST(allocateRequest), RESPONSE(allocateResponse))
54
55     em.verifyVnfInstantiatedNotification(VNF_INSTANCE_ID(vnfInstanceId))
56     nfvo.verifyVnfInstantiatedNotification(VNF_INSTANCE_ID(vnfInstanceId))
57 end
```

Figure A.2: JADL Test case example

The following is a line-by-line description of this test case example.

Lines 1 and 2 of the test case header specify the fully qualified test case ID and the test case description.

Line 3 points to high-level function API specification for MANO high level functions. This specification places dynamically loaded constraints on high-level function calls in the execution flow section of the test case that are checked at compile time. This corresponds to clause 7.5.4 and implements the reuse guidelines in clause 5.4.

Line 4 points to the Test Data schema for test data required by the test case to execute. The data schema places dynamically loaded constraints on valid data lookups in the execution flow section that are also checked at compile time. This corresponds to clause 7.5.5 and implements the reuse guidelines in clause 5.5.

Lines 6 through 9 are the Resource Declaration section. Line 6 points to the environment meta-model definition, against which the test case is written. Lines 7, 8 and 9 request test resources to be allocated to the test case. Line 7 allocates an NFVO resource by specifying the abstract resource name `"nfvo"` described in the environment meta-model and providing a dynamic configuration identifier `"nfvoConfig"` for any kind of dynamic configuration that the test resource may (optionally) require after it is allocated to the test case. The allocated resource type `NFVO` is a resource API type defined in the environment meta-model for NFVO test resources. The allocated test resource is assigned to the variable `nfvo`.

The environment meta-model places several dynamically loaded constraints on the test case code. First, resources with the name `"nfvo"` exist; second, it is necessary that they implement the `NFVO` API; and third, execution flow statements are only allowed to call methods on variable `nfvo` that are defined in the `NFVO` API. If any of these constraints are violated a compile-time error will be generated. Lines 8 and 9 allocate a VIM and an EM resources in a similar fashion. This corresponds to clause 7.6 and implements the reuse guidelines in clauses 5.2 and 5.3. The Resource Declaration section aggregating abstract resources that implement APIs called by the Execution flow is a part of the Test Case Model described in clause 6.2 and shown on Figure 3.

When the code in the Resource Declaration section is executed, abstract resources (as described in clause 4.2) are instantiated and a request is sent to the resource management system to map them to a set of concrete resources. The environment meta-model specified in line 6 is used for this mapping as shown in Figure 4. Dynamic configuration identifiers such as `"nfvoConfig"`, `"vimConfig"`, and `"emConfig"` in lines 7-9 correspond to the provisioning data applied to the allocated concrete resources which in this instance are Test Tools elements.

Please note that allocation of the NFVO, the VIM, and the EM resources, as separate unrelated entities, is vastly simplified and would be insufficient in practice. Since test resource management and abstract environment modelling is outside the scope of the present document, for the purpose of this example, it is assumed that required relationships among the allocated resources are satisfied. One such relationship can be connectivity for example.

The test execution flow consists of one segment (described in clause 4.6 and shown on the diagram in Figure 3) and is delimited with the keywords `start` and `end`. Before the VNF instantiation flow can be executed, it is necessary for the VNF creation flow to be executed. This is a necessary step and one that creates the VNF instance ID used throughout the test case, but the specifics of the VNF creation flow are of no particular interest to the test case since its main focus is on the VNF instantiation procedure. Therefore it is accomplished in one single step in line 12 by calling a high-level function `createVNF()` defined in the MANO HLF specification referenced in line 3. This corresponds to the "Calls" relationship between the Execution Flow and the High-Level Functions in Figure 3. Parameters passed to this call are the EM test resource and the VNFD to be used by the VNF creation procedure. The parameters are label-value pairs for order-independence and greater flexibility with a potentially large number of optional parameters.

In this example the VNFD is assumed to be defined in the test data. It is passed to the `createVNF()` function call by specifying the label `VNFD` and the value dynamically looked up in the test data by its qualified name `vnfd.VNFD1`. This name complies with the Test Data schema definition in line 4, otherwise a compile-time error will be generated. The function call returns the VNF instance ID of the newly created VNF to be used in the rest of the test case.

Line 14 calls `instantiateVnf()` method on the `nfvo` resource. This corresponds to the "Calls" relationship between the Execution Flow and the Abstract Resource APIs in Figure 3. In this case the deployment flavour ID is passed to the method call by specifying the label `FLAVOUR_ID` and the value that is also dynamically looked up in the test data. The qualified name looked up in the test data is `deployment.flavour.FLAVOUR1`.

Dynamic test data lookup in lines 12 and 14 corresponds to the "Dynamic lookup" relationship between the Execution Flow and the Test Data in Figure 3.

After receiving the Instantiate VNF request the VNFM is supposed to send a Grant Lifecycle Operation request to NFVO. Line 31 verifies that by calling the method `verifyGrantLifecycleOp()` on the `nfvo` test resource. The parameters passed to this call are built in lines 16 through 29. The data models used for these parameters are defined in ETSI GS NFV-SOL 003 [i.3], clause 9.5. Since the VNF Lifecycle Granting interface allows both synchronous and asynchronous implementations of the Grant Lifecycle Operation flow [i.3], clause 9.3, this call also instructs the NFVO test resource to verify the synchronous implementation by specifying the `MODE` parameter.

Lines 16 to 20 create an instance of the Grant Request and set its attributes to the values that are to be verified. Line 22 creates an instance of VIM Connection Info object to be included in the Grant to be returned to the VNFM. In lines 23 and 24 its attributes `vimId` and `vimType` are set to the `id` and `type` values obtained from the `vim` test resource respectively by accessing its resource-specific data. Lines 26 to 29 create an instance of the Grant to be returned and set its attributes.

In this example `GrantRequest`, `Grant`, and `VnfConnectionInfo` data elements are assumed to have been defined in the NFVO resource API definition, hence all three types are qualified to the `NFVO` type. Alternatively, they could be defined separately in a global VNF Lifecycle Operation Granting interface definition included in the NFVO resource API definition, in which case they would be qualified to the global definition. In either case the type information is loaded from the metadata referenced from within the environment meta-model in line 6 and can be validated at compile time.

After the INSTANTIATE lifecycle operation is granted by the NFVO the VNFM is supposed to allocate virtualised compute resources from the VIM. Line 53 verifies that by calling the method `verifyAllocateVirtualisedComputeResourceOp()` on the `vim` test resource. Similarly to lines 16 to 29, lines 33 through 51 build the parameters passed on this call and should be self-explanatory. The data models used for these parameters are defined based on ETSI GS NFV-IFA 006 [i.4], clause 7.3.1.2.

Finally, lines 55 and 56 verify notifications the VNFM is supposed to send to the EM and the NFVO upon the successful completion of the VNF insanitation by calling the method `verifyVnfInstantiatedNotification()` on the `em` and the `nfvo` test resources respectively. The VNF instance ID is passed on both calls for verification.

Please note that for the purpose of this example, the instantiated VNF is not a test resource and the verification of step 10 of the flow in Figure 1, configuration of the VNF with the deployment-specific parameters, is not performed. A more complete example would allocate a VNF test resource, configure it with parameters required by the test scenario, return it to the VNFM, and verify step 10 of the flow by calling appropriate methods on the VNF test resource.

Annex B: Authors & contributors

The following people have contributed to the present document:

Rapporteur:
Frank Massoudian, Huawei Technologies Co., Ltd.

Other contributors:
Edward Pershwitz, Huawei Technologies Co., Ltd.

Pierre Lynch, Keysight Technologies UK Ltd.

History

Document history		
V1.1.1	March 2019	Publication