Edward Pershwitz, PhD US RDCC

FUTUREWEI TECHNOLOGIES, INC. A Subsidiary of Huawei Technologies Co., Ltd

5700 Tennyson Parkway, Suite 600, Plano, TX 7502

JAD Test Case DSL Design

# Copyright Notification

No part of this document may be reproduced, distributed, transmitted, or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of Futurewei Technologies, Inc.

The content shall not be modified without the written authorization of Futurewei Technologies, Inc. The copyright and the foregoing restriction extend to reproduction in all media.

© Futurewei Technologies, Inc. 2018.

All rights reserved

# Introduction

JAD Test DSL is designed to express tasks in the domain of testing ICT. Concepts of the test domain are abstractions that are general enough to allow leveraging of different test technologies yet specific enough to reflect common test practices.

In the context of JAD multiple vendors have their own test technologies, interfaces into the system under test, and test languages (usually GPLs like Java, Ruby, Python, etc.) Likewise, in the context of a single company different Business Units (BUs) may also have their own test technologies. In order to create a common test language the test cases need to 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 artifacts 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 needs 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 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.

For the necessary background for JADL design refer to *ETSI GR NFV-TST011, Test Domain and Description Language Recommendations [1]*.

# JADL Components

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.

The task of writing a test case in a DSL is a task of building the semantic model of the test using the DSL syntax. This semantic model (also commonly referred to as “semantic graph”) is an object model that captures what the test case is supposed to do when it executes.

The test case domain model described in [1] has the following elements: test execution flow, test case resources, test data, high-level functions, and test segment model. The test case semantic model needs to define 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 modeler that will manipulate the conceptual model directly from the modeler UI (Figure 1).

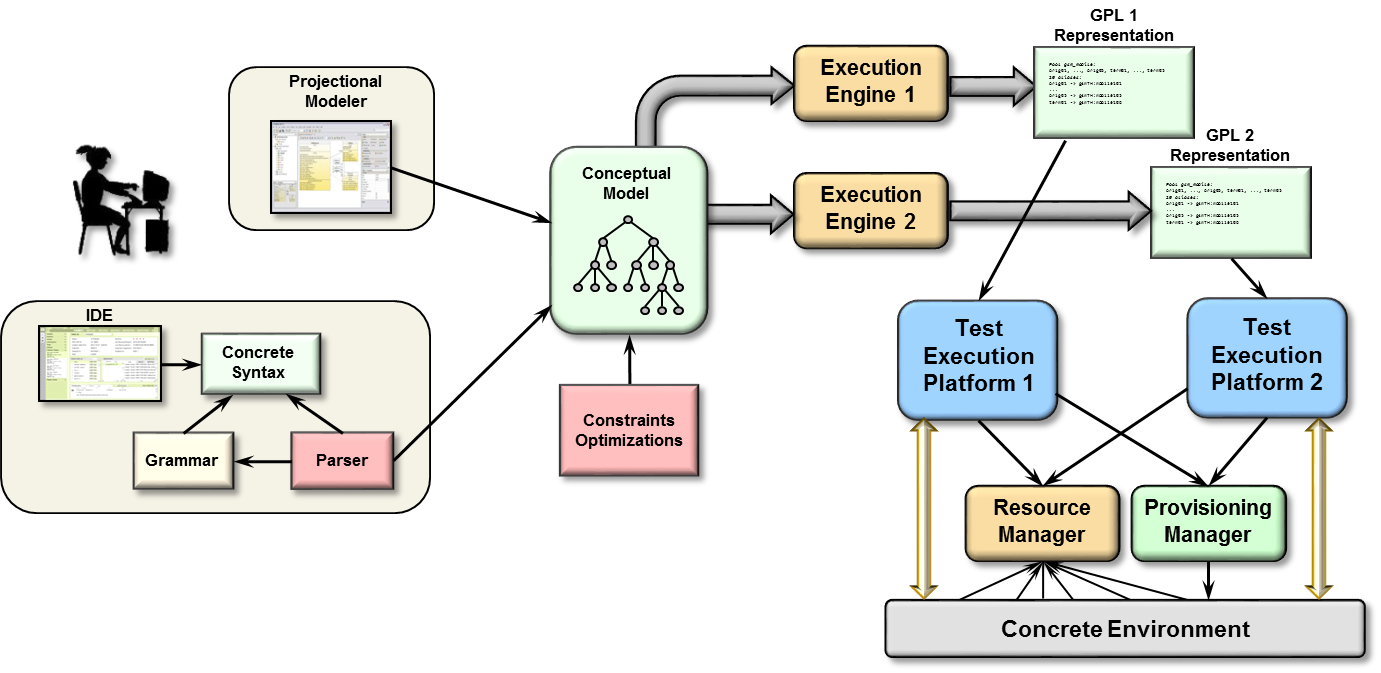


Figure 1. JADL Components

The same conceptual model can have multiple execution engines that generate GPL representations of the test for multiple target platforms. Figure 1 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 modeler implementation), the resulting execution semantics, and code generation of various GPL representations.

The remainder of this white paper will be focused on the text-based JAD Test DSL and the mechanisms it provides to build a conceptual model based on the standardized Test Case Domain meta-model ([1], Figure 6).

# Text-based JADL

JADL implementation is based on Xtext, the framework for building external DSLs based on Eclipse Modeling Framework (EMF). When a DSL grammar is created with Xtext it is used to generate an entire language infrastructure: the abstract syntax meta-model, the parser, the validator, and the IDE with content assist and code completion. This code is packaged as a set of OSGi bundles that can be installed into an existing Eclipse configuration or run as a standalone RCP application.

## Abstract Syntax meta-model

When a JADL Test is processed by the parser a semantic model of the test is generated. A different semantic model is created for each test, 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 abstract syntax meta-model is in turn an instance of Ecore – a model from which all EMF meta-models are instantiated. It defines the structure of the abstract syntax nodes and associations between them. Ecore thus serves as the meta-meta-model for all programs written in the Test DSL. It is thanks to Ecore that abstract syntax meta-models can be generically edited (a custom validator attached to a node, for example) or that semantic models can be generically accessed using a generated API (getting a specific attribute value from a resource declaration element instance, for example).

The relationships between Ecore, the abstract syntax meta-model and test case semantic models are illustrated in Figure 2.

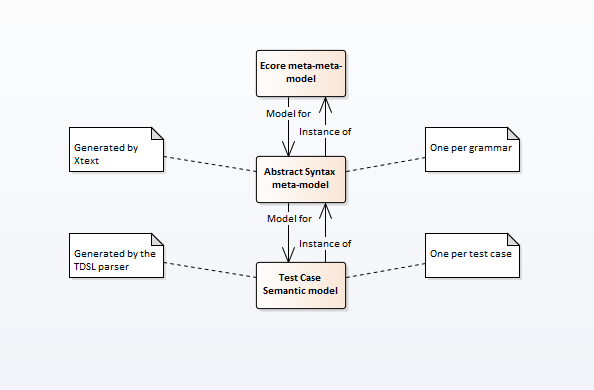


Figure 2. Meta-model Relationships

## Custom Validation

Syntax validation code is part of the language infrastructure generated by Xtext from the grammar. The generated validator will produce compile errors for syntactically incorrect programs. In addition, as mentioned above, custom validators can be attached to nodes of the abstract syntax meta-model. JADL takes advantage of this functionality to not only add statically defined constraints, but also validate the code against dynamically loaded artifacts. The following example will clarify this concept.

Figure 3 shows an example of the JADL resource declaration code. The URL following the **@RESOURCES** keyword points to a concrete environment resource meta-model (or environment meta-model for short) against which the test case is written. The relationship between JADL and concrete environment resource models will be described in more detail in the [Environment Decoupling](#_Environment_Decoupling) section of this white paper. For discussion of custom validation is suffices to say that the environment meta-model provides a namespace for abstract resources available from the test environment and their APIs.

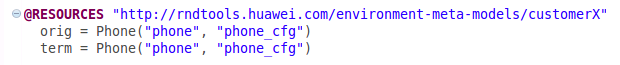


Figure 3. Resource Declaration

Resource declaration section starting with **@RESOURCES** keyword is a mandatory part of any JADL test case – the test case will not compile without it.

Once the environment meta-model is loaded and processed, the information of all available abstract resources and their APIs is stored in a special module of the test case’s sematic model and is made available to the custom validator attached to the resource declaration element of the abstract syntax meta-model. Thus, the validator can check at compile time that the abstract resource “phone” is present in the test environment and that it realizes the “Phone” API. If either of these constraints is violated it will generate descriptive compile errors as illustrated in Figure 4 and Figure 5.

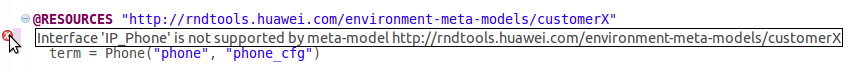
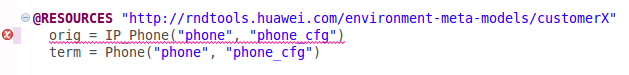


Figure 4. IP\_Phone is an unsupported API

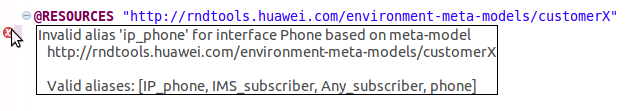
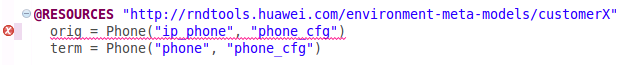


Figure 5. “ip\_phone” is an invalid resource alias

Likewise, the same custom validator can check for the validity of API calls on abstract resources and high-level function calls based on resource API and high-level function metadata dynamically loaded into their respective modules of the test case semantic model and made available to the validator.

It is important to note that even though the environment meta-model and the resource and high-level function metadata are loaded dynamically, they are statically defined artifacts. This is a particularly important design consideration for the environment meta-model and will be discussed in much more detail in the [Environment Decoupling](#_Environment_Decoupling) section.

## Custom code assist and code completion

Code assist and code completion of the JADL IDE follow the same pattern as validation. The IDE generated by Xtext will provide code assist and code completion proposals based on the language grammar. Additional custom proposals are generated from dynamically loaded artifacts. Figure 6 illustrates custom proposals for all resources that realize API “Phone” in the loaded environment meta-model.

Since the custom validator and the custom proposal provider code utilize the same semantic model modules they work consistently with each other: all proposals will always validate and all valid completions will always be proposed.

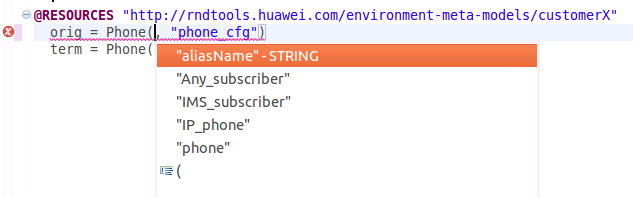


Figure 6. Code proposals showing all available resources

Valid resource API and high-level function calls are proposed in the same fashion based on loaded metadata.

## Execution Engines

An executable program needs to have an execution engine to be able to run on its target platform [2]. JADL is compiled into GPLs for which their own execution engines already exist. Figure 1 illustrates compiling JADL into two different GPLs. Adding support for additional GPLs is only a matter of writing an execution engine for each GPL.

It must be noted that this approach should not compromise test case standardization. Test execution platform implementations for different GPLs should be able to manipulate the same conceptual model, “understand” various test case elements and the test environment [1], and interact with the dynamic resource management system to allocate test resources. The test representation can be generated to match any new or existing TEP implementation but the test case model needs to stay the same to allow integration across multiple TEPs. This places specific requirements on existing TEPs wishing to provide JADL support.

The proof-of-concept Java TEP that has been developed in Futurewei US RDCC supports the standardized test case model natively and can serve as a reference implementation. JADL execution engine for Java is a part of the same proof-of-concept implementation.

It is also possible to design hybrid solutions where support for the standardized test case model is divided between the Java TEP and existing TEPs. For example, test resource allocation and interactions with the DRM is handled by the Java TEP and the actual test case execution is supported by the existing TEP. This idea is illustrated in Figure 7.

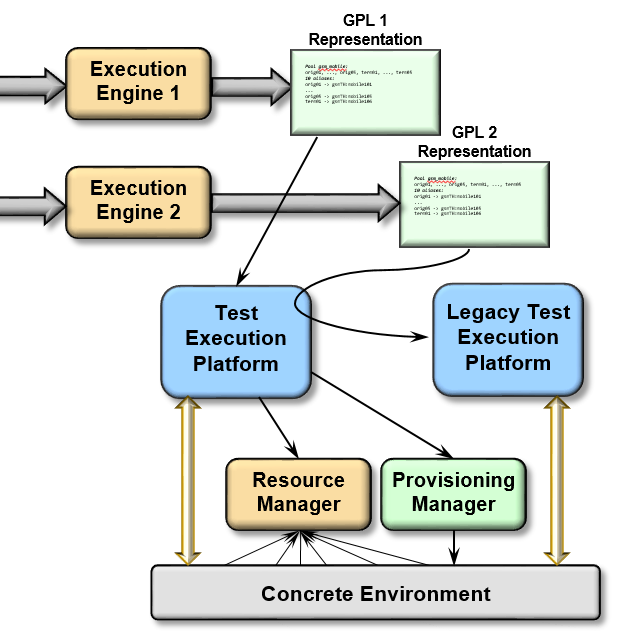


Figure 7. Hybrid solution for a legacy TEP

Specific design details of such hybrid systems are outside of the scope of this white paper.

# Environment Decoupling

Decoupling the test environment from test cases is achieved by strict separation of abstract and concrete resources and using the Dynamic Resource Manager (DRM) to map the abstract resource space to the concrete resource space. This section summarizes decoupling concerns as related to JADL design.

The relationship between JADL 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. Not only is this true for concrete environments, which is an expected consequence of decoupling, but what’s not as intuitive, it is also true for abstract environments. This will become clear from the further discussion in this section.

The question that needs to get answered is how a test case designer can write a test case that uses a yet unknown test environment. In order to answer this question the dynamic resource management tiers need to be briefly introduced.

A global-level tier DRM (we will call it tier-1) manages the global scope of 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 concrete environment resource model (environment model for short). The environment model is used by the DRM to find a concrete resource for an abstract resource request from the resource consumer.

The environment model is an instance of its own model that specifies resource abstractions in the environment model and their relationships. This model is defined statically and is a meta-model for a family of environment models, hence its name: concrete environment resource meta-model. These resource abstractions are visible to resource consumers using any specific environment instance modeled with the environment model. For example, an environment model may have 1000 or 2000 RNCs but if the environment meta-model defines a pool of RNCs called “RNC” then a resource consumer can request 10 members of the RNC pool without the knowledge about the concrete resource space modeled with the specific environment model.

When a test activity needs a concrete environment to run on (a wireless network with a specific configuration for example) an abstract environment gets designed to describe this concrete environment. This abstract environment is defined in terms of specific resource abstractions provided by the environment meta-model (the “RNC” pool for example). This abstract environment is a reusable artifact that multiple test activities can use to build concrete environments from multiple environment models as long as they are all instances of the same environment meta-model.

When a concrete environment is built for a test activity a new instance of the DRM is created to manage this concrete environment (we will call it tier-2) – the management is needed to parallelize test case execution sharing the same environment. This new instance of the DRM will build its own environment model to model the (now known) concrete resources allocated to the test activity by the tier-1 DRM. The environment models will be different for different concrete environments but they will also be instances of a statically defined environment meta-model that defines resource abstractions available to individual test cases. For example, if a test case needs to make a basic A-calls-B call and needs two phones for it then it can request two members of the Phone pool defined in the tier-2 environment meta-model.

This is the point where it becomes relevant to JADL design. As long as the test case is written against a particular tier-2 environment meta-model it does not matter which specific tier-2 environment model is used to model the concrete resource space. It even does not matter what abstract environment the test activity uses as long as it can be modeled with an instance of this environment meta-model – therefore a JADL test case can run as a part of multiple test activities that require different abstract environments, each of which can have any number of concrete environments allocated to it.

The following example will pull it all together. Test activity X runs on a wireless network in New York. Test activity Y runs on an IMS network in Chicago. Each needs its own (reusable across multiple test activities in New York or multiple test activities in Chicago) abstract environment. New York has a 1000 wireless phones and Chicago has 500 IP phones. Each abstract environment is designed against the same tier-1 environment meta-model. When both activities request their own abstract environments from the tier-1 DRM it uses its environment model that models the global scope of resources to map these two abstract environments to a concrete set of resources. For each activity an instance of its own tier-2 DRM gets created – one to manage resources in New York and one to manage resources in Chicago. Our basic-call JADL test case needs two phones to run and is written against a tier-2 environment meta-model that has a pool “Phone”. When the New York DRM instantiates the environment meta-model all 1000 wireless phones become members of the Phone pool in its environment model. When the Chicago DRM instantiates the same environment meta-model all 500 of its IP phones become members of the Phone pool in its environment model. The basic call JADL test case can now run on any two of 1000 wireless phones in New York or any two of 500 IP phones in Chicago.

A URL for the environment meta-model against which a JADL test case is written is specified after **@RESOURCES** keyword in the resource declaration section as illustrated in Figure 3. The language can then provide validation and code assist/completion based on this environment meta-model not only decoupled from resources in the specific concrete environment, but also decoupled of the specific abstract environment required by the test activity.

Needless to say, none of this complexity is exposed to the end user of JADL. 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 “phone”) they need to use in the test case. The language will propose available resource abstractions and the APIs they support as discussed in [Custom Validation](#_Custom_Validation) and [Custom code assist and code completion](#_Custom_code_assist) sections.

As a side note, the ability to provide this type of support to the end user is contingent upon having statically-defined environment meta-models. If environments are defined in a completely dynamic fashion with no prior domain knowledge captured in the environment meta-model no such support from the language is possible. The decision of how much domain knowledge to capture in statically-defined meta-models and how much to model completely dynamically also lies with solution and automation architects and may vary from organization to organization depending on specific test domain requirements.

# Resource API Decoupling

Test resources are contributed from multiple sources. Test cases written in JADL need to be able to execute on multiple TEPs that may have their own implementation of test resources. It is even more relevant in the context of JAD because different test technologies may come from different vendors. This is not a problem for the DRM because it treats all resources generically, but it becomes a problem for JADL because it needs to be able to manipulate different contributed and not known beforehand implementations of the abstract resources, provide compile-time error checking, and work interchangeably with multiple test technologies.

The only way this can happen is if the resource API is decoupled from any resource implementation and is defined outside of any specific test technology. Continuing with the New York/Chicago example above, a wireless phone resource implementation may be supported by code written in TTCN and/or C++, and an IP phone resource implementation may be supported by code written in TCL and come from a different organization. There may be other yet unknown phone implementations coming from yet another organization running on its own test technology. Our basic-call JADL test case has no knowledge of that but as long as the Phone API is defined outside of any specific resource contribution the test case can be written against that API, compile-time error checking can be performed, and code assist/code completion can be provided to the user.

Figure 8 illustrates a sequence of resource API calls to bring up a basic call. No assumption is made about the type of phones used by the underlying implementation or what concrete resources may be allocated to the test case by the DRM.

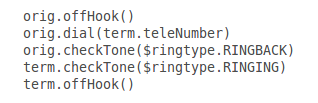


Figure 8. Resource API call example

When GPL code is generated for a particular contributed test technology the JADL execution engine for that GPL needs to perform the required mapping from resource API calls in JADL program to calls on specific resource implementations defined within individual test technologies.

*It needs to be noted that the term GPL is used loosely here. Both TTCN and TCL are specialized for particular applications and neither is a general-purpose language strictly speaking. For the purpose of this discussion it does not matter. What matters is that JADL compiles into a language for which an execution engine is assumed to exist.*

As mentioned in the previous section, each JADL test case is written against a particular environment meta-model that defines resource abstractions available to the test case. These resource abstractions are only meaningful if an API is defined for each of them that tells the test case designer how the resource can be used. Thus a reference to a platform-independent resource API definition needs to be specified for each resource abstraction in the environment meta-model.

Test Environment Management Service API, including Environment Meta-Model management, Environment Instance (abstract environment definition) management, and Resource API management, has been contributed to TM Forum as part of Phase 3 JAD Catalyst and is available from Swaggerhub. This contribution is a part of industry-wide test standardization effort for the digital marketplace.

# High-level Function Decoupling

JADL High-level functions (HLF) can come from two different sources. HLFs can be written in JADL and then compiled into a GPL just like any other JADL code, or they may be already implemented by the existing test technology provided as a contribution. JADL HLFs provide a high degree of reuse and a single point of truth (written or fixed once – used everywhere). However, most existing test technologies will have some form of their own HLFs already implemented that need to be leveraged.

Just like with resource contributions this necessitates having an externally defined contract for using these HLFs with which all implementations must comply. And just like with resource APIs it is the responsibility of individual execution engines to compile HLF API calls into calls on specific HLF implementations. In this way JADL can validate HLF calls and provide code assist/code completion to the user.



Figure 9. High-level function API call example

Figure 9 illustrates a single Telephony API call that performs the same basic call function as in the previous example. The HLF can be coded in JADL as shown in Figure 8 and then compiled into a GPL by the corresponding execution engine. Alternatively, the underlying test technology can provide its own implementation for the “makeCall” functionality.

# Test Data

JADL provides a mechanism for separation of the test case execution flow from the data used to parameterize the test. Such separation encourages reuse and is a part of test case standardization.

Test data can be resource-specific and non-resource-specific. Resource-specific data typically comes from the concrete resource allocated to the test case and is not known in advance to the test case designer. An example could be a dialed number for the terminating phone in our basic call example. Resource-specific data is obtained by simply calling a getter for the needed datum as provided by the resource API. term.teleNumber in Figure 8 and Figure 9 is an example of calling such a getter. This will be further discussed in the [JADL Syntax](#_TDSL_Syntax) section.

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 defaults down to individual test activities. The data is dynamically looked up by the test using fully qualified symbols as keys. Each fully qualified name represents a path from the root of the data hierarchy to the node holding the value. These nodes may or may not be leaf nodes. If a branch node is specified then the value is a meaningful representation of the sub-tree below the node. $ringtype.RINGBACK and $ringtype.RINGING in Figure 8 are examples of dynamic lookup of fully qualified symbols ringtype.RINGBACK and ringtype.RINGING. This will also be further discussed in the [JADL Syntax](#_TDSL_Syntax) section.

The symbol lookup functionality is provided by the test execution platform 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. It is the responsibility of the execution engine to compile JADL symbol lookup into code in the corresponding GPL that invokes any TEP-specific data access functionality.

In order to use test case data the test case designer needs 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 already solved by making getters for any data elements the resource exposes to the user a part of the resource API specified in the environment meta-model. The getters are supported in the same fashion as any other resource API as described in the [Resource API decoupling](#_Resource_API_Decoupling) section.

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. To make things worse the namespace can be very large and fully qualified symbol names could also be long and prone to spelling errors. An example of such large data sets with relatively long paths from the root to the leaves could be a set of protocol defaults.

The problem can be rectified if a schema for test data is made available to the test case at design time. Unfortunately it’s not always possible because the test data is by nature a dynamic structure that can be applied to a large number of test cases and can be changed, augmented, or customized at any time to accommodate new requirements.

JADL takes the “best effort” approach to this problem. A schema can be specified by the test case, loaded to a module of the semantic model and made available to the validator and code assist. Symbol names will be validated against this schema, auto-proposed, and auto-completed, however if the symbol typed by the user does not validate only a compile warning, rather than a compile error will be generated. In this way as long as test data instance contains a symbol that the test case is looking up it does not have to comply with the schema. Compile warning will keep reminding the user that the schema needs to be updated to accommodate the new requirement.

If more than one schema is specified validation will be attempted against all available schemas until the symbol either validates or a warning is generated. Code assist/code completion will work in a similar fashion: code proposals will be generated for the symbol name typed so far from all available schemas.

In many cases the schemas can stay fairly stable. In our example of protocol defaults the schema does not need to change unless the protocol changes. The actual defaults are instance data and can change anytime.

# JADL Syntax

A test case in JADL is roughly separated into three sections: the test case header, the resource declaration, and the execution flow. The order of these sections is fixed to give test cases a uniform structure. JADL in general takes the approach of enforcing best practices through syntax wherever possible to improve readability, reduce the maintenance effort, and simplify test case template generation.

JADL syntax closely follows the standardized test case model described in [1]. The diagram shown in Figure 10 is included as a reference to help visualize the relationship between syntactic elements described below and the test case elements of the standardized test case model shown in the diagram.

The execution segments and the segment ordering data are temporarily omitted from this white paper and will be included in future revisions when JADL support for modeling test case segments is added.

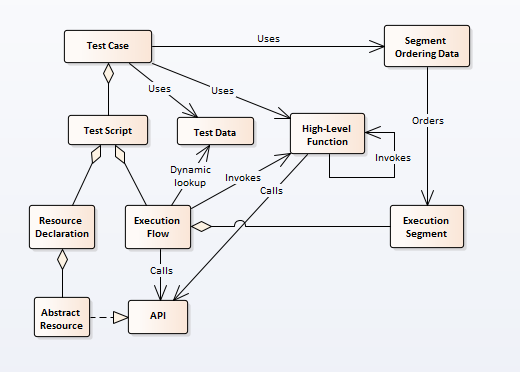


Figure 10. Test Case Elements

## Test Case Header

The header describes the test case and its relationship to other artifacts the test case interacts with. The elements of the header appear in the test case in the same order they are described below. These elements at this point are a minimalist set which will be expanded in the future as the language matures.

The test case header reflects the “Uses” relationships shown in Figure 10. The test case data and high-level functions used by the test case are both specified in the test case header as described below.

Figure 11 illustrates a simple example of the JADL header.

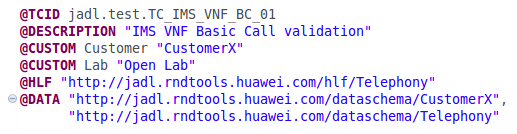


Figure 11. JADL header example

### Test Case Identifier

A test case ID is a string uniquely identifying the test case. Uniqueness cannot be enforced by the language and is a concern for the test case management, which is outside of the JADL itself. However, since individual users are only in control of a specific namespace within their organization, separation of these namespaces are essential to avoid conflicts. Such separation is enforced by JADL with the requirement that the test case ID be a fully qualified name. In the context of JAD this is even more important to avoid name conflicts between different vendors.

The test case id header element consists of **@TCID** keyword followed by a dot-separated fully qualified name, as illustrated in Figure 11. Each segment of the name needs to be a valid identifier consisting of upper- and lower-case alpha characters, numbers, and underscores, and starting with a character.

Specifying the test case id is mandatory.

### Test Case Description

Test case description element is a free-flowing natural language description of the test case following the **@DESCRIPTION** keyword as illustrated in Figure 11. A meaningful description simplifies maintenance of the test cases and is therefore enforced by JADL.

Specifying the test case description is mandatory.

### Custom Test Case Attributes

Custom attribute elements are provided for test case maintenance purposes. There can be any number of custom attributes. Each attribute has a name that needs to be a valid identifier and a value that can be any string. Each attribute occupies one test case header line and starts with the keyword **@CUSTOM** as illustrated in Figure 11.

Specifying custom test case attributes is optional.

### High-Level Functions

High-level functions is a comma-separated list of URLs following the **@HLF** keyword. Each URL specifies an API for a library of high-level functions. JADL uses these externally defined APIs to validate high-level function calls in the execution section of the test case. Please refer to [High-level Function Decoupling](#_High-level_Function_Decoupling) section for more detail.

Figure 11 illustrates the use of the high-level function header element. It only specified one library, *Telephony* that has functions for general telephony, e. g. making phone calls, verifying the voice path, etc.

The validator will check the validity of the URL syntax by converting it to a Java URL object. If the conversion fails a compile error indicating that the URL is malformed will be generated. Specifying high-level function APIs is optional.

### Test Case Data

Similarly to high-level functions, the test data element is a comma-separated list of URLs following the **@DATA** keyword. Each URL specifies a schema for test case data each of which will be used for non-resource-specific data lookup validation and code assist. Please refer to Test Data section for more detail.

Figure 11 illustrates the use of the test case data element. It specifies two data schemas, *Telephony* and *CustomerX*, each providing their own hierarchies of symbol names.

The order in which the schemas appear in the list is insignificant. The validation will be attempted against both schemas and all code proposals will be sorted alphabetically for both schemas together. 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. The schemas are only used at design time for validation and code assist/code completion. The look-up order will be further discussed in Symbol Lookup section of this white paper.

Specifying test case data schemas is optional.

## 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. Environment meta-models and their role in the JADL context are described in detail in the Environment Decoupling section of this white paper.

Resource declaration relationship to other standardized test case elements is illustrated in Figure 10. It is a part of the test case script and it aggregates abstract resources each realizing its own API.

The syntax of the resource declaration section is illustrated in Figure 12. It starts with the **@RESOURCES** keyword followed by the environment meta-model URL, indicating that all following resources will come from a concrete environment that can be modeled by the specified environment meta-model.

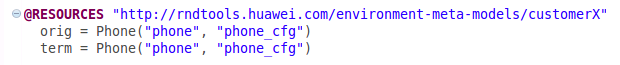


Figure 12. Resource Declaration Example

The rest of the resource declaration section is a list of resource allocation statements. The syntax of the resource allocation statement is straightforward. It allocates an instance of an abstract resource and assigns it to a variable (orig or term in this example).

The right hand side of the assignment looks like a function call where the name of the function is the name of the API the resource has to realize (Phone in this example), and the arguments are the resource abstraction name as defined in the environment meta-model (**“**phone**”** in this example) and an optional dynamic configuration name that needs to be applied to the resource at allocation time (**“**phone\_cfg**”** in this example). The same resource may have multiple dynamic configurations it can be provisioned with. Resource provisioning is outside the scope of this white paper.

JADL will validate that a resource abstraction with the name **“**phone**”** that realizes the API Phone is defined in the specified environment meta-model (http://drm.rndtools.huawei.com/environment-meta-models/customerX in our example). Validation is described in more detail in the Custom Validation section of this white paper.

Once an abstract resource instance is assigned to the variable its API type becomes the type of the variable. In our example any method defined in the Phone API is callable on the orig and term objects as will be illustrated in the Execution Flow section.

Please note that both allocation statements in this example allocate the same resource name with the same API for both the originator and the terminator. This is possible because **“**phone**”** is the name of a resource pool, of which the concrete environment may contain multiple instances. It is not possible for JADL to validate that multiple instances of the specified resource are available from the concrete environment modeled by the environment meta-model, as this information is only available at run time. If only one instance is available resource reservation for such test case will fail immediately.

## Execution Flow

The execution flow is a sequence of executable statements that manipulate abstract resources defined in the resource declaration section. As shown in the diagram in Figure 10, these statements call resource APIs and invoke high-level functions from high-level function libraries specified in the test case header. The execution flow functionality will be evolved as JADL matures.

### Basic Call Example

An execution flow example for a basic call is shown in Figure 13. The start of the execution flow is marked by the keyword **start** and the end by the keyword **end**. Between these two keywords there is a sequence of execution flow statements. All statements in this example make API calls on two objects, orig and term. This corresponds to the resource declaration example in Figure 12. Each API call (e.g. offHook, onHook, dial, etc.) is a part of the Phone API provided by the customerX environment meta-model.

Note: Setup and cleanup sections of the execution flow will be added to JADL in the future.

A method call on an object has the following syntax: the object name followed by the reference operator (dot) followed by the method name followed by the comma-separated list of 0 or more arguments in parentheses. Each argument needs to be a valid expression. The expression syntax follows intuitively familiar rules and is formally defined in JADL grammar.

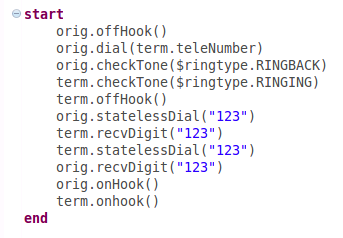


Figure 13. Basic call example

Any API call may succeed or fail. If an API call fails the test case fails immediately. If the required behavior for the passing scenario is for the API call to fail it needs to be preceded by the keyword **fail**. The actual meaning of the API call failure is defined by the underlying test technology and may vary from implementation to implementation.

The basic call example in Figure 12performs a two-way voice verification: orig to term and term to orig. This is done by dialing digits **“123”** and receiving them on the other end. If the test case needed to verify that the voice path only exists in one direction it could look like this:

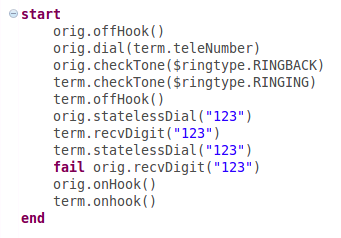


Figure 14. Basic call example with one-way voice path verification

The execution flow in Figure 14 verifies that no voice path exists from term to orig. If orig receives the digits dialed on term the test case will fail.

### Getters for Resource-Specific Data

As mentioned in the Test Data section, resource-specific data is retrieved by calling a getter defined in the resource API. In JADL a simple name of the data element will bind to the getter for this data element. For example, if the resource API defines a method with the name getTeleNumber and no arguments and if the variable term refers to a resource realizing this API then the syntax term.teleNumber will have the meaning of calling the getter getTeleNumber on this resource and passing no arguments.

It is important to note that even though this syntax “looks” like dereferencing a property of the resource it is semantically quite different. 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.

To the test case designer however the syntax suggests the simple meaning of referencing a resource-specific datum rather than worrying about contract/implementation relationship and having to use a more verbose method invocation syntax.

In the examples in Figure 13 and Figure 14 the originator is dialing the terminator’s number to properly route the call. Before the terminator is allocated to the test case by the DRM this number is not known because the terminator comes from a pool of Phone resources. The number needs to be obtained at runtime from the concrete resource. Even though this operation will ripple through multiple layers of the framework and will likely be implemented differently by different GPLs in multiple test execution platforms, to the test designer this is simply the number to call in order to reach the terminator: term.teleNumber.

### Symbol Lookup

Non-resource-specific test data comes from the test case context and is looked up dynamically as described in the Test Data section. Figure 10 also shows the relationship between the test case execution flow and the test data.

The syntax for the dynamic symbol lookup is the dollar sign operator followed by the fully qualified datum name that is being looked up. $ringtype.RINGBACK and $ringtype.RINGING in Figure 13 and Figure 14 are examples of dynamic symbol lookup of data elements ringtype.RINGBACK and ringtype.RINGING.

Just like in the case of resource-specific data getters, this syntax 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.

Section Test Case Data has an example of test data declaration in the test header that has two data schemas, one for general telephony and one for a specific customer, each providing their own symbol namespaces. This suggests that the data instance provided to the test case will have two separate data sets, each corresponding to its respective schema. These datasets may have conflicts. For example ringtype.RINGBACK may have one value in the *Telephony* data set and a different value in the *CustomerX* data set.

While it is possible to create different qualified names in different schemas to avoid ambiguity (e.g. customerX.ringtype.RINGBACK for *CustomerX*) it is not possible to enforce, especially if these data sets come from different organizations. Additionally, an argument can be made that using a more generic name makes the test case more reusable. In our example of a basic call it can run on any network, not necessarily *CustomerX*’s.

*Of course if the test case already declares the CustomerX data schema then the CustomerX data set better be available at run time because otherwise a test case with no compile errors can easily break at run time trying to look up a symbol that does not exist.*

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. Modeling of such data instances has to include a mechanism to customize the data set with additional data sets that get searched first. This is 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.

The ability to selectively specify which dataset to search first will be added to JADL in the future. 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.

### 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 **@HLF** element of the test case header as shown in the High-Level Functions section.

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 has to be a stateless component, but 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.

The way this is reflected in JADL syntax is that rather than calling a method on an API instance (e.g. orig is an instance of the Phone API – see Resource Declaration section) high-level functions are called on the API itself.

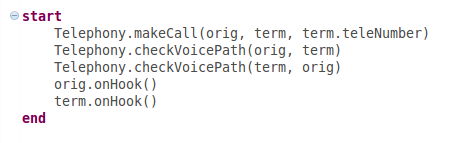


Figure 15. Basic call with high-level functions

The example in Figure 15 is the same basic call execution flow as in Figure 13 rewritten to use high-level functions. This example assumes that the **@HLF** element of the test case header points to an API definition that defines an API called Telephony. Please note that the last two statements are simple API calls. Even though they could also be rewritten as Telephony.terminateCall(orig, term), they were left as written to illustrate combining different levels of granularity in the same test case.

A high-level function call has the following syntax: the API name followed by the reference operator (dot) followed by the function name followed by the comma-separated list of 0 or more arguments in parentheses. Each argument needs to be a valid expression.

### Logging

In JADL logging is included in the actual language syntax. There are two main reasons for that. First, logging is an essential element of test case writing and second, since handling of the test case logs is done by the TEP most of the intricacies of general purpose logging are of no concern to JADL. Therefore it can provide a simple and intuitive logging syntax not worrying about the complexity of implementation. The actual logging implementation needs to be provided by the GPL into which JADL compiles.

The JADL logging syntax is the keyword **log** followed by a comma-separated list of expressions. There are three logging levels: **DEBUG**, **INFO**, and **ALWAYS** that are also JADL keywords. The default logging level is **INFO** unless the level is explicitly specified by preceding the **log** keyword with a level keyword.

It is up to the TEP to enable or disable test case logs at a particular level depending on the type of the test activity. The meaning of the logging levels is just what their names suggest: **DEBUG** logs are typically verbose and used for debugging test cases; **INFO** logs are the regular test case logs that are supposed to be seen in automation; and **ALWAYS** logs should very rarely, if ever, be used – they should be seen regardless of the TEP logging level setting.

One use case where **ALWAYS** logs may be useful is debugging performance test scenarios. Since logging is inherently very expensive because of string processing it requires, the common practice for running high levels of traffic is to disable all logs to improve performance of the test tools. The **ALWAYS** logs provide a mechanism to enforce logging of critical information even if all other logs are disabled.

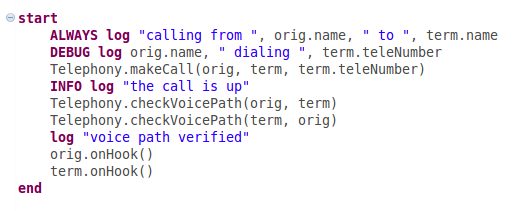


Figure 16. JADL Logging example

Figure 16 illustrates an example of JADL logging. The first log will be seen regardless of the TEP logging settings. In this somewhat artificial example the idea is that this log captures the names of the actual concrete resources being used which may be important to know in traffic if test cases fail intermittently. The second log will only be seen if the TEP enables the debug logs and will output the specific number being dialed on the originator. The third and the fourth logs are info logs that will be seen in regular automation runs. The fourth logs illustrates that specifying the **INFO** level is optional.

It must be noted that limiting JADL test case logs to three levels in no way limits the ability of the TEP to log at any level available to it. Most notably, the absence of error logs does not preclude the resource implementation (or any other part of the framework involved in test case execution) to output error logs when necessary. Error logs are not present in JADL because errors result in test case failures and it is assumed that a descriptive reason for the failure is provided to the TEP when a failure occurs. It is the responsibility of the TEP to provide the user with results of the test case execution in the format that may depend on the test scenario (individual, aggregated, etc.)

The following example in Figure 17 puts together everything discussed so far and shows a complete JADL test case that will compile and execute. This is a real test scenario that has been executed on a live IMS network in the Futurewei Plano Lab and used in multiple demonstrations internally and as part of JAD/JADL demonstrations internationally.

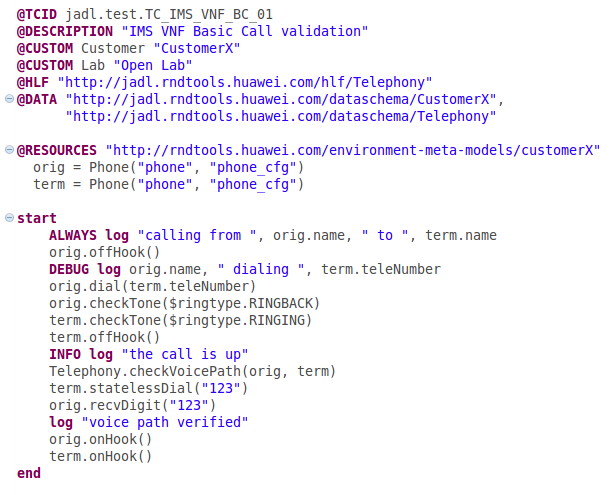


Figure 17. A complete JADL test case example

### Control Flow

Control-flow statements are used in most programming languages to allow multiple paths of control flow through computer programs. Execution of a control flow statement results in passing control to a specific part of the execution flow, usually depending on a condition. This is a common and very powerful mechanism without which writing computer programs other than most trivial ones is essentially impossible.

It would, therefore, be natural to require that a Test DSL have control-flow capability to allow branching and loops since they are so essential to general-purpose programming. The problem is that when writing a test case control-flow statements 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. For this reason JADL expressly disallows general-purpose conditionals (such as “if…then…else”) and general purpose loops (such as “while…do…”).

There are however specific use cases common to test case design that require a form of specialized control-flow statements. Such statements do not introduce branching, and therefore uncertainty of what was tested so the meaning of the test result remains clearly defined.

#### Assert Statement

Assert statement causes the test case to fail if a particular condition is not satisfied. It doesn’t create a branch in the execution flow so it continues to be sequential. It only provides an additional reason for the test case to fail at a particular point of the execution flow.

The JADL syntax for the assert statement is the keyword **assert** followed by a condition. The condition is a Boolean expression that follows the common rules for Boolean expressions in most programming languages. The supported operators are the left-associative OR written as “||”, the left-associative AND written as “&&”, and the NOT written as “!”. The supported operands are comparison expressions (defined below) and expressions resulting from applying Boolean operators to comparison expressions. The operator precedence is also common: “!” precedes “&&” and “&&” precedes “||”. The syntax also supports parentheses to alter the order of precedence as needed.

Comparison expressions compare two values and evaluate to either **true** or **false**. Each value needs to be a valid expression.

The operators currently supported within comparison expressions are:

* **equals** for content equality,
* “>” for numeric “greater than”,
* “>=” for numeric “greater than or equal to”,
* “<” for numeric “less than”,
* “<=” for numeric “less than or equal to”,
* **matches** for matching a string against a regular expression, and
* **contains** applicable to strings and collections.

All comparison expressions are binary so operator precedence and associativity rules do not apply.

The example in Figure 18 illustrates how an assert statement might be used in a test case. It has the same basic call execution flow as the example in Figure 13 but makes sure that the Telephony Application Server (TAS) is up and running before making the call. In order to do that it checks the status of TAS using a special TAS maintenance resource and asserts that the status equals the value of status.RUNNING element of test data.

Please note that before the TAS maintenance resource can be used in the test execution flow it has to be allocated to the test case by the DRM. Therefore the TAS allocation statement is added to the resource declaration section of the test case. The JADL validator will check that the corresponding resource abstraction is defined in the specified environment meta-model and that it implements the required API. Otherwise a compile error will be generated.

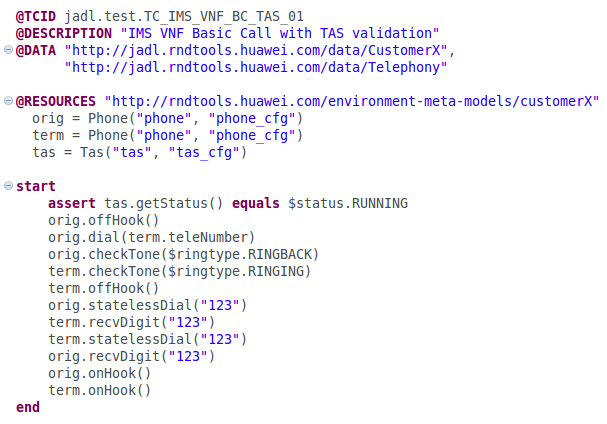


Figure 18. Basic call example with assertion

#### Attempt statement

Attempt statement causes the test case to repeat a particular action up to a predefined number of times and allow it to fail without failing the test case. If the action succeeds the control is passed to the statement following the attempt statement. If the maximum number of failed attempts is reached the test case fails. While reattempting the action versus proceeding with the rest of the execution flow is technically a form of conditional branching the important distinction is that all such branches are limited to the scope of the attempt statement and which branch is taken by any given execution is irrelevant to the test case pass/fail criteria.

A common use case for using the attempt statement is to invoke a potentially lengthy SUT action, completion of which does not result in any messaging that the test case can verify, and then check for its successful completion. The rationale for introducing this functionality into the language rather than pushing it into the test resource implementation is that it allows release of test case thread and associated computing resources between attempts. A blocking call on a test resource would tie up the test case thread for the duration of the SUT operation. Since this duration may be long and the TEP may be running a large number of test cases concurrently this can have a negative impact on the TEP performance. Additionally, this would also force multiple re-implementation of the attempt functionality in multiple resource types.

The JADL syntax for the attempt statement is the keyword **attempt** followed by an action to attempt followed by the specification of the attempt parameters. This specification has the following elements:

* Optional delay before the first attempt: the keyword **withdelay** followed by an integer delay value;
* Mandatory time interval between attempts: the keyword **every** followed by an integer interval value;
* Mandatory time unit for the delay and the time interval. The valid time units are **msec**, **sec** or **min**, indicating milliseconds, seconds or minutes respectively;
* Mandatory maximum number of attempts followed by the keyword **times**.

Actions can be any sequences of valid JADL execution flow statements.

The example in Figure 19 illustrates how an attempt statement may be used in a test case.

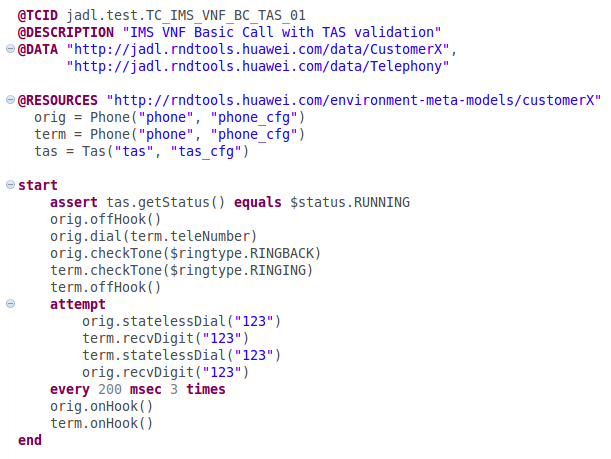


Figure 19. Basic call example with multiple attempts

Sometimes voice path verification can fail because the test system is too fast and it tries to verify the voice path before it has been established. This example is the same basic call example as in Figure 18 but three attempts to verify the voice path are made 200 milliseconds apart to allow the maximum of 400 milliseconds for the voice path establishment. A similar effect could of course be achieved without the attempt statement by introducing a time delay before the voice path is checked. The downside would be two-fold: first, the test case thread will be tied up for the duration of the time delay, and second, the test case will have to wait for the entire 400 milliseconds even if the voice path is established much quicker, within the first 100 milliseconds for example. Using the attempt statement the extra wait is limited to the maximum of the time interval between attempts (200 milliseconds in this example).

Another example of using the attempt statement is shown in Figure 20. In this case the test scenario not only requires for the TAS to be running before the call is made as in examples in Figure 18 and Figure 19, but also resets it and makes sure that it is back up and running after the reset. It assumes that it would take between 30 and 45 minutes for the TAS to reset.



Figure 20. Basic call with TAS reset example

This example is similar to the example in Figure 19 but this time the TAS is reset and 16 attempts are made one minute apart after a 30-minute initial delay to check if it is back up before making a call.

## Protocol Testing

When testing various telecommunications protocols the test case needs the ability to create, populate, send, receive, and verify protocol messages. Test resources involved in protocol tests are able to encode and decode their supported protocol stacks and even run their own state machines of low-level protocol messaging. The test case only controls protocol state transitions that are important to the test.

JADL assumes that the actual protocol implementation is provided by the implementing technology. However, it still needs to provide the facility for protocol message manipulation.

### Protocol Definition Header

Similarly to high-level functions and test data, the protocol element of the test case header is a comma-separated list of URLs following the **@PROTOCOL** keyword. Each URL specifies a protocol tree definition that will be loaded into a special module of the test case semantic model and used for protocol message validation and code assist/code completion.

A test case header example with a protocol header element is shown in Figure 21. It declares that the test case will be using the protocol definition (SIP in this example) found at URL http://JADL.rndtools.huawei.com/protocols/Sip.



Figure 21. Protocol test case header example

### Creating Protocol Messages

Once the protocol definition is loaded every protocol message defined in it becomes a message type in JADL. This is very similar to API names defined in the environment meta-model becoming valid resource types. The message instantiation syntax of JADL is the keyword **msg** followed by the message type and the message name as illustrated in Figure 22.



Figure 22. Protocol message instantiation example

Messages can be instantiated anywhere within the test execution flow and are placed into the global scope of the test case. This means that the same message can be used anywhere in the test including different test segments, but it also means that the message names have to be unique within the test case scope. Redefining a message will generate a compile error in JADL.

JADL will validate that the instantiated message is defined in one of the protocol definitions listed in the test case header. It will also provide code proposals and code completion for valid message types.

### Populating Protocol Messages

JADL models protocol messages as trees where every node is a protocol information element (IE) and nesting of the protocol elements is modeled as parent-child relationship. It needs to be clarified that individual protocol message instances are modeled as trees. The message meta-models defined in the protocol definition may contain cycles. This is particularly common for text-based protocols whose elements can be defined recursively. When the message is populated with values any such meta-model cycle will be traversed a finite number of times resulting in a finite depth of the model tree.

Protocol messages can also be models of protocol stacks. In this case more than one protocol definition meta-model is used to instantiate a message. A tree node of the message model containing a payload is modeled with a payload descriptor that includes a choice of available payloads and once the required payload is selected the payload meta-model is used to create a model of the payload sub-tree.

Protocol message modeling is generic and is not tied to a particular protocol implementation by an underlying test technology. The JADL execution engine for the GPL used in any individual technology is responsible for converting message models into test-technology-specific message models where they can be encoded and otherwise manipulated in the fashion adopted for its particular implementation.

JADL validation and code assist/code completion is based on message meta-model (or nested meta-models in the case of protocol stacks). At each node of the message tree the user can choose, and populate, a valid child sub-tree, which can be a nested element of the same protocol meta-model, a nested protocol meta-model root in the case of a payload descriptor, or a nested back-reference to an already traversed meta-model node in the case of a recursive definition.

JADL syntax of a message tree IE is a qualified (dot-separated) name representing the path to the IE from the message root. The example in Figure 23. Message population example illustrates population of a simple SIP message.

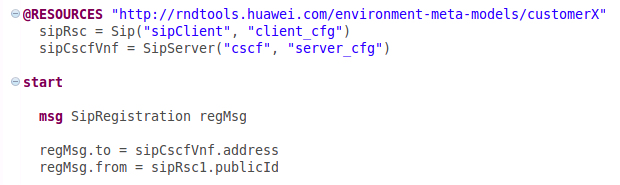


Figure 23. Message population example

First, in the resource declaration section two resources are defined: sipRsc that realizes API Sip and sipCscfVnf that realized API SipServer. Then, at the beginning of the execution flow a message regMsg is instantiated. The message type, SipRegistration, is one of defined message types in the Sip protocol meta-model.

The last two lines of this example is where the message is populated. The message IE to is set to be the address of the SCSF resource and the message IE from is set to be the public ID of the SIP client resource. Both IEs have qualified names where the message variable represents the root of the corresponding message type and each subsequent element of the path is the corresponding node name of the message meta-model. In this example to and from IEs are immediate children of the SipRegistration message root.

The JADL syntax for getting resource-specific data (described in section Getters for Resource-Specific Data) is also reused in this example: sipCscfVnf.address and sipRsc.publicId are JADL getters for data elements “address” and “publicId” defined in SipServer and Sip APIs respectively.

This example also introduces a new element of JADL syntax: a qualified name followed by “=” binds to a setter call on the model element represented by the part of the qualified name before the last dot for its child IE represented by the part of the qualified name after the last dot. For example, “a.b.c.e =“ will bind to a setter for e called on the a.b.c object. In our example “regMsg.to =” and “regMsg.from =” map to setter calls on regMsg for to and from IEs respectively.

As in the case for JADL getters that “look” like referencing an attribute, JADL setters “look” like assigning a value to an attribute. In neither case does JADL assume any particular implementation of the underlying model elements or their attributes. The only assumption that is made is that there is a setter in the API for the meta-model node that will populate the nested element of the model node whose (element’s) value is passed on the setter.

Protocol messages typically contain a large number of IEs. Populating all of them from the test case for every message is impractical and unnecessary. To allow the user to only set the IEs that are important to the test the rest of the IEs need to be populated automatically with default values. These default values may vary from organization to organization, from team to team, from user to user, and from test scenario to test scenario.

A flexible mechanism for providing customizable data sets to the test case already exists in JADL. This mechanism utilizes test data. Protocol defaults can be progressively customized from global to individual in the fashion described in the Symbol Lookup section.

When no default values can be found in the test data message IEs are populated with “default defaults” based on the type of the IE. Default defaults have an intuitive “uninitialized” meaning: zero for integers, empty string for strings, false for booleans, all zeroes for fixed-length binary data, etc.

A common requirement in protocol testing is the ability to set a protocol IE to a particular encoded value. This is often needed for error scenario testing when the normal message encoding mechanism needs to be bypassed to introduce an error. JADL provides a special syntax for setting encoded values: the value to be set is preceded with the keyword **encoded**, as illustrated in the example in Figure 24.

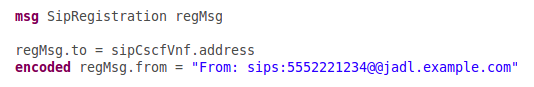


Figure 24. Setting encoded value

In this example the “from” IE is set to a malformed address.

While setting IE values is only allowed on the leaf nodes of the message model, setting encoded values is allowed on any node of the message. If an encoded value is set on a branch node the entire sub-tree under that node gets overridden by its encoded value. The codec implementations of the underlying test technologies must check if encoded values are set on branch nodes and bypass encoding of the message sub-trees under these nodes.

### Sending Protocol Messages

The JADL syntax for sending a protocol message on a test resource is the message name followed by the send operator “->“ followed by the resource name, as illustrated in Figure 25.



Figure 25. Sending protocol message example

This statement triggers population of the IEs of reqMsg that have not been set by the user with default values and passing it to sipRsc to be encoded and sent to the SUT.

### Receiving and Verifying Protocol Messages

In most cases the only action performed on messages received from the SUT is verifying that their IEs match a set of expected values. Rather than receiving the message, getting its IE values and comparing them to the expected values one by one, JADL has a “receive and verify” combined syntax that will match the received message to the expected one. The expected message is created and populated with the expected values beforehand. The following example in Figure 26 should clarify this concept.

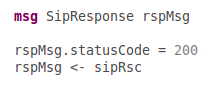


Figure 26. Verifying protocol message example

The first line instantiates an expected SipResponse message rspMsg. The second line populates its statusCode IE with the value 200, and the third line receives a message on sipRsc and verifies it against the expected message rspMsg.

The JADL combined syntax for receiving and verifying a protocol message on a test resource is the expected message name followed by the receive operator “<-“ followed by the resource name followed by the optional verification mode clause. The verification mode clause consists of the **verify** keyword followed by the verification mode keyword.

The currently supported verification modes are:

* **ALL** – all message IEs are verified, including those set by default
* **SET** – only IEs set by the user are verified, all other IEs are ignored (default)
* **NONE** – no verification of message IEs is performed, only the message type is verified.

More verification modes will be added in the future as JADL matures.

If no verification mode clause is present the default verification mode is **SET**. The example in Figure 26 does not have the verification mode clause, which means that only IEs set by the user (statusCode in this example) will be verified. If all IEs of the response message need to be verified the verification mode needs to be specified explicitly as shown in Figure 27.

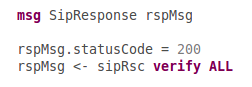


Figure 27. Verifying all IEs of the received message

If the message is not received within a timeout period (specified in test data) or if the received message fails to verify the test case fails with the appropriate cause. Similarly to API calls, if the required behavior for the passing scenario is for the verification statement to fail it needs to be preceded by the keyword **fail**, as illustrated in Figure 28 and Figure 29.

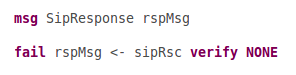


Figure 28. Verifying that message is not received

In the example in Figure 28 the test case will fail if a SipResponse message is received and continue running otherwise.

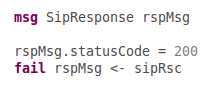


Figure 29. Verifying that the message is not received or not verified

In the example in Figure 29 the test case will fail if a SipResponse message is received and its statusCode IE value is 200 (default verification mode is **SET**) and continue running otherwise.

There may be scenarios where it is important that the verification statement fails because the message is received but failed verification. To support this scenario **failVerify** keyword should be used instead of the **fail** keyword as illustrated in Figure 30.

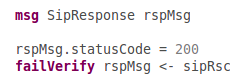


Figure 30. Verifying that the message is received bit not verified

In this example the test case will only continue running if SipResponse message is received and its statusCode IE value is not 200. It will fail otherwise.

In some cases IEs of the received message need to be referenced explicitly. A common use case is to populate an IE of the message to be sent with the value of an IE of previously received message. In this case in addition to verifying the received message it also needs to be returned to the test case for future reference.

JADL syntax for keeping the received message for the future use in the test case is assigning it to a message variable as illustrated in Figure 31. In this example the default verification mode is specified explicitly (for illustration purposes) even though it could be omitted.

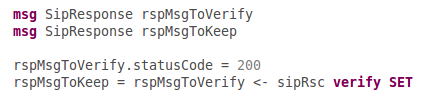


Figure 31. Keeping the received message for future use

In this example rspMsgToKeep will refer to the received message after it has been verified against rspMsgToVerify. If the received message does not verify the test case fails immediately.

If the required behavior for the passing scenario is for the message verification to fail and the received message still needs to be retained for the future use the **failVerify** keyword can be used in the similar fashion as before. The example in Figure 32 illustrates the corresponding JADL syntax.

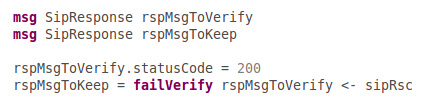


Figure 32. Keeping a received message that failed verification for future use

In this example the test case will continue running if a SipResponse message is received and its statusCode IE value is not 200. The variable rspMsgToKeep will refer to the received message.

Please note that the **fail** keyword cannot be used for this scenario. Using **fail** would allow the test case to continue running if the SipResponse message is not received at all. Clearly in this case rspMsgToKeep cannot refer to a non-existing message. JADL will generate a compile error if **fail** is used in place of **failVerify** in this context.

### Receiving messages in an unknown order

A common scenario in protocol testing is having to receive, and verify, a known set of messages when the order of their arrival is unknown. This often happens when messages arrive on different resources as a result of the same stimulus (a message sent on a test resource) generating multiple responses from the SUT sent on different interfaces.

JADL supports this type of scenario by defining a block of verify statements, each of which needs to succeed in order for the test case to continue running. Each statement will wait for its respective message and the order in which these statements get executed depends on the order of message arrivals and is not known to the test case beforehand.

The JADL syntax for defining such unordered blocks is the keyword **unordered** followed by one or more message receive/verify statements followed by the keyword **end\_unordered**, as illustrated in the example in Figure 33.



Figure 33. Receiving unordered messages

In this somewhat artificial example two SipRegistration messages are sent from two clients, sipRsc1 and sipRsc2, to the same server sipCscfVnf. The server waits for both of them before proceeding to reply. The order in which these registration requests arrive to the server is not known to the test case. While this example represents a poor testing practice of introducing unnecessary non-determinism into the test, it serves to illustrate JADL means of handling such non-determinism in much more realistic scenarios.

The **unordered** blocks provide support for the most common scenario of receiving and verifying messages in an unknown order. This may not be sufficient for some scenarios when, for example, an immediate response has to be generated once each message in an **unordered** block is received instead of waiting for all unordered messages to be received before any responses are generated. The **unordered** block functionality will be evolved as JADL matures.

# Conclusion

JADL is still in its infancy. This white paper presented a very general language framework with many more details still to be fleshed out. As JADL gradually matures parts of this white paper may become obsolete and other parts may be added in the future revisions.

The JADL grammar is still in the prototype stage: it supports the language features described here at the “demo” level but it lacks completeness, precision, and finesse. Many necessary constraints and cross-references are still missing. The grammar will undergo significant changes in the future.

An execution engine has been developed that generates valid Java code from the test case semantic models created from the meta-model generated from the JADL grammar. This Java code has been successfully executed on the Java-based Test Execution Platform utilizing the Dynamic Resource Management system, which made the basis of various JADL demonstrations. However, at present this system only supports very basic test scenarios and needs to be significantly evolved.

# References

[1] ETSI GR NFV-TST011 “Test Domain and Description Language Recommendations”

[2] Marcus Voelter et al. “DSL Engineering. Designing, Implementing and Using Domain-Specific Languages”