|  |  |
| --- | --- |
|  |  |

EPTF CLL LGenBase, Function Description

Contents

[1 Introduction 4](#_Toc453593691)

[1.1 Revision history 4](#_Toc453593692)

[1.2 How to Read this Document 5](#_Toc453593693)

[1.3 Scope 5](#_Toc453593694)

[1.4 Recommended way of reading 5](#_Toc453593695)

[1.5 Typographical conventions 5](#_Toc453593696)

[1.6 Abbreviations 5](#_Toc453593697)

[1.7 Terminology 6](#_Toc453593698)

[2 General Description 10](#_Toc453593699)

[2.1 Entities, behaviors, entity types, entity groups 10](#_Toc453593700)

[2.2 FSM tables 11](#_Toc453593701)

[2.3 Scenario and traffic case types 11](#_Toc453593702)

[2.4 Traffic management 13](#_Toc453593703)

[2.5 The states of an entity 14](#_Toc453593704)

[2.6 States of a traffic case 16](#_Toc453593705)

[2.6.1 The process of stopping and aborting a traffic case 17](#_Toc453593706)

[2.7 States of a scenario 17](#_Toc453593707)

[2.8 Start an entity from user code 18](#_Toc453593708)

[2.9 Originating and terminating traffic cases 18](#_Toc453593709)

[2.10 FSM, event handling 18](#_Toc453593710)

[2.10.1 Levels of events 18](#_Toc453593711)

[2.10.2 FSM tables 19](#_Toc453593712)

[2.10.3 Structure of FSM tables 20](#_Toc453593713)

[2.10.4 Event types in FSM-s 20](#_Toc453593714)

[2.10.5 Add/remove listeners 21](#_Toc453593715)

[2.10.6 Listener database 22](#_Toc453593716)

[2.10.7 Dispatch/post events 25](#_Toc453593717)

[2.10.8 FSM timers 26](#_Toc453593718)

[2.11 FSM tables in the traffic cases 26](#_Toc453593719)

[2.12 DTE handling in LGenBase 26](#_Toc453593720)

[2.13 Prototypes 27](#_Toc453593721)

[2.14 Instances 27](#_Toc453593722)

[2.15 Other structures 27](#_Toc453593723)

[2.16 Identifying prototypes 27](#_Toc453593724)

[2.17 Identifying instances 27](#_Toc453593725)

[2.17.1 Entity groups 27](#_Toc453593726)

[2.17.2 Entities 28](#_Toc453593727)

[2.17.3 Scenarios 28](#_Toc453593728)

[2.17.4 Traffic cases 28](#_Toc453593729)

[2.18 Describe configurations in configuration files 28](#_Toc453593730)

[2.18.1 Compact and indexed FSM tables in configuration files 29](#_Toc453593731)

[2.19 Use cases of the traffic case management 29](#_Toc453593732)

[2.20 Workflow of preparing the traffic case management 31](#_Toc453593733)

[2.21 External Templates 32](#_Toc453593734)

[2.21.1 Overview 32](#_Toc453593735)

[2.21.2 How to use the external templates 32](#_Toc453593736)

[2.21.3 Runtime modification of template types 37](#_Toc453593737)

[2.22 FSM table declaration 39](#_Toc453593738)

[2.22.1 EPTF variables and statistics in FSM-s 39](#_Toc453593739)

[2.22.2 Referring to FSM objects 40](#_Toc453593740)

[2.22.3 Naming of the FSM objects 40](#_Toc453593741)

[2.22.4 Hookpoints in FSM tables 41](#_Toc453593742)

[2.23 FSM tables in traffic cases 42](#_Toc453593743)

[2.24 Communication with FSM tables 43](#_Toc453593744)

[2.24.1 API functions to dispatch/post events 43](#_Toc453593745)

[2.24.2 Communication among the FSM tables 43](#_Toc453593746)

[3 Functional Interface 46](#_Toc453593747)

[3.1 Naming Conventions 46](#_Toc453593748)

[3.2 Using names 46](#_Toc453593749)

[3.3 Public Functions 46](#_Toc453593750)

[3.3.1 Initialization 46](#_Toc453593751)

[3.3.2 Declaring functions 47](#_Toc453593752)

[3.3.3 Declaring behavior types 48](#_Toc453593753)

[3.3.4 Declaring entity types 49](#_Toc453593754)

[3.3.5 Creating entity groups 49](#_Toc453593755)

[3.3.6 FSM table declaration 49](#_Toc453593756)

[3.3.7 Declaring step context arguments 51](#_Toc453593757)

[3.3.8 Declaring FSM tables with f\_EPTF\_LGenBase\_declareFSMTables 55](#_Toc453593758)

[3.3.9 Obsolete versions of declaring FSM tables 55](#_Toc453593759)

[3.3.10 FSM variable accessing functions 55](#_Toc453593760)

[3.3.11 Step-based FSM object accessing functions 56](#_Toc453593761)

[3.3.12 Declaring traffic case types 57](#_Toc453593762)

[3.3.13 Declaring scenario types 58](#_Toc453593763)

[3.3.14 The EPTF\_LGenBase\_ScenarioTypeDeclarator record 58](#_Toc453593764)

[3.3.15 Traffic cases of scenarios 59](#_Toc453593765)

[3.3.16 Associate scenarios with entity group 60](#_Toc453593766)

[3.3.17 Start traffic case 61](#_Toc453593767)

[3.3.18 Stop traffic case 61](#_Toc453593768)

[3.3.19 Abort traffic case 61](#_Toc453593769)

[3.3.20 Handling of Burst Calculation methods 61](#_Toc453593770)

[3.3.21 Managing Phases 62](#_Toc453593771)

[3.3.22 Target CPS Change Handling 63](#_Toc453593772)

[3.3.23 Limited Execution 64](#_Toc453593773)

[3.4 Predefined test steps 64](#_Toc453593774)

[3.4.1 Steps administering the start of an entity 64](#_Toc453593775)

[3.4.2 Steps reporting finish of execution of an entity 66](#_Toc453593776)

[3.4.3 Steps replying to stop/abort/reset LGenBase events 68](#_Toc453593777)

[3.4.4 Timer handling steps 69](#_Toc453593778)

[3.4.5 FSM variable handling steps 71](#_Toc453593779)

[3.4.6 FSM chronometer manipulation steps 77](#_Toc453593780)

[3.4.7 Test management steps 78](#_Toc453593781)

[3.4.8 Sibling FSM event management steps 79](#_Toc453593782)

[3.4.9 Event stack handling steps 82](#_Toc453593783)

[3.4.10 FSM DebugLight 85](#_Toc453593784)

[3.5 Predefined functions 85](#_Toc453593785)

[3.5.1 Entity success decision functions 85](#_Toc453593786)

[3.6 LGenBase events 86](#_Toc453593787)

[3.6.1 Traffic management events 86](#_Toc453593788)

[3.6.2 Entity execution result report events 87](#_Toc453593789)

[3.6.3 Answering traffic management events 88](#_Toc453593790)

[3.6.4 Traffic state notification events 88](#_Toc453593791)

[3.6.5 Other events 90](#_Toc453593792)

[3.6.6 LGenBase events and their responses 90](#_Toc453593793)

[3.7 Built-in statistics 91](#_Toc453593794)

[3.8 GUI support 92](#_Toc453593795)

[3.9 Generalization 94](#_Toc453593796)

[3.10 Public functions for EPTF LGenBase 94](#_Toc453593797)

[4 References 94](#_Toc453593798)

# Introduction

## Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Rev | Characteristics | Prepared |
| 2007-12-06 | PA1 | First draft version | EGBOZIE, ELSZSKU |
| 2007-12-06 | PA2 | Updated after review | EGBOZIE |
| 2008-02-01 | PA3 | LGenBaseStats feature | ELSZSKU |
| 2008-02-18 | PA4 | LGenBaseStatsUI feature | ELSZSKU |
| 2008-02-26 | PA5 | Public functions of LGenBaseStats and LGenBaseStatsUI | ELSZSKU |
| 2008-03-10 | PA6 | More detailed descriptions of LGenBase | ELSZSKU |
| 2009-01-27 | PB1 | New event dispatch methods | ELSZSKU |
| 2009-01-28 | PB2 | Corrections after review | ELSZSKU |
| 2009-02-06 | PB3 | External templates feature added | EISTFAL |
| 2009-02-19 | PB4 | Updated after external templates review | EISTFAL |
| 2009-02-26 | PB5 | Burst calc functions and phase functions added | EBENMOL |
| 2009-03-23 | PB6 | FSM variables and statistics, finish actions per conditions, FSM statistics and variables | ELSZSKU |
| 2009-05-22 | PB7 | Runtime modification of template types | EISTFAL |
| 2009-07-14 | PB8 | Explaining the stateful handling of external template argument callbacks | EGBOZIE |
| 2009-08-27 | PC1 | State definition of FSM cells in declaration | ELSZSKU |
| 2009-11-11 | PC2 | Cancel all timers FSM step,  New function description | EGBOTAT, EANDRKS |
| 2010-01-08 | PC3 | FSM even and FSM step descriptions, CPS change callback | ELSZSKU,ETHJGI |
| 2010-10-11 | PD1 | Updated step lists | EJNOSVN |
| 2010-10-26 | PD1 | Multi FSM in traffic cases, communication with FSMs | ELSZSKU |
| 2010-11-05 | PD2 | Added StatHandler-Provider relation according to scope into the section “FSM table declaration” | EJNOSVN |
| 2010-11-19 | PD3 | FSM hookpoints | ELSZSKU |
| 2011-01-19 | PE1 | Limited execution | EJNOSVN |
| 2011-07-18 | PF1 | New timer test steps | EBALLUG |
| 2012-07-05 | PG1 | FSM DebugLight | ELSZSKU |
| 2014-06-26 | PH1 | Callback for missing template parameter | EJNOSVN |
| 2014-09-02 | PH2 | Added parameter restoreTCAtStartTC | EDNISND |
| 2014-09-03 | PH3 | Modified parameter restoreTCAtStartTC | EDNISND |
| 2014-09-22 | PH4 | Added section DTE handling in LGenBase | EDORNAG |
| 2014-10-09 | PH5 | Documentation numbers updated | ESZILSZ |
| 2014-11-21 | PH6 | Added comment on DTE handling | ESZILSZ |
| 2014-12-02 | H | Updated for release | ESZILSZ |
| 2015-05-20 | PJ1 | Graphics replaced with new ones | ESZILSZ |
| 2015-06-19 | J | Updated for release and graphics changed back to the old ones. | ESZILSZ |
| 2016-03-22 | PK1 | Fixed position based substitution for external template | EZOLZSI |
| 2016-06-09 | K | Updated for release | ESZILSZ |

## How to Read this Document

This is the Function Description for the LGenBase feature of the Ericsson Performance Test Framework (TitanSim), Core Library (CLL). EPTF CLL is developed for the TTCN-3 [1] Toolset with TITAN [2]. For more information on the EPTF CLL please consult the Product Revision Information.

## Scope

This document is to specify the content and functionality of the LGenBase feature of the EPTF CLL. This feature is the basis for all load generators (LGens) components of TitanSim, that is, components whose main purpose is to *generate messages.*

## Recommended way of reading

The readers are supposed to get familiar with the concept and functionalities of EPTF CLL [4]. They should get familiar with the list of acronyms and the glossary in Section 1.7 and 0, respectively.

## Typographical conventions

Important concepts are denoted by *italic* font wherever they are first used in the given context. Moreover, whenever a concept is mentioned that has a special meaning as described in the Glossary (Section 0) of this document, then these occurrences are marked with an initial arrow, e.g., *🡪 TitanSim Statistics*.

## Abbreviations

CLL Core Library

CSCF Call Session Control Function

EPTF Ericsson Performance Test Framework

EPTF\_Var is the abridged name of the EPTF\_CLL\_Variable type variables

FBQ Free-busy queue

FSM Finite State Machine

FSM variables are EPTF\_CLL\_Variables declared in FSM table declarations

GGSN Gateway GPRS Support Node

GUI Graphical User Interface

IMS IP Multimedia Subsystem

ISUP Integrated Services Digital Network (ISDN) User Part Protocol

LGen Load Generator

SIP Session Initiation Protocol

SUT System Under Test

TitanSim New synonym for the EPTF Framework

TTCN-3 Testing and Test Control Notation version 3 [1]

UA User Agent

## Terminology

*Application Library (Applib)* is that part of the TitanSim software that is although protocol, or application-area dependent, but can be *reused* across many TitanSim applications. Applibs are dependent on the CLL.

*Behavior context* is an index-able (set of) data record(s) that stores data and state information of a given entity with respect to a given *🡪 behavior type.* Storage for behavior contexts is to be provided by the respective Applibs.

*Behavior type* is a concept for collectively referring a set of *🡪 behavior contexts, 🡪FSM inputs, 🡪 test steps,* andfunctions realizing methods conforming to given function signatures prescribed by the LGenBase feature. Behavior types are to be declared dynamically during run-time to the LGenBase by the component-type initialization function of some Applib. Usually an Applib product has only one behavior type, but it is permitted for Applibs to declare more than one behavior types, if necessary. Behavior types are identified by their indices determined by the order of declaration and the declarations are stored in v\_LGenBase\_behaviorTypes

*Control Logic* isthat part of the TitanSim software that is specific to a particular TitanSim application program. It is usually not directly reusable and is built upon the CLL and several AppLibs.

*Entities* are “things/objects” that are simulated by the LGen. Such “things” are usually the “users”, “calls”, “servers”, “terminations”, or anything whose external behavior is to be simulated by the LGen. Usually an LGen simulates more than one entity.

*Entity context* is an index-able generic data record describing an entity. It stores index cross-references to *🡪 behavior contexts* and it directly stores the *🡪 FSM contexts* of the entity. The entity contexts are identified via their indices with respect to the entity context database stored in the v\_LGenBase\_entities component variable.

*Entity group* A group of entities with the same entity type. The indices of the entities of the group make a continuous interval that does not overlap with the index range of any other entity group.

*Entity type* is a named list of references to declared *🡪 behavior types*. As such, it declares what type of behavior contexts is available on entities of this type.

*Event listener* is a call-back function to be dynamically registered that will be called whenever an *🡪 event reporter* calls f\_EPTF\_LGenBase\_dispatchEvent

*Event reporter* is analtstep, or function of a given *🡪 behavior type* that eventually calls f\_EPTF\_LGenBase\_dispatchEvent

*Finite State Machine* is a formal description of *🡪 Control Logic*. It is a tabular description of what *🡪test steps* to call when a given *🡪reported event* occurs, when the entity is in a given declared 🡪 *state of the FSM*. After executing these *🡪 test steps* an optional next *🡪 FSM state* can be also defined.

*FSM table* is a formal specification of the communication rules of Finite State Machines. These indexable records are stored in v\_LGenBase\_fsmTables. FSM tables realizing *🡪 traffic cases* must obey certain rules.

*FSM state* is a locally indexed declaration of a state within an FSM table.

*FSM timer* is a locally indexed declaration of a timer within an FSM table.

*Group finish condition* is list of conditions that defines situations when a traffic case is over on group level, that is, no new traffic is launched on that group. Group finish conditions are evaluated whenever any entity of the group reports the event that it has just finished executing a traffic instance of that traffic case.

*"Group finish actions"* is list of action that is executed when any of the 🡪 group finish conditions has been met.

*LGenBaseStats* is an extension of the LGenBase. It provides accessibility to component variables of traffic cases through EPTF\_Var variables.

*LGenBaseStatsUI* is an extension of the LGenBaseStats, which provides GUI support for LGenBase traffic cases and scenarios using the EPTF\_Var support of the LGenBaseStats.

*Named functions* are functions that have an associated name (label), are declared to LGenBase metadata with that label; and can be referred by their labels.

*Reportable event* is a declared notification message of a given *🡪 behavior type* that can be distributed to *🡪 event listeners* by 🡪 *event reporters* via f\_EPTF\_LGenBase\_dispatchEvent

*Reported event* is an instance of a *🡪reportable event* supplied with reported arguments that is actually being distributed via f\_EPTF\_LGenBase\_dispatchEvent.

*Scenario* is a collection of traffic cases deployed on an entity group

*StatHandler* is a feature of the CLL. See [5][6]

*StatMeasure* is a feature of the CLL. See [5]

*Test step* Test steps are functions whose signature conforms to a given function type. They *must not* contain TTCN-3 statements with blocking semantics.

*EPTF Core Library (CLL)* is that part of the TitanSim software that is totally project independent. (i.e., which is not protocol-, or application-dependent). The Core Library is to be supplied and supported by the TCC organization. Any Core Library development is to be funded centrally by Ericsson.

*Traffic case type* is a prototype used as an input for instantiating traffic case instances on a specific entity group. It defines which *🡪 FSM table* to use, which *🡪 entity type* must be used by the *🡪 entity group* where the *🡪 traffic case* is to be deployed

*Traffic case* is a behavior, defined by an FSM table, that is executed on a given entity group. All entities of the entity group participate in executing the traffic case. An entity group may have more than one concurrent traffic cases.

*User finish actions* is list of action that is executed when any of the *🡪 user finish* conditions have been met.

*User finish condition* is list of conditions that defines situations when a *🡪 traffic case* is over for a given entity, that is, the entity is not considered for selection for launching a new traffic, even if the entity returns to its idle *🡪 state*. User finish conditions are evaluated whenever any entity of the group reports finishing the traffic case, that is, returning to the “idle” state of the respective FSM context.

*External templates* are parameterized text modules are stored by LGenBase. Parameter values can be substituted dynamically during run-time.

*Template type* is a named declaration of a template assigned to a behavior type.

*Template set* is set of templates belonging to a traffic case. It is a list of <template type, external template> pairs.

# General Description

This document specifies the Load Generator Base (LGenBase) feature of the TitanSim CLL.

The EPTF LGenBase feature makes it possible to

* Develop load generator component with built-in support for effectively executing multiple FSM behaviors concurrently and independently
* Use built-in support for timer handling of FSM tables
* Provide basic traffic case management (scenario)

The aim of the EPTF LGenBase feature is to make the writing of efficient reusable load generators simple.

## Entities, behaviors, entity types, entity groups

A TitanSim application tests the behaviors of “entities”. An entity can be a complete media gateway, or a termination-instance of media gateway, or a SIP UA, or an IMS CSCF, or a GGSN, etc. Usually an entity is a user of the SUT as described on Figure 1.

System Under Test

entity#5

entity#6

entity#7

entity#8

entity#N

entity#4

entity#3

entity#2

entity#1

SIP

ISUP

SIP

ISUP

SIP

ISUP

SIP

ISUP

ISUP

SIP

ISUP

ISUP

SIP

Figure 1 A typical N user – 1 implementation configuration

The entity type is a defined set of behaviors that can be simulated by the entities. Zero, one, or many behavior types can be simulated on any entity.

Usually the tests are performed on a set of entities working similarly. This continual group of entities having the same behavior types is an entity group.

## FSM tables

The behavior of an entity is mostly described by FSM tables. The LGenBase manages the traffic through FSM tables, the other parts of the TitanSim (e.g. application libraries, transport layers, etc.) communicate through event dispatches to FSM tables.

## Scenario and traffic case types

A traffic case is a logical part of the test. Let’s get a sample:

In the test scenario first all the entities must “register”. Every entity must try to register as long as it can’t register successfully. When an entity successfully “registered”, it must “re-register” periodically. When all the entities are registered successfully, the re-registration must stop, and they must “generate traffic” for 40 seconds. In this scenario we can form three traffic cases: registration, re-registration, traffic generation.

Scenario types join the traffic cases; describe the flow of the test. They contain one or more traffic cases and describe their relations and working behavior:

* The delay of start of the traffic after the start of the traffic cases
* The enabled state (start) of the traffic cases at the start of the scenario
* The enabled state of the entities at the start of the traffic cases
* The conditions, when to finish launching traffic on traffic cases
* The conditions, when to finish launching traffic on an entity
* Actions what to do when an entity has finished
* Actions what to do when a traffic case has finished
* Range parameters of the traffic cases
* Custom string parameters of the traffic cases

In our sample the scenario contains three traffic cases as we saw above. The registration must start at the start of the scenario with all the entities enabled. The re-registration must start at the start of the scenario, but none of the entities available. The traffic generation traffic case must be disabled at the start of the scenario.

In the registration the finish condition of the entities must be one successful traffic, so after a successful registration it won’t continue the registration process. The action at the finish of an entity must be enabling the entity on the re-registration traffic case. The finish condition of the registration traffic case must be as many successful traffic, as many entities there are. The action at the finish of the registration traffic case must be: disabling (stop) the re-registration traffic case, and enabling (start) the traffic generation traffic case with enabling all the entities in it. And finally the finish condition of the traffic generation traffic case must be 40 seconds execution time. And we implemented our sample test.

More than one scenario type can be associated with the same entity group at the same time, as it’s described on Figure 2.

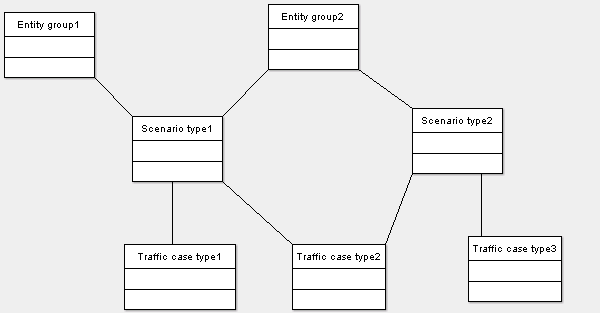


Figure 2 Entity group, scenario and traffic case associations

Scenario types must be associated with entity groups. At this time LGenBase creates the entity group, entity, scenario and traffic case instances.

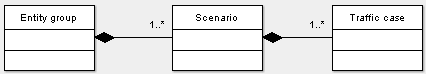


Figure 3 Composition of the entity group, scenario, traffic case trio

## Traffic management

The LGenBase uses FSM tables to manage the traffic.

Each traffic case has at least one associated FSM table. The first one is used to dispatch the messages to start, stop or abort the entities.

The LGenBase picks the first available entity from the associated entity group, starts traffic on the entity, and administers the result of it.

Starting traffic on an entity means that the LGenBase dispatches a c\_EPTF\_LGenBase\_inputName\_testMgmt\_startTC FSM event to the FSM of the appropriate entity, marks it as busy, increments the appropriate statistics. At the end of the traffic the entity must report the result by dispatching c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcSuccess, c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcFail, c\_EPTF\_LGenBase\_inputName\_testMgmt\_finishedTcError or c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcTimeout generic event as described on Figure 4.



Figure 4 Traffic handshake

The LGenBase checks the finish conditions of the entity (see 2.3). If one of them comes true, the entity remains unavailable; otherwise it will be placed back to the end of the list of available entities. So the order of the available entities depends on their returning order.

At the end of each traffic the LGenBase checks entity and group finish conditions and whether the traffic in the traffic case has been finished. If one of them comes true, LGenBase executes the declared actions and disables (stops) the traffic case also as described in the Figure 5.

[Match]  
Execute entity finish actions

[Not match]   
Put back entity to the free chain

Check entity finish conditions

[Match]  
Execute group finish actions

Check group finish conditions

[Not match]   
No action

[Match]  
Execute traffic finish actions

Check finish of launched traffic

[Not match]   
No action

Figure 5 Activity of the LGenBase when a traffic finished response has been received

## The states of an entity

All the entities in the traffic case have a state. These states are:

* none
* pass
* fail

While there was no traffic started on an entity, its state is “none”. At the end of each traffic LGenBase sets the state of the entity to “pass” or “fail”.

In the traffic case descriptor users can declare a custom function to calculate that state. See section 3.5.1.1 and 3.5.1.2. Every time LGenBase receives a traffic finished answer, calls this function to specify the new state of the entity.

If there is no user defined function specified, the state of the entity will be “pass” only if there had been only successful traffic on the entity earlier.

There are predefined functions to make it easy. See section 3.5.1.

If the aMode transfer mode in the enableEntity4Tc, enableAllTc, disableAllTc or enableTc group finish actions is enableIfPreviousSuccess, the entity will be available in the specified traffic case only when its state was “pass”.

## States of a traffic case

Figure 6 describes the state transitions of a traffic case.

Stopped

Stopping

Idle

Running

Paused

Aborting

Aborted

Finished

Terminated

[All the entities answered with c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityAborted]

[After tsp\_EPTF\_LGenBase\_abortStopProcess]

[All the started entities reported their finish]

f\_EPTF\_LGenBase\_restoreTC()

f\_EPTF\_LGenBase\_restoreTC()

[Match group finish condition]

f\_EPTF\_LGenBase\_abortTrafficCase()

f\_EPTF\_LGenBase\_startTrafficCase()

f\_EPTF\_LGenBase\_pauseTrafficCase()

f\_EPTF\_LGenBase\_pauseTrafficCase()

[All the entities answered with c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityStopped]

[After tsp\_EPTF\_LGenBase\_abortStopProcess]

Figure 6

### The process of stopping and aborting a traffic case

When the user stops a traffic case, LGenBase sends a c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_stopTC FSM event to all the FSM-s of the entities of the traffic case, and sets the state of the traffic case to c\_EPTF\_LGenBase\_tcStateStopping. Then the entities can walk through their stopping process, and at the end they should reply with a c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityStopped generic event. When all the entities replied, LGenBase sets the state of the traffic case to c\_EPTF\_LGenBase\_tcStateStopped.

In some cases not all the entities can reply in acceptable time interval. Therefore LGenBase waits for a time period described by the tsp\_EPTF\_LGenBase\_abortStopProcess module parameter, and then it sets the state of the traffic case to c\_EPTF\_LGenBase\_tcStateStopped and writes a warning message to the log file.

The process is the same in case of aborting the traffic case with messages c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_abortTC and c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityAborted, and states c\_EPTF\_LGenBase\_tcStateAborting and c\_EPTF\_LGenBase\_tcStateAborted.

A really important use of the implementation of these steps in the FSM-s of the entities is to set back the state of the FSM to idle state.

## States of a scenario

When the state of a traffic case changes the state of the scenario will be calculated. The calculation process goes by the following priorities:

If there is at least one running traffic case the scenario will have c\_EPTF\_LGenBase\_tcStateRunning state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStatePaused, the scenario will have c\_EPTF\_LGenBase\_tcStatePaused state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStateAborting, the scenario will have c\_EPTF\_LGenBase\_tcStateAborting state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStateStopping, the scenario will have c\_EPTF\_LGenBase\_tcStateStopping state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStateAborted, the scenario will have c\_EPTF\_LGenBase\_tcStateAborted state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStateStopped, the scenario will have c\_EPTF\_LGenBase\_tcStateStopped state.

If there is at least one traffic case having state c\_EPTF\_LGenBase\_tcStateFinished, the scenario will have c\_EPTF\_LGenBase\_tcStateFinished state.

The only exception is the pausing a weighted scenario, when the scenario has c\_EPTF\_LGenBase\_tcStatePaused state while the traffic cases have c\_EPTF\_LGenBase\_tcStateRunning state.

## Start an entity from user code

There are cases when the users don’t want to use the traffic management of the LGenBase to generate traffic. Probably they want to reflect to events generated in the outside world, but they want to use the benefits of the traffic management such as traffic case finish conditions and action, built-in statistics, etc. There are predefined steps to do this. See 3.4.1.

## Originating and terminating traffic cases

When the traffic case initiates the traffic launch, we call it originating traffic case. But sometimes there are traffic cases which don't launch traffic, they only react to outer events (work as a proxy or the target of the traffic). They are called terminating traffic cases.

Users can declare the traffic type of a scenario, and the traffic cases of a scenario must be the same type.

## FSM, event handling

An event is a record with several fields, signaling that "something happened". Users of the LGenBase can create "listeners". These functions execute the reactions to the events. These listeners can be registered. During the registration the users describe the parameters of the event they want to listen to, and the listener which must be executed when the event happens.

### Levels of events

An event can have the following parameters:

* behaviorIdx: the behavior index of the target. Usually all the features using the LGenBase have their own behavior index
* inputIdx: the ID of the event in the specified behavior
* targetEntityIdx: the absolute index of the target entity
* targetFsmCtxIdx: the FSM context index of the target entity identifying the FSM to which the event should be dispatched
* sourceEntityIdx: the absolute index of the entity which sent the event
* sourceFsmCtxIdx: the FSM context index of the source entity where the event had been sent from

During event listener activation the user must specify some of these parameters. Whenever an event is dispatched the listener will be executed if these parameters are matching exactly. Not all parameters must be specified. The parameters that are left out won’t be taken into account during the event dispatch. The accepted combinations are described in Table 1.

The following abbreviations are used:

* bIdx: behavior index.
* iIdx: input (event) index
* tEIdx: target entity index
* tFIdx: target FSM context index
* sEIdx: source entity index
* sFIdx: source FSM context index

Table 1 Event parameters

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Event type** | **bIdx** | **iIdx** | **tEIdx** | **tFIdx** | **sEIdx** | **sFIdx** |
| Behavior | M | - | - | - | - | - |
| General | M | M | - | - | - | - |
| Entity | M | M | M | - | - | - |
| FSM | M | M | M | M | - | - |
| EntityIndex | - | - | M | - | - | - |
| FSMOfEntity | - | - | M | M | - | - |
| BehaviorOfEntity | M | - | M | - | - | - |
| SourceEntity | M | M | - | - | M | - |
| SourceFSM | M | M | - | - | M | M |
| SourceEntityIndex | - | - | - | - | M | - |
| FSMOfSourceEntity | - | - | - | - | M | M |
| BehaviorOfSourceEntity | M | - | - | - | M | - |

### FSM tables

The FSM tables are special descriptors of the event listeners.

An FSM table of the LGenBase describes the responses given to specified FSM events in specified states.

### Structure of FSM tables

An FSM table of the LGenBase contains the name of the FSM table, a list of available states of the FSM represented by their names, a list of FSM timer available in the FSM, and table rows.

Each table row is responsible for handling an event. Table rows contain the description of the event they are responsible for, and a list of cell rows.

It is possible to define not only one but a set of events to a certain cell row by FSM table declaration. In this case the reaction defined by the cell row occurs to all events that are member of the event set. The specified event set can contain *single events*, *event lists* and *event ranges*. An event range can be defined by the name of first and last events. There are two kind of special events supported by the FSM tables: *catchall* means that the associated listener is executed to all events, while *unhandled event* listener is executed to all events, which is not handled by any other registered listener. If a catchall listener already registered, unhandled event listeners are never executed.

The ordinal number of the cell row describes the ordinal number of the state in which the reactions described in the following must be executed.

Each cell row contains the list of reactions must be given to the event specified in the table row being in the state described by the ordinal number of cell row.

The cell row contains also a reference to a method to calculate the state the FSM must step into after executing the actions described in the cell row, or the ordinal number of the next state.

If any of the described items is empty (e.g. the FSM doesn’t handle the specified event in the specified state, doesn’t have next state calculation method, etc.), the field must be omit.

If the next state calculation method is omit, and the next state is omit too, the FSM remains in the same state after executing the actions.

There are two kinds of FSM-s. In compact FSM tables the cells of the FSM-s are function references. In indexed FSM-s, they are registered test steps referred by their names or indexes, but finally there are test step functions.

### Event types in FSM-s

In the FSM tables users can specify the following event types:

* general
* entity
* FSM

When the user specifies "general" as the listened event type, the cell row will be executed whenever an event is dispatched with the specified behavior and input index, regardless of the target entity index and target FSM context index specified in the dispatched event. Practically the FSM cell row behaves as a general event listener.

Similarly, when the user specifies "entity" as the listened event type, the FSM cell row behaves as an entity event listener.

However, the "general" and "entity" event listeners in FSM tables are not recommended. These event types can cause misunderstandings in the behavior of the FSM execution, and can cause great memory consumption if there are a lot of entities in the appropriate entity group. The suggested way is declaring general or entity event listeners, and processing the events in these listeners.

### Add/remove listeners

The listeners of each type of events have the same function header. So every listener functions can be registered listening to any type of events, except FSM events. (FSM event listeners can be described only in FSM tables.)

There is a function to register a listener to the system:

function f\_EPTF\_LGenBase\_addListener(  
 in EPTF\_LGenBase\_EventListener\_FT pl\_fn,  
 in EPTF\_IntegerList pl\_args  
 )  
 runs on EPTF\_LGenBase\_CT  
 return integer

And of course there is a function to remove the listeners:

function f\_EPTF\_LGenBase\_removeListener(in integer pl\_idx)  
 runs on EPTF\_LGenBase\_CT

Simply adding a listener is not enough to execute it. The listener must be activated for a specific event. This way the same listener can be used for several different events, and only the listenerIdx must be used as a pointer.

Each listener type has its function to activate and deactivate function pair. Their names are:

* f\_EPTF\_LGenBase\_activate<event type name>Listener
* f\_EPTF\_LGenBase\_deactivate<event type name>Listener

### Listener database

Since all the listeners have the same signature, and the listeners can listen to all kinds of events (except FSM events), they are stored in a homogenous flat database. The lists that join the listeners to the events contain the indexes of the listeners in this listener list.

The implementations of the event-listener associations are different, they depend on the event type. At the bottom of all applied solutions there is the FBQ as the basic storage of the list of pointers to the listeners. The busy items of the FBQ-s hold the indexes of the active listeners. The indexes are stored in the first ([0]) element of the data field of the FBQ elements.

*Behavior, EntityIndex, GeneralSource, General, SourceEntity, FSMOfEntity, SourceFSM*

In case of these event types the list of the listeners is stored in a single FBQ.

Table 2 collects how to access the listener lists in these event types. Invoking the listeners means walking through the busy chain of the appropriate FBQ and calling the listeners addressed by the first data field of the busy items.

Table 2 The access of the listener lists

|  |  |
| --- | --- |
| **Event type** | **FBQ** |
| Behavior | v\_LGenBase\_behaviorTypes[bIdx].listeners |
| General | v\_LGenBase\_genericEventListeners[bIdx][iIdx] |
| EntityIndex | v\_LGenBase\_entities[tEIdx].entityIdxListenerList |
| SourceEntity | v\_LGenBase\_entities[sEIdx].sourceEntityListenerList |
| FSMOfEntity | v\_LGenBase\_entities[tEIdx].fsmCtxList[tFIdx]. fsmOfTargetEntityListeners |
| SourceFSM | v\_LGenBase\_entities[sEIdx].fsmCtxList[sFIdx]. fsmOfSourceEntityListeners |
| GeneralSourceFSM | v\_LGenBase\_entities[sEIdx].fsmCtxList[sFIdx]. fsmOfGeneralSourceFSMListeners |

*Entity, GeneralSource, BehaviorOfEntity, BehaviorOfSource, GeneralSourceFSM*

In case of these events it would have been too expensive to create a database providing direct indexing access to the listener lists of the separated events. Therefore there are arrays of listener list FBQ-s collecting the listeners of separated events, and hashmaps to find which FBQ-s should be used.

So the accessing of listeners of these element consists the following steps:

* find the appropriate listener list array and hashmap by indexing
* find the index of the appropriate listener list FBQ index in the hashmap
* invoke the listeners pointed by the FBQ as it's described in 0

Table 3 collects the access parameters of these listener lists.

Table 3 Access parameters of the listener lists

|  |  |  |  |
| --- | --- | --- | --- |
| **Event type** | **FBQ array** | **Hashmap** | **Search parameter** |
| Entity | v\_LGenBase\_entities [tEIdx]. entityListenerLists | v\_LGenBase\_behaviorTypes [bIdx]. entityListenerHashmapRefs [iIdx] | tEIdx |
| GeneralSource | v\_LGenBase\_entities [sEIdx]. entityListenerLists | v\_LGenBase\_behaviorTypes [bIdx]. generalSourceListenerHashmapRefs [iIdx] | sEIdx |
| BehaviorOfEntity | v\_LGenBase\_entities [tEIdx]. entityListenerLists | v\_LGenBase\_behaviorTypes [bIdx]. behaviorOfEntityHashMap | tEIdx |
| BehaviorOfSource | v\_LGenBase\_entities [sEIdx]. entityListenerLists | v\_LGenBase\_behaviorTypes [bIdx]. behaviorOfSourceHashMap | sEIdx |

*GeneralSourceFSM*

In case of GeneralSourceFSM event the listener indexes are stored in the v\_LGenBase\_entities[sEIdx].fsmCtxList[sFIdx].fsmOfGeneralSourceFSMListeners field. The FBQ contains the indexes of listeners listening to the GeneralSourceFSM events with sEIdx and sFIdx. The second and third element of the data field of the FBQ elements describe the behavior and input indexes of the required events. So during the event dispatch the list must be parsed.

*FSM*

Using FSM tables the listeners are defined in the FSM tables itself. There are different kinds of events the FSM rows can listen:

* general
* entity
* FSM

In case of general and entity events during the FSM table activation the LGenBase registers a table listener as a general or entity event listener, and the event dispatching goes through the standard event dispatching methods.

In case of FSM type event dispatching the listeners are described by the FSM tables. The FSM tables can be identified by the following way:

v\_LGenBase\_fsmTables[v\_LGenBase\_entities[tEIdx].fsmCtxList[tFIdx].tableIdx]

For each FSM table in the v\_LGenBase\_fsmTableRows2EventsHashMaps array there is a hashmap to enfasten to find the table rows listening to the specified event.

### Dispatch/post events

Using the f\_EPTF\_LGenBase\_dispatchEvent function the dispatched events are forwarded to the listeners immediately. The function returns after the execution of all the appropriate listeners as it's described in Figure 7.

Listener1

Listener2

LGenBase

Figure 7 Event dispatching using the f\_EPTF\_LGenBase\_dispatchEvent

The other way is to use the f\_EPTF\_LGenBase\_postEvent. If you call the f\_EPTF\_LGenBase\_postEvent from event dispatching, then the function returns without the execution of the listeners of the posted event, and they will be called after all the running event dispatch functions finished, as it's described on Figure 8.

Listener1

Listener2

LGenBase

Figure 8 Event dispatching using the f\_EPTF\_LGenBase\_postEvent

### FSM timers

The functions used in the FSM-s must not contain any blocking statements, even timers. Therefore LGenBase provides FSM timers. An FSM timer generates a special FSM timer event. So if an LGenBase user needs timers, the functions must be separated to two test steps, start an FSM timer, and set the test step listening to the timer event to the second part of the function.

## FSM tables in the traffic cases

Each traffic case contains at least one FSM table. The FSM tables of the same traffic case are called to "sibling FSM tables". First one is used by the LGenBase to manage the traffic, but physically there is no hierarchy among the associated FSM tables.

## DTE handling in LGenBase

DTE generated by test steps in FSM will be caught by LGenBase and the event "LGenBase: Dynamic\_test\_case\_error\_occured!" is reported for the entity. The FSM of the entity is deactivated, the entity is disabled. If the reported DTE event is not handled in the FSM, LGenBase reports a traffic error.

The error string of the DTE can be retrieved by the function f\_EPTF\_LGenBase\_get\_dte\_str (in integer pl\_eIdx, in integer pl\_fCtxIdx).

If DTE occurs, then all events are cancelled for the entity in the FSM, test steps that follow the step that caused the DTE are not executed, and the state remains the same as it was when the failing step was started.

LGenBase will not report the DTE event for failing test steps that are executed when the DTE event is processed in the FSM (that is the DTE handling part).

By default DTE handling is disabled.

## Prototypes

Prototypes describe a class of objects. They are:

* behavior types
* entity types
* traffic case types
* scenario types
* FSM tables

## Instances

The instances are:

* Entity groups are created in the f\_EPTF\_LGenBase\_createEntityGroup function.
* Entities are created as the part of their groups.
* Scenarios are created in the f\_EPTF\_LGenBase\_createScenario2EntityGroup function.
* Traffic cases are created as a part of the scenarios.

## Other structures

Events, event listeners, functions, test steps can be used as building blocks of configurations.

## Identifying prototypes

Prototypes can be identified by their names or their indexes. The indexes are the return values of their registration functions.

## Identifying instances

### Entity groups

Entity groups have an absolute index and unique names. They can be identified by any of them.

### Entities

Entities have an absolute index. This absolute index uniquely identifies them.

As a part of an entity group, they have a relative index in it. The entity group and the relative index of the entity identify it too.

Entities don’t have individual names.

### Scenarios

As the part of an entity group, scenarios have a relative index. The index of the entity group and the relative index of the scenario identify it. Similarly the name of the entity group and the name of the scenario type together identify the scenario.

### Traffic cases

Traffic cases have an absolute index.

Also, as the part of a scenario, the index of the scenario and entity group of the traffic case identify it. Similarly the name of the entity group, scenario type and traffic case type identify a traffic case.

## Describe configurations in configuration files

There are record types that can be used to create module parameters declaring objects in configuration files. Table 4 describes the objects and their declarator types.

Table 4 Objects and their declarator record type

|  |  |
| --- | --- |
| Object type | Record type |
| Behavior | EPTF\_LGenBase\_BehaviorDeclarator |
| Entity type | EPTF\_LGenBase\_EntityTypeDeclarator |
| Entity group | EPTF\_LGenBase\_EntityGrpDeclarator |
| Scenario | EPTF\_LGenBase\_TcMgmt\_ScenarioDeclarator2  EPTF\_LGenBase\_TcMgmt\_WeightedScenarioDeclarator |
| Traffic case | EPTF\_LGenBase\_TcMgmt\_tcTypeDeclarator2 |
| Compact and indexed FSM table | EPTF\_LGenBase\_CompactFsmTableDeclarator |

### Compact and indexed FSM tables in configuration files

The structure of the declarator of the compact and the indexed FSM tables are the same. The only difference is in the stepOrFunctionName field of the actionList field. In case of indexed FSM tables this field means the name of the registered test step. In case of compact FSM tables this field means the name of the test step function.

## Use cases of the traffic case management

Figure 9 describes a recommended usage of the test configuration and management surface of the LGenBase feature.

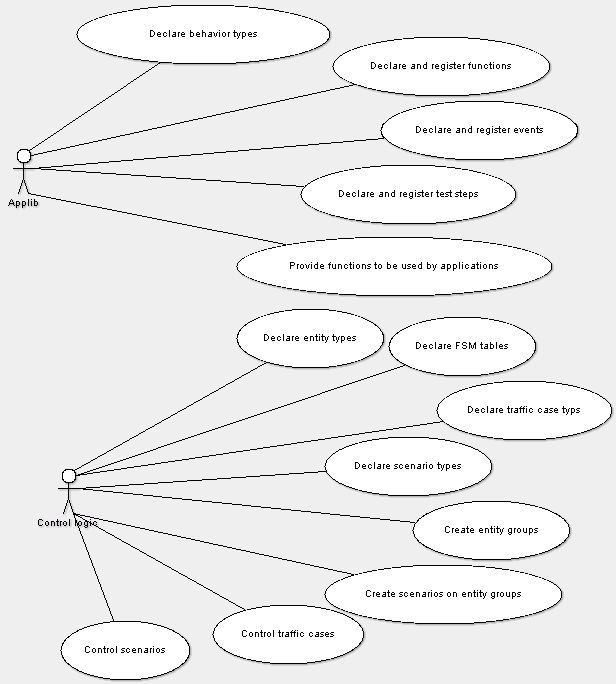


Figure 9 Uses cases of EPTF\_LGenBase\_CT

## Workflow of preparing the traffic case management

Figure 10 shows the synchronization points among use cases.

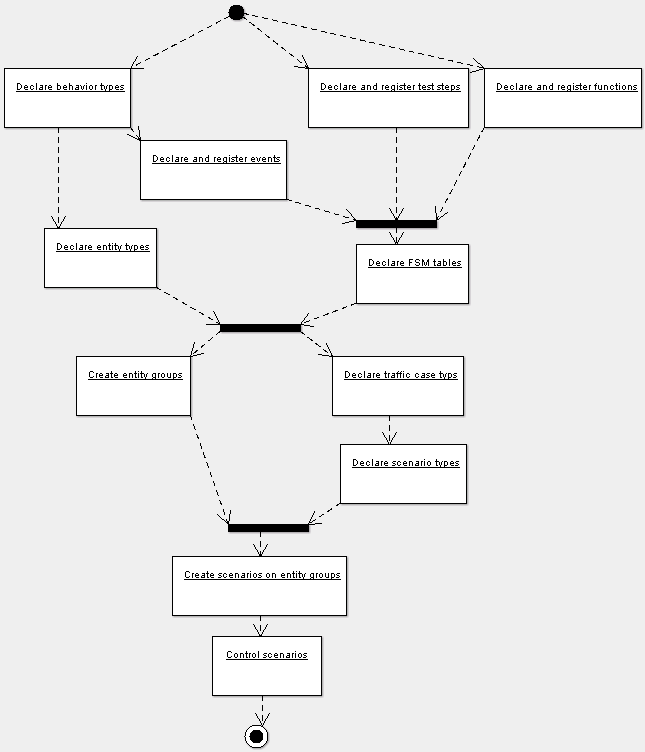


Figure 10 Typical workflow of EPTF\_LgenBase\_CT

## External Templates

### Overview

*External templates* are parameterized text modules where the parameters can be substituted with any values. Template parameters are tagged with special opening and closing patterns, which can be set by module parameters. Default patterns are ‘$(’ and ‘)’. An example for a simple template:

This is a sample template about fruits. The $(FRUIT) is my favorite fruit. Vegetables are healthy too, eat as much $(VEGETABLE) as you can. I especially like $(FRUIT) if it is fresh. Don't eat too much meat although $(MEAT) is very tasty.

LGenBase supports loading external templates from text files and storing them in memory. External templates can be referred by their unique name or index. LGenBase allows retrieving the substituted template content efficiently during runtime.

LGenBase allows a flexible way to define which external templates should be used for each traffic case. To do this first the required *template types* shall be declared. These are named types, which belong to a behavior type, and can be assigned to a stored external template. A *template set* is a list of such pairs, and it defines, which external templates are assigned to the template types in a traffic case.

A typical use case for external templates is the following:

* The Applibs declare the template types they can use;
* The application configures the list of external templates to load at initialization time;
* Application assigns loaded external templates to declared template types by traffic case declaration;
* Application uses the templates by substituting parameter values at runtime.

LGenBase supports runtime modification of template type declarations; it allows to define new parameters to template types and to register callback functions to each parameter. These features make possible for the applications to modify and extend the original template handling implementation of application libraries.

### How to use the external templates

1. Loading external templates

External templates can be loaded by creation of LGenBase component: module parameter tsp\_LGenBase\_extTemplLoadList is used to define the list of external templates to load.

External templates can also be loaded by f\_EPTF\_LGenBase\_loadExtTempl function.

This function loads the specified external template file into the memory and parses its content to find its parameters. The parsed content is stored in an internal database, which allows an efficient parameter substitution during runtime.

A <LF>─><CR><LF> conversion (or ‘unix2dos’ conversion) is often needed by using the external templates, since most protocol requires <CR><LF> characters by end of lines. Therefore LGenBase performs this conversion by loading the template files as default behavior. Although this automatic conversion can be switched off.

Example:

var integer myExtTemplIdx := f\_EPTF\_LGenBase\_loadExtTempl(“SIP\_INVITE1”, “/templates/sip\_invite.txt”, false);

This instruction loads the specified external template file without performing the <CR><LF> conversion.

1. Declaring template types

Applibs or applications can declare their template types by f\_EPTF\_LGenBase\_declareTemplateType function. A template type declaration can be later referred by its index or a <behavior name, template name> couple, called template key. By declaration two lists shall be given: the mandatory and the optional parameter names.

Example:

var integer myTemplTypeIdx := f\_EPTF\_LGenBase\_declareTemplateType(“SIP”, “INVITE”, {“callId”, “transactionId”}, {“contactAddr”});

This example declares a template type which has two mandatory parameters (callId and transactionId) and has an optional one (contactAddr).

1. Assigning templates to template types

It is possible to define which external template should be used for a declared template type. These template types are grouped to the so called template sets. New template set can be created by f\_EPTF\_LGenBase\_createTemplateSet function. Template sets can be referred by their index. A new assignment in a template set can be created by f\_EPTF\_LGenBase\_assignexternalTemplToTemplTypeByName function. The assign function checks whether the external template has all the mandatory parameters that the declared template type defines. If not, assignment is not possible. Optional parameters are not checked by assignment.

Example:

var integer myTsetIdx := f\_EPTF\_LGenBase\_createTemplateSet();  
var integer myTsetItemIdx := f\_EPTF\_LGenBase\_assignexternalTemplToTemplTypeByName(  
 v\_tset, {“SIP”, “INVITE”}, “SIP\_INVITE1”);

LGenBase also allows defining a template set in one step at the traffic case declaration. This can be done by f\_EPTF\_LGenBase\_declareScenarioType3 function. Calling this function creates a template set for each traffic case, and creates the specified template assignments. Before using this function it has to be assured that the required external templates are already loaded and the template types are declared.

Example:

var EPTF\_LGenBase\_TcMgmt\_TcOfScenario3 tcdef :=  
{ {  
 name := "register",  
 fsmName := "lGenRegFsm",  
 entityType := "etBubu0",  
 customEntitySucc := "",  
 templateSet :=   
 {  
 { “SIP”, “INVITE”, “SIP\_INVITE1” },  
 { “SIP”, “SUBSCRIBE”, “SIP\_SUBSCR3” },  
 { “XCAP”, “REQUEST”, “XCAP\_REQ” }  
 }  
 } }  
f\_EPTF\_LGenBase\_declareScenarioType3({“scen1”, … …, tcdef});

This example will create a template set with 3 templates for the traffic case named ‘register’.

Template set definition also can be done via configuration file very similar to the former scenario declaration. For details see scenario definition in section 2.17 .

1. Access functions

During runtime the content of the templates can be retrieved in three ways.

Dictionary based substitution

The f\_EPTF\_LGenBase\_getTemplateContent function waits an external template index and performs the template parameter substitution by the given dictionary, which is a list of <parameter name; value> pairs.

Example:

var charstring v\_content := “”;  
f\_EPTF\_LGenBase\_getTemplateContent(  
 myExtTemplIdx,   
 { {“FRUIT”,“apple”}, {“FISH”,“salmon”}, {“MEAT”,”pork”} },  
 v\_content  
);

In case of the sample template above the expected result is the following:

This is a sample template about fruits. The apple is my favorite fruit. Vegetables are healthy too, eat as much $(VEGETABLE) as you can. I especially like apple if it is fresh. Don't eat too much meat although pork is very tasty.

Those external template parameters which are not present in the dictionary, are not substituted (e.g. VEGETABLE). Those dictionary parameters which are not present in the external template are simple dropped (e.g. FISH).

Positional parameter based substitution

Positional parameter based substitution is more efficient than the dictionary based. This can be done by the f\_EPTF\_LGenBase\_getTemplateContentFast function. The position order is defined by the templatetype declaration.

Example:

var charstring v\_content := “”;  
f\_EPTF\_LGenBase\_getTemplateContentFast(  
 myTseIdx,   
 myTsetItemIdx,  
 { “pork”, “apple”, ”carrot” },  
 v\_content  
);

All parameters are defined by template type declaration shall be given.

In case of the sample template above the expected result is the following:

This is a sample template about fruits. The apple is my favorite fruit. Vegetables are healthy too, eat as much carrot as you can. I especially like apple if it is fresh. Don't eat too much meat although pork is very tasty.

Fixed position based substitution

Fixed position based substitution is the most efficient template substitution method. In order to use this feature the LGenBase component has to be initialized through the parameter pl\_templateGetterMode with fixedPosGetter chosen. An ordered list of parameter names has to be provided that will predetermine the order of keys in the lookup dictionary. Example initialization function call:

f\_EPTF\_LGenBase\_extTemplate\_init\_CT(

…,

pl\_templateGetterMode := {

fixedPosGetter := {

orderedMacroNameList := {

"FRUIT", "VEGETABLE", "MEAT", "SPICE"

}

}

});

Substitution can be done by calling the function f\_EPTF\_LGenBase\_applyTemplateContentGetter which is a function pointer set to f\_EPTF\_LGenBase\_extTemplFixedPos\_getTemplateContentin this mode.

Example:

var charstring v\_content := “”;

f\_EPTF\_LGenBase\_extTemplFixedPos\_getTemplateContent(

extTemplIdx,

{

{

paramName := "FRUIT",

paramValue := "APPLE"

},

{

paramName := "VEGETABLE",

paramValue := "CARROT"

},

{

paramName := "MEAT",

paramValue := "TENDERLOIN"

},

{

paramName := "SPICE",

paramValue := "PEPPER"

}

},

v\_content)

It is important that the order of the parameter key – value pairs must follow the order of the orderedMacroNameList provided in the initialization function. If the orderedMacroNameList contains duplicates or the template contains a key that is not provided in this list, the substitution will fail and an error will be thrown. The result of the substitution with the above call would be the following:

This is a sample template about FRUITs.

The APPLE is my favorite fruit.

Vegetables are healthy too, eat as much CARROT as you can.

I especially like APPLE if it is fresh.

Don't eat too much meat although TENDERLOIN is very tasty.

My favorite spice is PEPPER.

1. Find templates during runtime

LGenBase templates feature provides numerous functions to find a template during runtime. First the current template set shall be asked. For this purpose the f\_EPTF\_LGenBase\_templateSetIdxOfStep and f\_EPTF\_LGenBase\_templateSetIdxOfTc functions can be used.

External template index or template set item index can be gained by f\_EPTF\_LGenBase\_getExtTemplIdxFromSet or f\_EPTF\_LGenBase\_getTemplateItemIdxFromSet functions.

### Runtime modification of template types

It is possible that an application needs to modify a template type declaration, which is probably defined by an application library. For this purpose LGenBase makes possible to add new optional parameters to an existing template type declaration and to register callback functions to the template type.

#### Extending a template type with new parameters

New optional parameters can be added to an existing template type declaration by the function f\_EPTF\_LGenBase\_extendTemplateType. This function takes care that LGenBase’s internal database remains consistent, i.e. it updates all template set which contains the modified template type.

#### Template type callback functions

LGenBase allows defining callback functions to a template type. There are two kinds of callback functions supported: start/finish callbacks are called direct before/after the parameter substitution of the assigned external template. These callback functions can be set by f\_EPTF\_LGenBase\_setStartCallbackToTemplateType and f\_EPTF\_LGenBase\_setFinishCallbackToTemplateType functions.

For each parameter of a template type a parameter callback function can be registered too. This function is called every time when a value is substituted to a parameter and the result of the callback is substituted to the template instead of the original value. It can be used when the value of the parameter is not constant but shall be changed, e.g. if it is a counter which has to be incremented on every substitution.

Note, that it is possible to realize a stateful substitution with the help of so-called “context-arguments”, whose type is inout EPTF\_IntegerList. The context-argument will be initialized to empty before calling the start callback function. The value of the context arguments will be preserved by LGenBase between two consecutive argument-callback invocations of the same template replacement, that is, it can be used for storing data between consecutive callbacks. The context argument will be deleted after the call to the finish callback.

Template types with parameter callbacks can be declared by f\_EPTF\_LGenBase\_declareTemplateTypeWithCallbacks function. During runtime new parameter callbacks functions can be registered by f\_EPTF\_LGenBase\_extendTemplateTypeWithCallback function.

Note, that callbacks functions are only usable in case of position parameter-based substitution, e.g. by usage of f\_EPTF\_LGenBase\_getTemplateContentFast function. If dictionary-based substitution is applied, callback function won’t be called, since callbacks belong to the template type declarations and not to the external templates itself.

#### Template Parameter substitution callback function

LGenBase gives the opportunity to modify the behavior of the parameter substitution: if a parameter cannot be substituted normally, the previously registered callback functions are called in reversed order and if one of them can handle the parameter, then he can do it. These callbacks should be register with the function f\_EPTF\_LGenBase\_setTemplateMissingParameterCallback().

There is a default function registered into the system, if the tsp\_LGenBase\_enableDefaultArithmeticOperationsInExtTemplate is true (default : false), or if the f\_EPTF\_LGenBase\_init() is called with pl\_enableDefaultArithmeticOperationsInExtTemplate := true parameter. This function gives an arithmetic functionality to the template parameters: it handles addition, subtraction, multiplication and division on a parameter.

Example:

The $(USERNAME) is my favorite user. I especially like $(USERNAME+2) because he is more clever. I dont like $(USERNAME-2), he is less clever than me. I especially like $(USERNAME\*2) because he is the twin of the original. I don’t personally know $(USERNAME/2).

If normally $(USERNAME) would be substituted to [user\_0012@ericsson.hu](mailto:user_0012@ericsson.hu), then $(USERNAME+2) would be substituted to [user\_0014@ericsson.hu](mailto:user_0014@ericsson.hu), the $(USERNAME-2) would be [user\_0010@ericsson.hu](mailto:user_0010@ericsson.hu), the $(USERNAME\*2) would be [user\_0024@ericsson.hu](mailto:user_0024@ericsson.hu) and the $(USERNAME/2) would be [user\_0006@ericsson.hu](mailto:user_0006@ericsson.hu). No space is allowed between the original param name, the operator and the number.

## FSM table declaration

In R3 CLL version a new FSM declaration method was invented in the LGenBase. The EPTF\_LGenBase\_FsmTableDeclarator type allows users to declare only the necessary FSM elements. E.g. if there are no FSM timers used by the FSM, users don't have to write down an empty FSM timer declarator or omit value. On the other side, some missing elements can be found runtime only.

The new features of the FSM are implemented using this record type.

### EPTF variables and statistics in FSM-s

Using the f\_EPTF\_LGenBase\_declareFSMTable users can declare EPTF variables, StatMasure and StatHandler statistics in FSM declarations.

These FSM object have an attribute called "scope". The type of this attribute is:

**type** **enumerated** EPTF\_LGenBase\_FsmObjectScopeDeclarator {FSM, TC}

The scope FSM means that each FSM contexts having the specified FSM type will have their own FSM object after their activation.

The scope TC means that the traffic case joined to the specified FSM table will have a single FSM object, and all the FSM-s of the traffic case can access this common object. This means also that FSM objects with TC scope can be created only in FSM-s which are activated by the f\_EPTF\_LGenBase\_createTc2EntityGroup function.

The scope SiblingFSM means that each sibling FSM of the same entity will have the same object, but the FSMs of the different entities will have different objects.

A typical use case of statistics with scope "FSM" is to collect the number of successfully finished traffic executions of an entity. A typical use case of statistics with scope "TC" is to collect the number of successfully finished traffic executions in the traffic case.

SiblingFSM scope objects can be used in communication among sibling FSMs.

In case of StatHandler statistics, the case gets more difficult as the scope of the providers also affects the behavior of the statistic. In not trivial cases, e.g. when a StatHandler with FSM scope with siblingFSM scope provider, the effect is also not trivial. In the previous example the statistic will be calculated separately for all the FSMs in which it exists, for the provider with the given name and siblingFSM scope belonging to the FSMs declared as sibling FSMs in the same traffic case as the StatHandler. The reversed case is more common: when StatHandler has siblingFSM scope and the provider has FSM scope, all the FSM scoped providers in the FSMs of the given traffic case (aka siblingFSMs) will be calculated for one-one statistic for every entity. The providers can have different scopes under the same StatHandler, it is handled separately from each other, but the result is much harder to predict.

### Referring to FSM objects

In FSM declarations users can add parameters to the FSM steps. In the EPTF\_LGenBase\_FsmTableDeclarator users can refer to FSM objects by their names. E.g. the FSM step declaration {"LGenBase: Set variable",{varParams := {"var1ofFSM1",{floatVal := 7.1}}}} will declare a step registered by the name "LGenBase: Set variable", and the name "var1ofFSM1" refers to a variable declared with name "var1ofFSM1".

The step function in its step arguments will get the reference to the specified EPTF variable, and there are functions to retrieve the identifiers of the FSM object from those step arguments. See also 3.3.11.

### Naming of the FSM objects

#### The naming rule of the FSM objects with FSM scope

c\_EPTF\_LGenBase\_fsmVarPrefix&

c\_EPTF\_LGenBase\_fsmVarScopePrefixes[c\_EPTF\_LGenBase\_fsmObjectScopeFSM]&

The name of the entity&

tsp\_LGenBase\_nameSeparator&

FSM table name&

tsp\_LGenBase\_nameSeparator&

FSM index in the entity&

tsp\_LGenBase\_nameSeparator&

The declared name of the object

#### The naming rule of the FSM objects with TC scope

c\_EPTF\_LGenBase\_fsmVarPrefix&

c\_EPTF\_LGenBase\_fsmVarScopePrefixes[c\_EPTF\_LGenBase\_fsmObjectScopeTC]&

The unique name of the traffic case&

The declared name of the object

### Hookpoints in FSM tables

There can be cases when the users want to add extra actions to existing FSM tables, such as own statistics, logging, or anything else. This feature provides the later manipulation of an existing FSM table. Using this feature users can specify a point in an action list of an FSM table as a hookpoint, and insert a list of actions before specified hookpoints with the f\_EPTF\_LGenBase\_insertFsmSteps function.

To prevent the further modification of the EPTF\_LGenBase\_FsmTableDeclarator data type there is a specialFSM step with name c\_EPTF\_LGenBase\_stepName\_hookpoint. Each occurrence of this step defines a hookpoint. The hookpoints can be identified by names. The name can be described as a charstringValue step argument.

Here is a sample use of the hookpoint:

**var** EPTF\_LGenBase\_FsmTableDeclarator vl\_fsm2BeMerged := {

name := "c\_IMS\_SIP\_SigFSM\_Register",

fsmParams :={

{stateList := {"config", "idle", "initiated"}}

},

table := {

extendedTable := {

{

events2Listen := { events := {{ singleEvent := {

c\_SIP\_Behavior,

c\_IMS\_SIP\_eventName\_config,

fsm

}}}},

cellRow :={ statedCellRow := {{

inState := { state := "config"},

cell :={

{

{ c\_IMS\_SIP\_stepName\_SIPinit,omit},

{ c\_EPTF\_LGenBase\_stepName\_hookpoint,

{charstringValue := "abort"}

}

},

omit, "idle"

}

}

}

}

}

}

}

}

f\_EPTF\_LGenBase\_insertFsmSteps(

vl\_fsm2BeMerged,

"abort",

{

{ c\_EPTF\_LGenBase\_stepName\_log,

{charstringValue := "Abort called"}},

{ c\_EPTF\_LGenBase\_stepName\_setVerdict,

{verdict := {verdict := pass, reason := omit}}}

}

)

This code must result the following FSM:

**var** EPTF\_LGenBase\_FsmTableDeclarator vl\_fsm2BeMerged := {

name := "c\_IMS\_SIP\_SigFSM\_Register",

fsmParams :={

{stateList := {"config", "idle", "initiated"}}

},

table := {

extendedTable := {

{

events2Listen := { events := {{ singleEvent := {

c\_SIP\_Behavior,

c\_IMS\_SIP\_eventName\_config,

fsm

}}}},

cellRow :={ statedCellRow := {{

inState := { state := "config"},

cell :={

{

{ c\_IMS\_SIP\_stepName\_SIPinit,omit},

{ c\_EPTF\_LGenBase\_stepName\_log,

{charstringValue := "Abort called"}},

{ c\_EPTF\_LGenBase\_stepName\_setVerdict,

{verdict := {verdict := pass, reason := omit}}}

{ c\_EPTF\_LGenBase\_stepName\_hookpoint,

{charstringValue := "abort"}

}

},

omit, "idle"

}

}

}

}

}

}

}

}

As it can be seen from the sample, the hookpoints remain in the FSM table declaration even after the call of the f\_EPTF\_LGenBase\_insertFsmSteps function, which can be called later again.

## FSM tables in traffic cases

As it's described in 2.4, there is always at least one FSM associated to a traffic case. But there can be more FSM tables.

In the declaration of the traffic case there are traffic case parameters, the fsmList, and the siblingFsmList, which can be used to declare FSM tables for the traffic case.

Using the fsmList parameter, users only have to enlist the names of the associated FSM table types. These names can be used to identify the associated FSM tables in the traffic case, therefore they have to be unique. If more than one pieces of FSM tables the same type must be added to the traffic case, or simply the FSM tables must be identified by names different from their type names, use the siblingFsmList parameter. There each FSM table has a sibling name, and a type name. This case only the sibling names must be unique.

## Communication with FSM tables

There are API functions and predefined test steps to provide the communication with the FSM tables.

### API functions to dispatch/post events

The base of the communication is event dispatching/posting, as it's described in **Error! Reference source not found.**.

The f\_EPTF\_LGenBase\_dispatchEvent and f\_EPTF\_LGenBase\_postEvent functions dispatch/post events to FSM tables or event listeners.

### Communication among the FSM tables

The event dispatching functions described above can be called from any user code, either it's a part of an FSM test step, or it isn't.

If the LGenBase user dispatches/posts event from a step of an FSM, to fill up the source of the event is practical. There are functions, which fill up these data: f\_EPTF\_LGenBase\_dispatchEventFromStep and f\_EPTF\_LGenBase\_postEventFromStep.

This way of communication can be used only as function calls, since it requires the knowledge of several indexes, e.g. entity, FSM context index, etc., which can be counted only runtime.

There is also a simplified way where the communication of the FSM tables can be described in the declaration of the FSM table itself, using test steps and step arguments.

#### Declaring events in FSM tables

To make the FSM tables independent from their use, they can declare their own events. During the process of the FSM table declaration LGenBase declares these events automatically. The behavior of the events will be the name of the FSM table type, and the input names are the input names declared in the FSM table:

**const** EPTF\_LGenBase\_FsmTableDeclarator c\_LGenBaseDemo\_SubFSM :=

{

name := "SubFSM",

fsmParams := {

{declareEvents := {

useDefaultEvents := false,

eventList := {"start","finished"}}

}

...

There is a module parameter tsp\_LGenBase\_FSMDefaultEvents in the LGenBase, which is a record of charstring. If the useDefaultEvents is true in the declaration above, the enlisted events will be declared by the same way.

#### Use sibling FSM tables in traffic cases

In the traffic cases users can use more than one FSM tables, as it's described in **Error! Reference source not found.**. These FSM tables can cooperate with each other.

Let's take a sample use case: There are two sub-processes which can be managed by the same FSM types, let's name their type to "SubFSM".

There is an FSM, which manages the main flow of the test. Let's name it to "MainFSM". The sub-process to be managed by the SubFSM in this sample is to start a timer, and when it timed out, the FSM signs its finish to the MainFSM.

The FSM tables of the traffic case can be described this way:

v\_dummyInt := f\_EPTF\_LGenBase\_declareScenarioType3(

{"SC1",

{

{"TC1",

{

{siblingFsmList :=

{

{

siblingName := "MainFSM",

fsmType := "MainFSM"

},{

siblingName := "SubFSM1",

fsmType := "SubFSM"

},{

siblingName := "SubFSM2",

fsmType := "SubFSM"

}

}

...

#### Communication between sibling FSM tables

The MainFSM first starts the first sub-process dispatching an event to the first SubFSM.

{

c\_EPTF\_LGenBase\_stepName\_dispatchEventToSibling,

{eventToSibling := {

siblingName := "SubFSM1",

behaviorName := "SubFSM",

inputName := "start",

argVarName := omit

}}}

Then it waits for its finish:

{

//Waits for the finish of the first sub-FSM

events2Listen := {siblingEvents := {

singleEventFromSibling := {

siblingName := "SubFSM1",

iName := "finished",

eventType := fsm}}},

cellRow := { statedCellRow := {

Each SubFSM waits for the trigger event. When it arrives, the FSM stores the event into a stack to be able to reply to its source FSM, than starts a timer:

{

events2Listen := {events := {{

singleEvent := {"SubFSM","start",fsm}}}},

cellRow := { statedCellRow := {

{

inState := {state := "idle"},

cell :=

{{

{c\_EPTF\_LGenBase\_stepName\_pushEventToStack, omit},

{c\_EPTF\_LGenBase\_stepName\_timerStart,

{timerName := "subTimer"}

}

}, omit, omit}}}}

},

When the timer timed out, the SubFSM replies to the source FSM of the event on the top of the stack with an event declared in the FSM, than pops up the top element from the stack:

{

events2Listen := {events := {{ singleEvent := {

c\_EPTF\_LGenBase\_specialBName\_timerTimeout,

"subTimer",

fsm}}}},

cellRow := { statedCellRow := {

{ inState := {state := "idle"},

cell :=

{{

{

c\_EPTF\_LGenBase\_stepName\_dispatchReplyOwnEventToTopEventInStack,

{

eventOfFsm := {

inputName := "finished",

argVarName := omit}}

},

{c\_EPTF\_LGenBase\_stepName\_popEventFromStack, omit}

}, omit, omit}

}

}

}

}

When the MainFSM signed the finish of the first SubFSM, it starts the sub-process on the second SubFSM.

{

//Waits for the finish of the first sub-FSM

events2Listen := {siblingEvents := {

singleEventFromSibling := {

siblingName := "SubFSM1",

iName := "finished",

eventType := fsm}}},

cellRow := { statedCellRow := {

{

inState := {state := "idle"},

cell :=

{{

//Starts the second sub-FSM

{

c\_EPTF\_LGenBase\_stepName\_dispatchEventToSibling,

{eventToSibling := {

siblingName := "SubFSM2",

behaviorName := c\_demoBehaviorName,

inputName := "triggerSubFSM",

argVarName := omit

}}}

}, omit, omit}

}

}

}

},

# Functional Interface

Apart from this description a cross-linked reference guide for the TitanSim CLL Functions can be reached for on-line reading [4].

## Naming Conventions

All functions have the prefix f\_EPTF\_LGenBase\_.

## Using names

In order to make available configuring tests from configuration file, most of the traffic case management functions use the name properties to identify objects such as behaviors, entity types, functions, etc.

## Public Functions

### Initialization

Before using the EPTF LGenBase functions the

**function** f\_EPTF\_LGenBase\_init(  
 **in** **charstring** pl\_selfName,  
 **in** **integer** pl\_numEntities := 0,  
 **in** **charstring** pl\_entityNamePrefix,

**in** **EPTF\_LGenBase\_burstFunc** pl\_EPTF\_LGenBase\_burstFunc,  
 **in** **EPTF\_LGenBase\_extTemplLoadList** pl\_extTemplLoadList

in EPTF\_LGenBase\_TemplateGetterMode pl\_templateGetterMode := { defaultGetter:= {} },

in boolean pl\_extTemplRemoveLastNewLine := tsp\_LGenBase\_extTemplRemoveLastNewLine  
 )

function should be called. This initializes the EPTF LGenBase feature.

The parameter pl\_selfName defines the name of the component used to identify it via EPTF\_Base\_CT it extends.

The pl\_numEntities is an obsolete parameter. If you define it to more than 0, the LGenBase will create pl\_numEntities number of entities with undefined entity type and behavior. The automatic context management functions won’t be called to these entities and the traffic case management won’t handle these entities.

When the LGenBase creates the entities, it will use the pl\_entityNamePrefix to create their names.

With pl\_EPTF\_LGenBase\_burstFunc user can select the burst precalculating and postcalculating function. There are number of predefined burst calculating functions, but user can specify his own function too.

Setting the parameter to orig: pl\_EPTF\_LGenBase\_burstFunc:=orig the original EPTF type burst calculation function can be used. Setting the parameter to imst: pl\_EPTF\_LGenBase\_burstFunc:=imst the IMST type burst calculation function can be used. Setting the parameter to imst2: pl\_EPTF\_LGenBase\_burstFunc:=imst2 the IMST type burst calculation function can be used with a faster algorithm.

The parameter pl\_extTemplLoadList defines the list of external templates shall be loaded during the initialization of the component. The default list of external templates to load is defined in template parameter tsp\_LGenBase\_extTemplLoadList.

The pl\_templateGetterMode parameter defines the template substitution mode. The default mode will be the dictionary based substitution mode.

With the pl\_extTemplRemoveLastNewLine parameter the user can select if the last new line should be remove from the template or not.

### Declaring functions

In the traffic management there are several cases when the users can operate with user defined functions. E.g. functions to execute after the traffic case had finished. In the configuration they can be referred by names. Before using functions referred by their names, they have to be registered.

The f\_EPTF\_LGenBase\_declareFunction function has two parameters. pl\_name declares the name of the function. Later the functions can be referred by this name.

The pl\_fn parameter is an EPTF\_LGenBase\_RegisteredFunctions union. The chosen field describes what kind of functions the user wants to register.

The registered functions must mach one of the function types of registerable function types. These types associate registered functions with roles.

|  |  |
| --- | --- |
| Function type | Description |
| EPTF\_LGenBase\_‌customFinishCondition\_FT | At the end of each traffic it must be checked whether the traffic on the given entity or in the entity group shall be checked, or the traffic can be continued. (See also ‎2.4) Users can write functions that can decide it. |
| EPTF\_LGenBase\_‌evaluateSuccess4EntityAtEntityGroup2\_FT | These functions can be called at the end of each traffic of entities to calculate the next state of the specified entity. The return value describes the state. See also 2.5.  The pl\_tcIdx parameter specifies the absolute index of the traffic case, the pl\_eIdxInTc parameter specifies the relative index of the entity in the traffic case, the pl\_lastSuccess parameter specifies the result of the last traffic on the entity.  The return value of the function will be the state of the specified entity in the specified traffic case. |
| EPTF\_LGenBase\_‌customFinishFunction\_FT | In scenario declarations users can define what kind of actions must be executed when a group or entity finish condition became true. Users can write such functions, and they can be referred with their names. See also 2.4. |
| EPTF\_LGenBase\_‌TestStepFunction\_FT | If the user wanted to use compact FSM tables defined in configuration file, the test step functions used in the FSM tables must be registered. See also 2.17. |
| EPTF\_LGenBase\_‌NextStateCalc\_FT | If the user wanted to use compact or indexed FSM tables defined in configuration file, the next step calculation functions used in the FSM tables must be registered. See also 2.17. |

LGenBase provides predefined functions. See them in section 3.5.

### Declaring behavior types

Calling the f\_EPTF\_LGenBase\_declareBehaviorType function you can assign the declared entity type with a name. The pl\_maxCount parameter limits the number of maximal entities having this behavior. -1 means that the behavior supports unlimited number of entities.

The function described in the pl\_resetFn parameter will be called when the traffic case running on entities having the behavior will be reset.

The function described in the pl\_bindFn parameter will be called after creating the entity group. See Section 3.3.5

The function described in the pl\_unbindFn parameter will be called after deleting the entity group during the cleanup.

### Declaring entity types

f\_EPTF\_LGenBase\_declareEntityTypedeclareBehaviorType function joins behavior types together. It’s used to refer a group of behavior types easier.

### Creating entity groups

f\_EPTF\_LGenBase\_createEntityGroup function collects entities of the same type into a group. The name field of the parameter describes the identifier of the group, the eType field describes the entity type of the entities, and the eCount the required number of entities to be collected to the group.

The behaviors limit the maximum number of supported entities. If the eCount is greater than the available number of specified entities, than this function will create the available number of entities and returns their count.

When the function creates the entities, it calls the behavior context initialization function of all the behaviors of all the entities.

### FSM table declaration

#### FSM declaration with EPTF\_LGenBase\_FsmTableDeclarator

The EPTF\_LGenBase\_FsmTableDeclarator has the following main parts:

* fsmParams  
  Which contains the additional data such as
  + record of states in the FSM table
  + record of FSM timers
  + record of variables and statistics
* table  
  Which contains the description of the FSM table

*“table” field*

To make it possible to extend the type which declares the FSM table this field has a “records of unions” structure embedded in records of unions in several levels. Since there are types which have the same function in a bit different format or have less functionality, this section describes only the elements having the most functionality. All the other elements are subsets of these types.

*“extendedTable” element*

This field contains the record of rows of the FSM. Each row describes a record of events to be handled, and each column describes the answers to be taken in different states.

*“events2Listen” field*

This union describes the record of events to be handled in the specified row.

If there are different rows responding to the same event each row will be processed when the appropriate event arrives.

*“events” element*

Contains the record of events to be handled. If there are events in the list more than once, the associated rows will be processed only once when the appropriate event arrives.

*“singleEvent” element*

Describes a single event.

*“eventList” element*

Describes a list of single events.

*“eventRange” element*

Describes a range of events with the lower and higher boundaries.

*“catchall” element*

If this empty record element is chosen all the associated rows will be processed every time an event arrives regardless of there are rows associated to the appropriate event or not.

If there are more than one “catchall rows”, each of them will be processed every time an event arrives.

*“unhandled” element*

If this empty record element is chosen all the associated rows will be processed every time an event arrives which doesn’t have associated rows. The “catchall rows” doesn’t matter.

If there are more than one “unhandled event rows”, each of them will be processed every time an unhandled event arrives.

*“cellRow” field*

The statedCellRow field of this union contains the record of actions to be taken in different states when the associated event arrives. The elements of the record are called “columns”.

*“inState” field*

This field describes the state in which the enlisted actions must be executed.

*“state” element*

The name of the state in which the enlisted actions must be executed. The name of the state must exist in the record of FSM states in the fsmParams section.

Each state can be described only once in the row.

The order of the states is indifferent.

*“stateList” element*

The record of the names of the states in which the enlisted actions must be executed.

Each state can be described only once. There mustn’t be columns in the row associated to the states enlisted in this record.

The order of the states is indifferent.

*“anyUndefinedState” element*

The associated actions will be executed when an associated event arrives and there is no column associated to the present state of the FSM. There can be only one column in the record having this parameter.

*“cell” field*

This field describes the actions to be executed and the next state of the FSM after the execution.

### Declaring step context arguments

For each step enlisted in the cell of the FSM table users can declare “context arguments” which are passed for each step function in the refContext.fRefArgs field of the EPTF\_LGenBase\_TestStepArgs parameter.

There are several kinds of context argument declarations.However all of the declared step context arguments are converted to EPTF\_IntegerList (since the test step functions can receive only arguments of EPTF\_IntegerList type), don’t try to calculate their meaning. There are convenience functions to retrieve their content from the step arguments passed to the step functions, except the simplest ones.

Each case an argument refers to an element of the FSM (e.g. to a timer or an FSM variable) the name of the element must be valid.

#### stepContextArgs

Type: EPTF\_IntegerList

*Description:*

This argument type ha no special meaning, simply contains a list of integers.

#### timerName

Type: charstring

*Description:*

Describes an FSM timer. The passed EPTF\_IntegerList contains one integer, which is the index of the timer in the list of FSM timer declaration.

#### varNames

Type: EPTF\_CharstringList

*Description:*

Describes a list of FSM variables.

*Data access function:*

f\_EPTF\_LGenBase\_fsmVarIdListFromStep

*Return data:*

The list of the identifiers of the EPTF variables referred by the names.

#### varParams

Type: EPTF\_LGenBase\_FsmVarParams

*Description:*

Describes an FSM variable and an optional EPTF\_Var\_DirectContent data.

*Data access function:*

f\_EPTF\_LGenBase\_fsmVarParamsFromStep

#### statMeasParams

Type: EPTF\_LGenBase\_FsmStatMeasParams

*Description:*

Describes an FSM statMeasure statistics and an optional EPTF\_Var\_DirectContent data.

*Data access function:*

f\_EPTF\_LGenBase\_fsmStatMeasParamsFromStep

#### statName

Type: charstring

*Description:*

Describes an FSM statHandler statistics.

*Data access function:*

f\_EPTF\_LGenBase\_fsmStatisticNameOfStep

#### statMeasName

Type: charstring

*Description:*

Describes an FSM statMeasure statistics.

*Data access function:*

f\_EPTF\_LGenBase\_fsmStatMeasIdFromStep

#### eventToSibling

Type: EPTF\_LGenBase\_EventToSibling

*Description:*

Describes an event to be sent to a sibling FSM

*Data access function:*

f\_EPTF\_LGenBase\_fsmEventToSiblingFromStep

#### replyEvent

Type: EPTF\_LGenBase\_ReplyEvent

*Description:*

Describes a reply event to event sent from a sibling FSM

*Data access function:*

f\_EPTF\_LGenBase\_fsmReplyEventFromStep

#### eventToTC

Type: EPTF\_LGenBase\_EventToTC

*Description:*

Describes an event to be sent to the FSMs in the same traffic case.

*Data access function:*

f\_EPTF\_LGenBase\_fsmEventToTCFromStep

#### eventOfFsmToSibling

Type: EPTF\_LGenBase\_EventOfFsmToSibling

*Description:*

Describes an event defined previously in the FSM to be sent to a sibling FSM.

*Data access function:*

f\_EPTF\_LGenBase\_fsmEventOfFsmToSiblingFromStep

#### eventOfFsm

Type: EPTF\_LGenBase\_EventOfFsm

*Description:*

Describes an event defined previously in the FSM.

*Data access function:*

f\_EPTF\_LGenBase\_eventOfFsmFromStep

### Declaring FSM tables with f\_EPTF\_LGenBase\_declareFSMTables

There is also another function f\_EPTF\_LGenBase\_declareFSMTables that can be used to create a list of EPTF\_LGenBase\_FsmTableDeclarator. This function has one EPTF\_LGenBase\_FsmTableDeclaratorList type argument and it returns an EPTF\_IntegerList which contains the indexes of the created FSM tables.

### Obsolete versions of declaring FSM tables

The f\_EPTF\_LGenBase\_declareCompactFsmTable creates a compact FSM table.

The f\_EPTF\_LGenBase\_declareIndexedCompactFsmTable creates an indexed compact FSM table.

The f\_EPTF\_LGenBase\_TcMgmt\_declareCompactFsmTables creates a set of compact FSM tables from an EPTF\_LGenBase\_TcMgmt\_CompactFsmTableDeclaratorList record.

The f\_EPTF\_LGenBase\_TcMgmt\_declareIndexedFsmTables creates a set of indexed compact FSM tables from an EPTF\_LGenBase\_TcMgmt\_IndexedFsmTableDeclaratorList record.

### FSM variable accessing functions

The f\_EPTF\_LGenBase\_varNameOfFSMVar function retrieves the EPTF Variable name of the specified FSM variable if the FSM variable has FSM scope.

The f\_EPTF\_LGenBase\_varNameOfTCVar function retrieves the EPTF Variable ID of the specified FSM variable if the FSM variable has TC scope.

Using the return values of these functions the f\_EPTF\_Var\_getId function retrieves the EPTF Variable ID.

Since the naming of each FSM object (Variable, StatMeasure and StatHandler statistic) is the same, these functions can be used to get the name of these FSM objects too.

The f\_EPTF\_LGenBase\_fsmVarIdOfFSMStat function retrieves the EPTF Variable ID of the EPTF Var of the specified FSM StatHandler statistic. The statistic must be declared with FSM scope.

The f\_EPTF\_LGenBase\_fsmVarIdOfTCStat function retrieves the EPTF Variable ID of the EPTF Var of the specified FSM StatHandler statistic. The statistic must be declared with TC scope.

The f\_EPTF\_LGenBase\_fsmVarIdOfStat function retrieves the EPTF Variable ID of the EPTF Var of the specified FSM StatHandler statistic. The scope of the declared statistic must be the same as the scope specified in the parameter list of the function.

The f\_EPTF\_LGenBase\_fsmStatMeasIdOfFSM function retrieves the StatMeasure ID of the specified FSM StatMeasure statistic. The scope of the declared statistic must be FSM.

The f\_EPTF\_LGenBase\_fsmStatMeasIdOfTC function retrieves the StatMeasure ID of the specified FSM StatMeasure statistic. The scope of the declared statistic must be TC.

### Step-based FSM object accessing functions

LGenBase provides functions to retrieve data from the test step arguments. Users can use the FSM step parameterization to pass arguments to their own FSM steps in FSM declaration.

#### FSM variable accessing functions using FSM step arguments

Table 5 FSM variable accessing functions using FSM step arguments

|  |  |  |
| --- | --- | --- |
| Function | Description | Accepted step argument types |
| f\_EPTF\_LGenBase\_‌fsmVarIdFromStep | Retrieves the EPTF Variable ID of the specified FSM variable. | varNames  varParams |
| f\_EPTF\_LGenBase\_‌fsmVarIdListFromStep | Retrieves the list of the EPTF Variable IDs of the specified FSM variables. | varNames |
| f\_EPTF\_LGenBase\_ fsmVarInitValueFromStep | Retrieves the EPTF Variable ID and the EPTF\_Var\_DirectContent specified at the declaration of the FSM variable from the step arguments. | varNames  varParams |
| f\_EPTF\_LGenBase\_‌ fsmVarParamsFromStep | Retrieves the EPTF\_Var\_DirectContent from the step arguments. | varParams |

#### FSM StatMeasure statistics accessing functions using FSM step arguments

Table 6 FSM StatMeasure statistics accessing functions using FSM step arguments

|  |  |  |
| --- | --- | --- |
| Function | Description | Accepted step argument types |
| f\_EPTF\_LGenBase\_‌ fsmStatMeasIdFromStep | Retrieves the StatMeasure ID of the specified FSM StatMeasure statistic. | statMeasName  statMeasParams |
| f\_EPTF\_LGenBase\_‌ fsmStatMeasParamsFromStep | Retrieves the StatMeasure ID and the specified EPTF\_Var\_DirectContent from the step arguments. | statMeasParams |

The f\_EPTF\_LGenBase\_fsmStatMeasIdFromStep function retrieves the StatMeasure ID of the specified FSM StatMeasure statistic. The scope of the declared statistic is automatically retrieved from the arguments.

#### FSM StatHandler statistics accessing functions using FSM step arguments

Table 7 FSM StatMeasure statistics accessing functions using FSM step arguments

|  |  |  |
| --- | --- | --- |
| Function | Description | Accepted step argument types |
| f\_EPTF\_LGenBase\_‌ fsmStatisticNameOfStep | Retrieves the name of the specified FSM StatHandler statistic. | statName |

### Declaring traffic case types

A traffic case type declaration basically declares that the traffic case having the specified type on what kind of entity which FSM will execute. The f\_EPTF\_LGenBase\_declareTrafficCaseType function uses the EPTF\_LGenBase\_TrafficCaseTypeDeclarator record to specify a traffic case type. The mandatory fields of that record are the following:

* The name field describes the name which identifies the traffic case type.
* The fsmName field specifies the associated FSM table declared earlier. See Section 0.
* The entityType member specifies the entity type on which the FSM will be executed. About entity type declaration see section 3.3.4.

There are also optional parameters of the traffic case types.

#### Optional parameters of the EPTF\_LGenBase\_TrafficCaseTypeDeclarator

At the end of each traffic the state of entity must be calculated (see also 2.5). Users can write their own functions, or they can choose one from the predefined ones. That can be referred in the customEntitySucc field.

### Declaring scenario types

A scenario is a collection of traffic cases.

In the f\_EPTF\_LGenBase\_declareScenarioType3 function you can count the the traffic cases playing a part in the scenario, and define their parameters.

function uses the EPTF\_LGenBase\_ScenarioTypeDeclarator record to specify a scenario type.

### The EPTF\_LGenBase\_ScenarioTypeDeclarator record

This record type has the following parts:

* The name field identifies the scenario type. It must be unique.
* The tcList field encounters the traffic cases playing apart.
* The scParamsList field declares additional parameters of the scenario.

At least one traffic case must be specified in a scenario.

A traffic case of a scenario can be declared by the tcList field.

The scParamsList field is a list of parameters. The order of the declaration of the parameters is indifferent. None of the parameters are mandatory.

Table 8 Scenario type parameters

|  |  |  |
| --- | --- | --- |
| Parameter | Description | Default value |
| phaseListName | The name of the phase list of the scenario. See also 3.3.21 | "" |
| weightedScData | Parameters to specify the expected load of a weighted scenario | omit |
| enabled | Only enabled weighted scenarios can be started | true |
| phaseFinishConditions | The finish conditions of the phases of the scenario. See also 3.3.21 | empty list |
| phaseStateChangeActions | The actions to be executed at the changes of the phases of the scenario. See also 3.3.21 | empty list |
| trafficType | The type (originating/terminating) of the scenario. See also 2.9 | originating |

### Traffic cases of scenarios

In the scenario declaration users can describe the traffic cases in the tcList field. This field has the type EPTF\_LGenBase\_TcOfScenarioDeclaratorList. This is the list ofEPTF\_LGenBase\_TcOfScenarioDeclarator records. The name field identifies the traffic case of the scenario. It must be unique in the scenario. The tcParamsList field is a list of parameters. The order of the declaration of the parameters is indifferent. None of the parameters are mandatory. Table 9 describes the elements of it.

Table 9 The elements of the EPTF\_LGenBase\_tcParamsList

|  |  |  |
| --- | --- | --- |
| Parameter | Description | Default value |
| startDelay | The traffic case starts later after the call of the starter function with the specified value. It has effect only in normal traffic cases. | 0.0 |
| enableEntitiesAtStart | The enabled states of the entities of the traffic case at the start of the scenario or traffic case. | false |
| enabledAtStart | Whether to start the traffic case when the scenario started or not. | false |
| target | The expected target load. In weighted scenarios the trafficWeight must be specified, in normal scenarios the cpsToReach. | dummy |
| scheduler | The load generation calculation function. | "" |
| entitySelection | The method of selecting the next entity during traffic launch. | round\_robin |
| ranges | Parameter ranges of the traffic case. | empty list |
| params | Named custom parameters. | empty list |
| entityFinishConditions | Conditions of the entities to be finished. | empty list |
| entityFinishActions | Actions to be executed when an entity became finished. | empty list |
| trafficStartFinish | Conditions to stop the launch of the traffic of the traffic case, and the actions to be executed when the conditions became true. | empty list |
| trafficFinishedActions | Actions to be executed when the launch of the traffic finished, and all the started entities reported their execution finish | empty list |
| templateSet | External template list of the traffic case | empty list |
| tcTypeName | The name of the traffic case type assigned to the traffic case | the name of the traffic case |
| restoreTCAtStartTC | When set to false, restoreTC is not invoked when traffic case is started with startTC in ExecCtrl or from the CLI. | true |

If the enableEntities parameter is true, all the entities will be available for the traffic management after creating the appropriate traffic case.

If the enable parameter is true, the enabled state of the traffic case will be enabled after creation, and if the pl\_autoStart parameter of the f\_EPTF\_LGenBase\_createScenario2EntityGroup function is true, it will start traffic on the traffic case.

When a condition described in the trafficStartFinish parameter became true, the LGenBase stops executing the traffic case and executes the actions described in the actions field of the condition became true. If there were actions defined in the anythingFinished condition, these actions are executed too, but only first time.

When a condition described in the entityFinishConditions becomes true, the actions described in the entityFinishActions will be executed to the given entity.

The finish actions of the traffic cases of a scenario must not point out from the scenario i.e. they can start or stop traffic cases existing in the scenario.

### Associate scenarios with entity group

Before starting a traffic case, you must associate the scenario containing the traffic case with an entity group. The entity group name – scenario name – traffic case name triplet identifies a traffic case specimen during the traffic case management.

If the enableEntities member of the parameter of the f\_EPTF\_LGenBase\_declareScenarioType2 function was true (see Section 3.3.13), all the entities of the entity group become available for the traffic case from start, or else they become unavailable.

If the enable member of the parameter of the f\_EPTF\_LGenBase\_declareScenarioType2 function was true in the description of a traffic case, the traffic case starts automatically after the association.

### Start traffic case

If the traffic case didn’t start automatically, and the traffic case is the part of a normal scenario, users can start them with the f\_EPTF\_LGenBase\_startTrafficCase function. The index of the traffic case can be retrieved by the f\_EPTF\_LGenBase\_trafficCaseId function.

### Stop traffic case

The f\_EPTF\_LGenBase\_stopTrafficCase function stops executing the traffic case and dispatches a c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_stopTC event using the t\_stopTcOfFsm template. See also 2.6.

### Abort traffic case

The f\_EPTF\_LGenBase\_abortTrafficCase function stops executing the traffic case and dispatches a c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_abortTC event using the t\_abortTcOfFsm template. See also 2.6.

### Handling of Burst Calculation methods

There are various burst calculation methods can be assigned to the traffic cases.

There are two types of burst calculation methods pre defined and user defined. Pre defined methods can be for example “orig”, “cs”, “imst”, “imst2”, “poisson”. Users can define their own methods too.

The types of burst calculation can be assigned to each of the traffic cases or to each weighted scenario.

A global burst calculation method can be assigned to all of the traffic cases and weighted scenarios with the module parameter tsp\_LGenBase\_BustCalcMethod.

If a user wants to set his own burst calc method, then he has to define a new one with its name and its function references. The function f\_EPTF\_LGenBase\_addBurstCalcFunction saves this new method to a database. The function f\_EPTF\_LGenBase\_getBurstCalcFunction retrieves the function references of a burst calc method from the database. A method can be accessed by its index that can get with the function f\_EPTF\_LGenBase\_getBurstCalcFunctionId.

If the user already defined a new method or wants to use a predefined method, he has to assign this method to a traffic case or a weighted scenario. The functions f\_EPTF\_LGenBase\_setBurstCalcAlgByName and f\_EPTF\_LGenBase\_setBurstCalcAlgByIdx can be used to set a burst calc method to a traffic case or a weighted scenario with the help of its database index or its name.

If the user wants to know what kind of burst calculation method has already been assigned to a traffic case, he can use the functions f\_EPTF\_LGenBase\_getBurstCalcAlgByName and f\_EPTF\_LGenBase\_getBurstCalcAlgByIdx. The two functions retrieve the index of the burst calculation method

The burst calculation method can be assigned to a traffic case or to a scenario when the traffic case is declared. In EPTF\_LGenBase\_ScenarioTypeDeclarator in the traffic case list or in the weighted scenario the parameter scheduler can be used to set the correct values.

### Managing Phases

EPTF\_LGenBase\_CT can provide phase support. Phases can be declared and handled with various functions. The different phases and their states can be asked with other functions.

#### Handling phase declarators

The list of phase declarators can be saved with the function f\_EPTF\_LGenBase\_PhaseList\_Declarators\_store. One phase declarators can be saved with the function f\_EPTF\_LGenBase\_PhaseList\_store. These two functions store the phase declarators to a database. The function f\_EPTF\_LGenBase\_PhaseList\_get\_byIndex and f\_EPTF\_LGenBase\_PhaseList\_get\_byName can be used to retrieve a phase declarator from the database with the help of its database index or its name.

If the phase declarators have been set, the phases can be used.

The phase actions - which occur in the phase state change - and the phase finish conditions can be declared in the scenario declarator.

#### Using of phases

The following functions are defined for starting and stopping the phases manually:

The function f\_EPTF\_LGenBase\_startPhase can be used to start a specified phase. This function starts the phase in the specified scenario if there is no running phase. This function sets the state of the started phase to *running* and runs the phase actions defined in the scenario configuration parameters. If there is an already running phase, then the function does nothing. With the help of this function a reference to a handler function can be added. When the phase stops the handler function will be called.

The function f\_EPTF\_LGenBase\_stopPhase can be used to stop a phase if the user does not want to wait its automatic stopping. This function sets the state to *stopping* and runs the actions defined in this state.

The function f\_EPTF\_LGenBase\_skipPhase can be used to set the state of the phase to *skipping*. After setting the state of the phase, this function runs the actions defined in this state.

The function f\_EPTF\_LGenBase\_resetPhase can be used to set the state of the phase to *idle*. The scenario then restores to its starting phase.

#### Getting various phase information

The following functions can be used to get phase information:

The function f\_EPTF\_LGenBase\_getActualPhase can be used to get the actual phase, where the scenario stays and the state of this phase.

To get the phase actions declared in the scenario, the function f\_EPTF\_LGenBase\_getPhaseActions can be used. If user wants to ask the actions in the actual phase, the specific function f\_EPTF\_LGenBase\_getActualPhaseActions can be used too.

To get the phase finish conditions declared in the scenario in an arbitrary phase, the function f\_EPTF\_LGenBase\_getPhaseFinishConditions can be used. The specified function f\_EPTF\_LGenBase\_getActualPhaseFinishConditions can be used, if user wants to know the phase finish conditions in the actual phase, where the scenario stays.

### Target CPS Change Handling

The LGenBase provides ways to handle target CPS change in traffic cases and scenarios using callback functions.

#### Registration of CPS change callback functions for traffic cases and scenarios

Callback functions registered by f\_EPTF\_LGenBase\_registerCPSChangedCallback\_TC

are going to be called when the target CPS of the given traffic case changes.

Similar functionality for weighted scenarios is accessible by the function

f\_EPTF\_LGenBase\_registerCPSChangedCallback\_SC

Functions registered are called when the target CPS of the scenario changes.

#### Deregistration of CPS change callback functions

Deregistration of CPS change callback functions for traffic cases and scenarios can be done with the functions

f\_EPTF\_LGenBase\_deregisterCPSChangedCallback\_TC

f\_EPTF\_LGenBase\_deregisterCPSChangedCallback\_SC

The previously registered function will be deregistered and they are not going to be called when the CPS changes.

### Limited Execution

The LGenBase provides a way to limit the execution by reducing the number of entities in the EntitiyGroup. To do this, you should call the function f\_EPTF\_LGenBase\_limitExecution with the entity group identifier and the limit size. Warning : this function resets the statistics.

The function f\_EPTF\_LGenBase\_unlimitExecution restores the original status of the given EntityGroup. Warning : this function resets the statistics.

## Predefined test steps

In order to support a more comfortable programming, LGenBase registers some useful test steps.

For all the predefined test steps LGenBase have name constants, step index constants, and it registers all the functions that implement these test steps with the defined name constants.

The step name constants have the prefix c\_EPTF\_LGenBase\_stepName\_, and the step index constants have the prefix.

### Steps administering the start of an entity

For so-called “terminating traffic cases” the entities are not started by the LGenBase built-in scheduler. The start of the entities is triggered by user-defined means, such as reacting to incoming messages (e.g., by including such incoming events in the FSM table). See 2.8. In this case the LGenBase must be informed about the start of the entity to note the necessary information.

#### Start the entity

*Function*

f\_EPTF\_LGenBase\_step\_startEntity

*Step name*

"LGenBase: StepFunction\_startEntity"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_startEntity

*Description*

This function does the same as LGenBase does when it starts an entity. Finally LGenBase dispatches the regular c\_EPTF\_LGenBase\_inputName\_testMgmt\_startTC event for the entity.

This step is useful if users want to use the same FSM both when the LGenBase starts the entities (originating traffic cases), and when the FSM reacts to external events (terminating traffic cases).

#### Record the start of entity

*Function*

f\_EPTF\_LGenBase\_step\_recordEntityStart

*Function name*

"LGenBase: StepFunction\_recordEntityStart"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_recordEntityStart

*Step index*

c\_EPTF\_LGenBase\_stepIdx\_recordEntityStart

*Description*

This function does the same as LGenBase does when it starts an entity except dispatching the c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_startTC LGenBase FSM event.

This step notifies LGenBase that the entity have become busy. This step *must* be invoked whenever a terminating FSM kicks into execution due to an incoming event.

### Steps reporting finish of execution of an entity

As it’s described in 2.4, the entity must report the result of its execution. These steps provide a comfortable way to do it.

Warning: these steps can be called only if the entity had been started (the LGenBase had dispatched the c\_EPTF\_LGenBase\_inputName\_testMgmt\_startTC event for the entity, or its start had been administered by executing the f\_EPTF\_LGenBase\_step\_startEntity or f\_EPTF\_LGenBase\_step\_recordEntityStart steps). Otherwise it causes assertion.

#### Reporting successful execution

*Function:*

f\_EPTF\_LGenBase\_step\_trafficSuccess

*Step name:*

"LGenBase: StepFunction\_trafficSuccess"

*Step name constant:*

c\_EPTF\_LGenBase\_stepName\_trafficSuccess

*Description:*

Reports that the execution finished succesfully by dispatching a c\_EPTF\_LGenBase\_stepIdx\_trafficSuccess LGenBase generic event.

This step notifies LGenBase that the entity have become idle. This step *must* be invoked whenever the entity becomes idle and it considers that the current execution of the traffic case was *successful*.

#### Reporting failed execution

*Function*

f\_EPTF\_LGenBase\_step\_trafficFailed

*Step name*

"LGenBase: StepFunction\_trafficFailed"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_trafficFailed

*Description*

Sends a c\_EPTF\_LGenBase\_stepIdx\_trafficFailed LGenBase generic event reply.

This step notifies LGenBase that the entity have become idle. This step *must* be invoked whenever the entity becomes idle and it considers that the current execution of the traffic case was unsuccessful, i.e., *it has been failed* due to inappropriate answers from the SUT. NOTE: This notification *must not* be used for message sending failure due to transport errors (see section 3.4.2.3 for this case), nor for timeout situations (i.e., no timely answer from the SUT, see section 3.4.2.4 for this case).

#### Reporting erroneous execution

*Function*

f\_EPTF\_LGenBase\_step\_trafficError

*Step name*

"LGenBase: StepFunction\_trafficError"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_trafficError

*Description*

Sends a c\_EPTF\_LGenBase\_stepIdx\_trafficFailed LGenBase generic event reply. It’s useful at the end of a traffic finished with failure.

This step notifies LGenBase that the entity have become idle. This step must be invoked whenever the entity becomes idle and it considers that the current execution of the traffic case was unsuccessful due to message sending (i.e., transport) errors. NOTE: This notification must not be used for unsuccessful cases due to inappropriate answers from the SUT. (see section 3.4.2.2 for this case), nor for timeout situations (i.e., no timely answer from the SUT, see section 3.4.2.4 for this case).

#### Reporting timed out execution

*Function*

f\_EPTF\_LGenBase\_step\_trafficTimeout

*Step name*

"LGenBase: StepFunction\_trafficTimeout"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_trafficTimeout

*Description*

Sends a c\_EPTF\_LGenBase\_stepIdx\_trafficTimeout LGenBase generic event reply.

This step notifies LGenBase that the entity have become idle. This step must be invoked whenever the entity becomes idle and it considers that the current execution of the traffic case was unsuccessful due to timeout (i.e., no timely answer from the SUT). NOTE: This notification must not be used for message sending failure due to transport errors (see section 3.4.2.3 for this case), nor for failures due to inappropriate answers from the SUT (see section 3.4.2.2 for this case).

### Steps replying to stop/abort/reset LGenBase events

As it’s described in 2.6.1, LGenBase notifies the entities of the running traffic cases when the traffic case has been stopped or aborted. The entities must report when they processed these messages.

LGenBase provides steps to do it.

#### Reply to the stopTC event

*Function*

f\_EPTF\_LGenBase\_step\_entityStopped

*Step name*

"LGenBase: StepFunction\_entityStopped"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_entityStopped

*Description*

Sends a c\_EPTF\_LGenBase\_stepIdx\_entityStopped LGenBase generic event reply. It’s useful at the end of handling a c\_EPTF\_LGenBase\_inputName\_testMgmt\_stopTC FSM event.

NOTE: If the entity has been busy, then beyond invoking this test step the entity must also report one of the return codes of section 3.4.2!

#### Reply to the abortTC event

*Function*

f\_EPTF\_LGenBase\_step\_entityAborted

*Function name*

"LGenBase: StepFunction\_entityAborted"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_entityAborted

*Description*

Sends a c\_EPTF\_LGenBase\_stepIdx\_entityAborted LGenBase generic event reply. It’s useful at the end of handling a c\_EPTF\_LGenBase\_inputName\_testMgmt\_abortTC FSM event.

NOTE: If the entity has been busy, then beyond invoking this test step the entity must also report one of the return codes of section 3.4.2!

### Timer handling steps

FSMs can contain timers (see also 2.10.22.10.8). LGenBase provide steps to handle them.

#### Start timer

*Function*

f\_EPTF\_LGenBase\_step\_timerStart

*Function name*

"LGenBase: StepFunction\_timerStart"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_timerStart

*Description*

Starts an FSM timer.

In the FSM declaration the context argument of the step must contain the name of the timer:

{  
 stepOrFunctionName :=c\_EPTF\_LGenBase\_stepName\_timerStart,  
 contextArgs := {timerName := "responseTimer"}  
}

#### Start/Restart timer

*Function*

f\_EPTF\_LGenBase\_step\_timerStartOrRestart

*Function name*

"LGenBase: StepFunction\_timerStartOrRestart"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_timerStartOrRestart

*Description*

Only difference from f\_EPTF\_LGenBase\_step\_timerStart, that it does not log a warning when applied to an already running timer.

#### Cancel timer

*Function*

f\_EPTF\_LGenBase\_step\_timerCancel

*Function name*

"LGenBase: StepFunction\_timerCancel"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_timerCancel

*Description*

Cancels an FSM timer.

In the FSM declaration the context argument of the step must contain the name of the timer:

{  
 stepOrFunctionName:=c\_EPTF\_LGenBase\_stepName\_timerCancel,  
 contextArgs := {timerName := "responseTimer"}  
}

Executing this step with a not running timer causes warning in the log.

#### Cancel timer if running

*Function*

f\_EPTF\_LGenBase\_step\_timerCancelIfRunning

*Function name*

"LGenBase: StepFunction\_timerCancelIfRunning"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_timerCancelIfRunning

*Description*

Only difference from f\_EPTF\_LGenBase\_step\_timerCancel, that it does not log a warning when applied to a not running timer.

#### Cancel all timers

*Function*

f\_EPTF\_LGenBase\_step\_cancelAllTimers

*Function name*

"LGenBase: StepFunction\_cancelAllTimers"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_cancelAllTimers

*Description*

Cancels all the running timers of the FSM.

### FSM variable handling steps

LGenBase provides FSM steps to manipulate the FSM variables (see 2.21.1).

#### Set content

*Function*

f\_EPTF\_LGenBase\_step\_fsmVarSet

*Function name*

"LGenBase: Set variable"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarSet

*Description*

In the FSM declaration the context argument of the step must contain a varParams argument. The step sets the content of the FSM variable named in varName to the value given in the paramValue:

{  
 "LGenBase: Set variable",  
 {  
 varParams := {  
 varName := "var1ofFSM1",  
 paramValue := {floatVal := 3.27}  
 }  
 }  
}

The content type of the paramValue and the variable can be any type implemented in the EPTF\_Var\_DirectContent.

#### Increment variable

*Function*

f\_EPTF\_LGenBase\_step\_fsmVarInc

*Function name*

"LGenBase: Increment variable"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarInc

*Description*

In the FSM declaration the context argument of the step must contain a varParams argument. The step increments the content of the FSM variable named in varName to the value given in the paramValue. If the paramValue is not present, the value will be 1 or 1.0, depending on the type of the EPTF\_Var\_DirectContent. The content type of the paramValue and the variable can be intVal or floatVal.

{  
 "LGenBase: Increment variable",  
 {  
 varParams := {  
 varName := "var1ofFSM1",  
 paramValue := {floatVal := 3.27}  
 }  
 }  
}

#### Decrement variable

*Function*

f\_EPTF\_LGenBase\_step\_fsmVarDec

*Function name*

"LGenBase: Decrement variable"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarDec

*Description*

In the FSM declaration the context argument of the step must contain a varParams argument. The step decrements the content of the FSM variable named in varName to the value given in the paramValue. If the paramValue is not present, the value will be 1 or 1.0, depending on the type of the EPTF\_Var\_DirectContent. The content type of the paramValue and the variable can be intVal or floatVal.

{  
 "LGenBase: Decrement variable",  
 {  
 varParams := {  
 varName := "var1ofFSM1",  
 paramValue := {floatVal := 3.27}  
 }  
 }  
}

#### Reset variable

*Function*

c\_EPTF\_LGenBase\_stepName\_fsmVarReset

*Function name*

"LGenBase: Reset variable"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarDec

*Description*

Resets the content of the FSM variable to the value specified at its declaration. In the FSM declaration the context argument of the step must contain a varNames argument. The step processes only the first name. Any other specified variable names are ignored, but all of them must be valid.

{  
 "LGenBase: Reset variable",  
 {  
 varNames := {"var1ofFSM1"}  
 }  
}

#### Adding variables together

*Function*

c\_EPTF\_LGenBase\_stepName\_fsmVarAdd

*Function name*

"LGenBase: Add variables"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarAdd

*Description*

Adds the content of the specified FSM variables and stores the sum in the first variable. In the FSM declaration the context argument of the step must contain a varNames argument. There must be at least two variable names present in the argument. If there are more than two names, LGenBase adds all the specified variables.

{  
 "LGenBase: Add variables",  
 {  
 varNames := {"var1ofFSM1", "var2ofFSM1", "var3ofFSM1"}  
 }  
}

The content type of the variables can be intVal or floatVal, and they must have the same type.

#### Subtract variables

*Function*

c\_EPTF\_LGenBase\_stepName\_fsmVarSub

*Function name*

"LGenBase: Subtract variables"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmVarSub

*Description*

In the FSM declaration the context argument of the step must contain a varNames argument. The step subtracts the content of the second FSM variable from the content of the first FSM variable. There must be at least two variable names present in the argument. The step processes only the first two variables. Any other specified variables are ignored, but all the given names must be valid.

{  
 "LGenBase: Subtract variables",  
 {  
 varNames := {"var1ofFSM1", "var2ofFSM1"}  
 }  
}

The content type of the variables can be intVal or floatVal, and they must have the same type.

#### Sample of using the FSM variable manipulation steps

f\_EPTF\_LGenBase\_declareFSMTable(

{

name := "tc\_LGenBase\_Test\_fsmVariables2\_FSM",

fsmParams := {

{stateList := {"idle"}},

{statMeasStatList := {

{"chrono1","",chrono,FSM},

{"chrono2","",chrono,TC}

}},

{varList := {

{ name := "var1ofFSM1",

initValue := {floatVal := 1.0},

scope := FSM },

{ name := "TCvar2ofFSM1",

initValue := {floatVal := 2.0},

scope := TC },

{ name := "TCvar1ofFSM1",

initValue := {floatVal := 3.0},

scope := TC },

{ name := "var2ofFSM1",

initValue := {floatVal := 4.0},

scope := FSM }

}},

{statisticList :={

{ "statMax1",

{{"var1ofFSM1"},{"TCvar1ofFSM1"}},

c\_EPTF\_StatHandler\_Method\_Max,

{floatVal := 0.0},FSM},

{ "statMax2",

{{"var1ofFSM1"},{"TCvar1ofFSM1"}},

c\_EPTF\_StatHandler\_Method\_Max,

{floatVal := 0.0},TC},

{ "statMin1",

{{"var1ofFSM1"}},

c\_EPTF\_StatHandler\_Method\_Min,

{floatVal := 0.0},TC},

{ "statMin2",

{{"TCvar1ofFSM1"},{"var1ofFSM1"}},

c\_EPTF\_StatHandler\_Method\_Min,

{floatVal := 0.0},TC}

}}

},

table := {

classicTable := {

{eventToListen := {"b1","input3",fsm},

cellRow := {

{//state[0]==idle

{//Cell

{"LGenBase: Set variable",{varParams := {"var1ofFSM1",{floatVal := 7.1}}}},

{"LGenBase: Increment variable",{varParams := {"var1ofFSM1", omit}}},

{c\_EPTF\_LGenBase\_stepName\_fsmStartChrono,{statMeasName := "chrono1"}}

},

omit, omit

}

}

},

{eventToListen := {"b1","input4",fsm},

cellRow := {

{

{

{ "LGenBase: Decrement variable",

{varParams :={"var1ofFSM1",{floatVal:= 1.8}}}},

{ c\_EPTF\_LGenBase\_stepName\_fsmStopChrono,

{statMeasName := "chrono1"}}

},

omit, omit

}

}

},

{eventToListen := {"b1","input6",fsm},

cellRow := {

{

{

{"LGenBase: Set variable",{varParams := {"var1ofFSM1",{floatVal := 7.1}}}},

{"LGenBase: Set variable",{varParams := {"var2ofFSM1",{floatVal := 17.4}}}},

{"LGenBase: Set variable",{varParams := {"TCvar1ofFSM1",{floatVal := 3.27}}}},

{"LGenBase: Set variable",{varParams := {"TCvar2ofFSM1",{floatVal := 45.27}}}},

{"LGenBase: Add variables",{varNames := {"var1ofFSM1","var2ofFSM1","TCvar1ofFSM1"}}

}

},

omit, omit

}

}

},

{eventToListen := {"b1","input7",fsm},

cellRow := {

//state[0]==idle

{

{

{"LGenBase: Set variable",{varParams := {"var1ofFSM1",{floatVal := 7.1}}}},

{"LGenBase: Set variable",{varParams := {"var2ofFSM1",{floatVal := 17.4}}}},

{"LGenBase: Set variable",{varParams := {"TCvar2ofFSM1",{floatVal := 3.27}}}},

{"LGenBase: Subtract variables",

{varNames := {"var1ofFSM1","var2ofFSM1","TCvar2ofFSM1"}}

}

},

omit, omit

}

}

},

{eventToListen := {"b1","input8",fsm},

cellRow := {

//state[0]==idle

{

{

{"LGenBase: Reset variable",{varNames := {"var1ofFSM1"}}}

},

omit, omit

}

}

}

}

}

}

);

### FSM chronometer manipulation steps

LGenBase provides FSM steps to manipulate the FSM StatMeasure statistics with chronometer type. (About the chronometer statistics see [7].)

For all these steps the type of the step context argument must be statMeasName. The content must be the name of the chronometer to be manipulated. E.g.:

{  
 c\_EPTF\_LGenBase\_stepName\_fsmStartChrono,  
 {statMeasName := "chrono1"}  
}

#### Start chronometer

*Function*

f\_EPTF\_LGenBase\_step\_fsmStartChrono

*Function name*

"LGenBase: Start chronometer"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmStartChrono

*Description*

Starts the chronometer.

#### Stop chronometer

*Function*

f\_EPTF\_LGenBase\_step\_fsmStopChrono

*Function name*

"LGenBase: Stop chronometer"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmStopChrono

*Description*

Stops the chronometer.

#### Reset chronometer

*Function*

f\_EPTF\_LGenBase\_step\_fsmResetChrono

*Function name*

"LGenBase: Reset chronometer"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_fsmResetChrono

*Description*

Resets the chronometer.

### Test management steps

#### Reporting the finish of the test

*Function*

f\_EPTF\_LGenBase\_step\_testFinished

*Function name*

"LGenBase: StepFunction\_testFinished"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_testFinished

*Description*

TitanSim can’t decide when should be the test finished. This step reports a c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_testFinished generic event, indicating that the test should be finished. (This is equivalent with the pressing of the “exit” button on the GUI of the TitanSim Generic Application.)

### Sibling FSM event management steps

#### The following steps realize the event flow between FSMs in a testcase. We can dispatch (immediate handling) or post (handling after step ended) an event and reply to a sent event between sibling FSMs.

#### Dispatch an event to a sibling FSM

*Function*

f\_EPTF\_LGenBase\_step\_dispatchEventToSibling

*Function name*

"LGenBase: StepFunction\_dispatchEventToSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchEventToSibling

*Description*

Dispatches an event to the given sibling FSM with the given arguments.

#### Post an event to a sibling FSM

*Function*

f\_EPTF\_LGenBase\_step\_postEventToSibling

*Function name*

"LGenBase: StepFunction\_postEventToSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postEventToSibling

*Description*

Posts an event to the given sibling FSM with the given arguments.

#### Reply via dispatch to the source of the event previously sent

*Function*

f\_EPTF\_LGenBase\_step\_dispatchReplyFromSibling

*Function name*

"LGenBase: StepFunction\_dispatchReplyFromSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchReplyFromSibling

*Description*

Dispatches a reply event to the source FSM of the previously sent event with the given arguments.

#### Reply via post to the source of the event previously sent

*Function*

f\_EPTF\_LGenBase\_step\_postReplyFromSibling

*Function name*

"LGenBase: StepFunction\_postReplyFromSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postReplyFromSibling

*Description*

Posts a reply event to the source FSM of the previously sent event with the given arguments.

#### Dispatch a previously declared event to a sibling FSM

*Function*

f\_EPTF\_LGenBase\_step\_dispatchEventOfFsmToSibling

*Function name*

"LGenBase: StepFunction\_dispatchEventOfFsmToSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchEventOfFsmToSibling

*Description*

Dispatches a previously declared event to the given sibling FSM with the given arguments.

#### Post a previously declared event to a sibling FSM

*Function*

f\_EPTF\_LGenBase\_step\_postEventOfFsmToSibling

*Function name*

"LGenBase: StepFunction\_postEventOfFsmToSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postEventOfFsmToSibling

*Description*

Posts a previously declared event to the given sibling FSM with the given arguments.

#### Reply with a predefined event via dispatch to the source of the event previously sent

*Function*

f\_EPTF\_LGenBase\_step\_dispatchReplyWithEventOfFsmFromSibling

*Function name*

"LGenBase: StepFunction\_dispatchReplyWithEventOfFsmFromSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchReplyWithEventOfFsmFromSibling

*Description*

Dispatches a predefined reply event to the source FSM of the previously sent event with the given arguments.

#### Reply with a predefined event via post to the source of the event previously sent

*Function*

f\_EPTF\_LGenBase\_step\_postReplyWithEventOfFsmFromSibling

*Function name*

"LGenBase: StepFunction\_postReplyWithEventOfFsmFromSibling"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postReplyWithEventOfFsmFromSibling

*Description*

Posts a predefined reply event to the source FSM of the previously sent event with the given arguments.

#### Send an event via dispatch to all the FSMs of the same traffic case

*Function*

f\_EPTF\_LGenBase\_step\_dispatchEventToTC

*Function name*

"LGenBase: StepFunction\_dispatchEventToTC"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchEventToTC

*Description*

Dispatches an event to all the FSMs in the same traffic case with the given arguments.

#### Send an event via post to all the FSMs of the same traffic case

*Function*

f\_EPTF\_LGenBase\_step\_postEventToTC

*Function name*

"LGenBase: StepFunction\_postEventToTC"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postEventToTC

*Description*

Posts an event to all the FSMs in the same traffic case with the given arguments.

### Event stack handling steps

These steps handle an event stack in which incoming events can be stored and at a later time can be replied to.

#### Push an event on top of the stack.

*Function*

f\_EPTF\_LGenBase\_step\_pushEventToStack

*Function name*

"LGenBase: StepFunction\_pushEventToStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_pushEventToStack

*Description*

Push the incoming event on top of the stack.

#### Pop an event from the top of the stack.

*Function*

f\_EPTF\_LGenBase\_step\_popEventFromStack

*Function name*

"LGenBase: StepFunction\_popEventFromStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_popEventFromStack

*Description*

Pop the top event from the stack (it will be only erased, not given back in any kind of way, will be never available again). A push event must be called within the same FSM before a pop in order not to get an error message.

#### Reply to the source of the event on the top of the stack via dispatch.

*Function*

f\_EPTF\_LGenBase\_step\_dispatchReplyToTopEventInStack

*Function name*

"LGenBase: StepFunction\_dispatchReplyToTopEventInStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchReplyToTopEventInStack

*Description*

Reply to the source of the event on the top of the stack. It does not pop the top event from the stack, so it will not be erased from stack, the source info will be copied into target field of the event to be sent and a dispatch will happen. At least one push event must be called within the same FSM before a reply in order not to get an error message.

#### Reply to the source of the event on the top of the stack via post.

*Function*

f\_EPTF\_LGenBase\_step\_postReplyToTopEventInStack

*Function name*

"LGenBase: StepFunction\_postReplyToTopEventInStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postReplyToTopEventInStack

*Description*

Reply to the source of the event on the top of the stack. It does not pop the top event from the stack, so it will not be erased from stack, the source info will be copied into target field of the event to be sent and a post will happen.

#### Reply with a predefined event to the source of the event on the top of the stack via dispatch.

*Function*

f\_EPTF\_LGenBase\_step\_dispatchReplyOwnEventToTopEventInStack

*Function name*

"LGenBase: StepFunction\_dispatchReplyOwnEventToTopEventInStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_dispatchReplyOwnEventToTopEventInStack

*Description*

Reply with a predefined event to the source of the event on the top of the stack. It does not pop the top event from the stack, so it will not be erased from stack, the source info will be copied into target field of the event to be sent and a dispatch will be executed.

#### Reply with a predefined event to the source of the event on the top of the stack via post.

*Function*

f\_EPTF\_LGenBase\_step\_postReplyOwnEventToTopEventInStack

*Function name*

"LGenBase: StepFunction\_postReplyOwnEventToTopEventInStack"

*Step name constant*

c\_EPTF\_LGenBase\_stepName\_postReplyOwnEventToTopEventInStack

*Description*

Reply with a predefined event to the source of the event on the top of the stack. It does not pop the top event from the stack, so it will not be erased from stack, the source info will be copied into target field of the event to be sent and a post will be executed.

### FSM DebugLight

When there is a fault in the traffic flow, several times it’d be useful to follow the FSM table execution. A useful tool to do it is the FSM DebugLight.

There is a module parameter tsp\_EPTF\_LGenBaseDebugLightList to enlist the names of the FSM tables to be followed by the FSM DebugLight. When the DebugLight is enabled on an FSM table, the LGenBase writes a simple, well-formatted log statement each time if

* an event is reported to the FSM table
* there is no listener to a disdpatched event, i.e. there is an unhandled event in any state
* there is no listener to a disdpatched event, i.e. there is an unhandled event in the actual state of the FSM table.

The log statement always begins with “DebugLight:FSM:” followed by the name of the FSM table, and “:Entity”, followed by the entity index between # characters. Using these markers you can filter to the relevant log statements.

## Predefined functions

### Entity success decision functions

These functions can be used to calculate the state of the entities at the end of a traffic (see 2.5).

#### EPTF\_LGenBase\_evaluateSuccess4EntityAtEntityGroup2\_FT type

* LGenBaseEntitySucc\_successSucNoFail returns success only if there had been no failed traffic on the entity.
* LGenBaseEntitySucc\_successIfLastSuccess returns the result of the last traffic on the entity.
* LGenBaseEntitySucc\_successIfOnceSuccess returns success if there had been at least one successful traffic on the entity.

#### EPTF\_LGenBase\_evaluateSuccess4EntityAtEntityGroup2\_FT type

* LGenBaseEntitySucc\_successSucNoFail returns success only if there had been no failed traffic on the entity.
* LGenBaseEntitySucc\_successIfLastSuccess returns the result of the last traffic on the entity.
* LGenBaseEntitySucc\_successIfOnceSuccess returns success if there had been at least one successful traffic on the entity.

## LGenBase events

The following sections contain the descriptions of events registered by the LGenBase.

The titles of the topics contain the name of the TTCN-3 constants representing the input index of the events. According to the TitanSim Coding Guideline these constants have the prefix c\_EPTF\_LGenBase\_inputIdx\_. The **charstring** constants containing the names of the events have the prefix c\_EPTF\_LGenBase\_inputName\_. The end of the names of the name constants are the same as of the name of the index constants of the same event.

### Traffic management events

These events are dispatched by the LGenBase. User code should only listen to and answer them. Don’t dispatch or post any of it!

See also 2.4 and 2.6.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_startTC

Name: "LGenBase: Start\_the\_traffic\_case!"

Indicates that the entity has been started.

In the case when the LGenBase triggers the execution of the entities, or the FSM executes the f\_EPTF\_LGenBase\_step\_recordEntityStart step, the row listening to this event is the entry point of the FSM.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_stopTC

Name: "LGenBase: "LGenBase: Stop\_the\_traffic\_case!"

Indicates that the running traffic case related to the FSM of the entity has been stopped. When the entity finished processing the stop of the traffic case it must report the c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityStopped event or execute the f\_EPTF\_LGenBase\_step\_entityStopped step.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_abortTC

Name: "LGenBase: Aborts\_traffic\_case!"

Indicates that the running traffic case related to the FSM of the entity has been aborted. When the entity finished processing the abortion of the traffic case it must report the c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityAborted event or execute the f\_EPTF\_LGenBase\_step\_entityAborted step.

### Entity execution result report events

When an entity finished an execution, it must report it to the LGenBase. According to the result of the execution it should report different events.

For the sake of comfortable use LGenBase provides FSM steps to report these events.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcSuccess

Name: "Successfully finished the traffic case!"

Reporting step: f\_EPTF\_LGenBase\_step\_trafficSuccess

Reports successful execution.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcFail

Name: "Failed to finish succesfully the traffic case!"

Reporting step: f\_EPTF\_LGenBase\_step\_trafficFailed

Reports failed execution due to inappropriate answers from the SUT. See also section3.4.2.2.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcError

Name: "LGenBase: The traffic case on the entity finished with error!"

Reporting step: f\_EPTF\_LGenBase\_step\_trafficError

Reports that the execution failed because of a sending (i.e., transport) error. See also section 3.4.2.3.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_finishedTcTimeout

Name: "LGenBase: The traffic case on the entity timed out!"

Reporting step: f\_EPTF\_LGenBase\_step\_trafficTimeout

Reports that the execution failed because of timeout. See also section 3.4.2.4

### Answering traffic management events

When a running traffic case has been stopped LGenBase dispatches events about it to the entities. The entities must process it and answer when they processed.

For the sake of comfortable use LGenBase provides FSM steps to report these events.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityStopped

Name: "LGenBase: Entity\_has\_been\_stopped."

Reporting step: f\_EPTF\_LGenBase\_step\_entityStopped

Reports that the entity has been stopped after the LGenBase event c\_EPTF\_LGenBase\_inputName\_testMgmt\_stopTC.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_entityAborted

Name: "LGenBase: Entity\_has\_been\_aborted."

Reporting step: f\_EPTF\_LGenBase\_step\_entityAborted

Reports that the entity has been stopped after the LGenBase event c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_abortTC.

### Traffic state notification events

On some points of the traffic management LGenBase dispatches general events which can be used as synchronization points.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_trafficCaseStarted

Name: "LGenBase: Traffic\_case\_started."

A traffic case has been started.

The reported step argument contains the index of the appropriate traffic case on the 0th position.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_trafficCaseStopped

Name: "LGenBase: Traffic\_case\_has\_been\_stopped!"

A traffic case has been stopped.

The reported step argument contains the index of the appropriate traffic case on the 0th position.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_launchedTrafficFinished

Name: "LGenBase: All\_launched\_traffic\_finished!"

The last entity execution in the traffic case has finished. After this message the LGenBase won’t launch new traffic until the traffic case is restarted.

The reported step argument contains the index of the appropriate traffic case on the 0th position.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_scenarioStarted

Name: "LGenBase: Scenario\_started."

A scenario has been started.

The reported step argument contains the index of the entity group of the appropriate scenario on the 0th position, the relative index of the scenario on the 1st position, and the absolute index of the scenario on the 2nd position.

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_scenarioFinished

Name: "LGenBase: All\_traffic\_cases\_of\_scenario\_finished!"

All the traffic cases of the scenario have finished launching.

The reported step argument contains the index of the entity group of the appropriate scenario on the 0th position, the relative index of the scenario on the 1st position, and the absolute index of the scenario on the 2nd position.

### Other events

#### c\_EPTF\_LGenBase\_inputIdx\_testMgmt\_testFinished

Name: "User: Test\_finished."

Reporting step: f\_EPTF\_LGenBase\_step\_testFinished

TitanSim can’t decide when should be the test finished. This event indicates that the test should be finished. (This is equivalent with the pressing of the “exit” button on the GUI of the TitanSim Generic Application.)

To report this event at the finish of a traffic case, users can declare an EPTF\_LGenBase\_TcMgmt\_Action\_TestFinished testFinished as an action of trafficFinishedActions, or an action of an EPTF\_LGenBase\_TcFinishCondition in an EPTF\_LGenBase\_tcParams of a traffic case declaration.

#### c\_EPTF\_LGenBase\_inputIdx\_timerTimeout

Name: "LGenBase: Timer timeout"

Internal event, do NOT dispatch it, and do NOT register listener to this event.

### LGenBase events and their responses

The following table contains a brief summary of the events dispatched by the LGenBase and their responses.

Table 10 Table of traffic management LGenBase FSM events

|  |  |  |  |
| --- | --- | --- | --- |
| LGenBase generated event inputs | Description | Response event input(s) | Description |
| startTC | The entity has to start a traffic. | finishedTcSuccess | The test traffic finished successfully. |
| finishedTcFail | The test traffic finished with failure. |
| finishedTcError | The test traffic finished with error. |
| finishedTcTimeout | The test traffic timed out |
| stopTC | The traffic case had been stopped before matching a finish condition. | entityStopped | The entity has gone through its stopping process. |
| abortTC | The traffic case had been aborted before matching a finish condition. | entityAborted | The entity has gone through its aborting process. |

## Built-in statistics

The EPTF\_LGenBaseStats\_CT extens the EPTF\_LGenBase\_CT and provides variable support to the properties of the traffic cases and scenarios. Table 11 describes the variable names of the traffic cases.

Table 11 Variables of traffic cases

|  |  |  |
| --- | --- | --- |
| Variable name constant | Variable name | Description |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfStarts | NrOfStarts | Nr. of started traffics. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfSuccesses | NrOfSuccesses | Nr. of successfully finished traffics. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNnrOfFails | NnrOfFails | Nr. of traffics finished with failure. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfNotFinishedEntities | NrOfNotFinishedEntities | Nr. of entities that has not finished jet. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfAllEntities | NrOfAllEntities | Nr. of entities in the entity group of the traffic case. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfRunningEntities | NrOfRunningEntities | Nr. of entities that has started but their traffic has not finished jet. |
| c\_EPTF\_LGenBaseStats\_nameOfTcNrOfAvailableEntities | NrOfAvailableEntities | Nr. of available entities. |
| c\_EPTF\_LGenBaseStats\_nameOfTcState | State | The state of the traffic case. |
| c\_EPTF\_LGenBaseStats\_nameOfTcStateName | StateName | The state of the traffic case in readable text format. |
| c\_EPTF\_LGenBaseStats\_nameOfTcUserData | UserData | charstring data that can be set by the user. |
| c\_EPTF\_LGenBaseStats\_nameOfTcSentMessages | SentMessages | Nr. of started traffics. |
| c\_EPTF\_LGenBaseStats\_nameOfTcReceivedAnswers | ReceivedAnswers | Nr. of traffics that had finished. |
| c\_EPTF\_LGenBaseStats\_nameOfTcRangeLoops | RangeLoops | Nr. of range loops. |
| c\_EPTF\_LGenBaseStats\_nameOfTcCpsToReach | CpsToReach | Required CPS of a non-weighted traffic case. |
| c\_EPTF\_LGenBaseStats\_nameOfTcLastCps | LastCps | Nr. of last started traffics devided by the time period between the two last traffic case timer events. |
| c\_EPTF\_LGenBaseStats\_nameOfTcName | TCName | The name of the traffic case |
| c\_EPTF\_LGenBaseStats\_nameOfTcUniqueName | UniqueName | The unique name of the traffic case |
| c\_EPTF\_LGenBaseStats\_nameOfTcEnabled | Enabled | The enabled state of the traffic case. |
| c\_EPTF\_LGenBaseStats\_nameOfTcMinAvailable | MinAvailableEntities | The minimal nr. of the available entities. |
| c\_EPTF\_LGenBaseStats\_nameOfTcMaxRunning | MaxRunningEntities | The maximal nr. of the running entities. |
| c\_EPTF\_LGenBaseStats\_nameOfTcMaxBusy | MaxBusyEntities | The maximal nr. of the busy (running and finished) entities. |
| c\_EPTF\_LGenBaseStats\_nameOfTcWeight | Weight | The weight of the weighted traffic case. |

Table 12 describes the variable names of the scenarios.

Table 12 Variables of scenarios

|  |  |  |
| --- | --- | --- |
| Variable name constant | Variable name | Description |
| c\_EPTF\_LGenBaseStats\_nameOfScName | Name | The name of the scenario |
| c\_EPTF\_LGenBaseStats\_nameOfScEnabled | Enabled | The enabled state of the scenario |
| c\_EPTF\_LGenBaseStats\_nameOfScState | State | The state of the scenario |
| c\_EPTF\_LGenBaseStats\_nameOfScUserData | UserData |  |
| c\_EPTF\_LGenBaseStats\_nameOfScStateName | StateName | The state of the scenario in readable text form |
| c\_EPTF\_LGenBaseStats\_nameOfScLockCPS | LockCPS | charstring data that can be set by the user. |

LGenBaseStats creates its variables automatically when a scenario created.

## GUI support

The EPTF\_LGenBaseStatsUI\_CT extends the EPTF\_LGenBaseStats\_CT and provides GUI to display and/or manipulate properties of the traffic cases and scenarios.

The data is grouped by scenarios. Each scenario block has a header, where there are the properties of the scenario, and a list of traffic case data, each traffic case in one row. Figure 11shows a sample GUI.

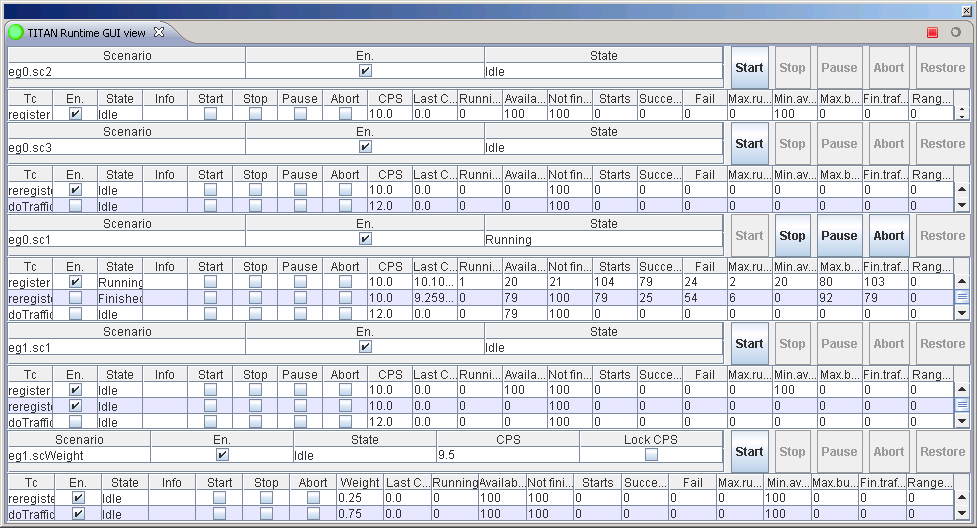


Figure 11 A sample GUI

The content of the traffic case rows is configurable through the tsp\_LGenBaseStatsUI\_columnDescriptorList module parameter. Users can describe which columns to display, the order and caption of the columns, and can write protect the properties if they were writeable.

To display data the f\_EPTF\_LGenBaseStatsUI\_prepareGUI function must be called. Then LGenBaseStatsUI creates the widgets under the widget described by the pl\_parentWidgetId parameter of the function.

## Generalization

Figure 12 shows the extension hierarchy of LGenBase feature.

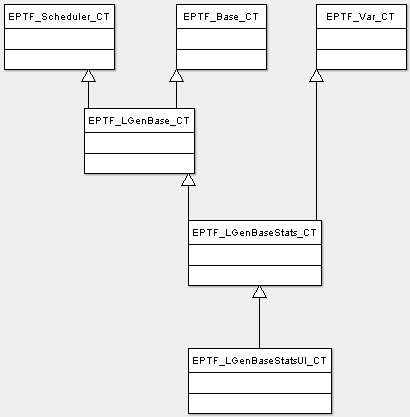


Figure 12 Extends hierarchy of the LGenBase

## Public functions for EPTF LGenBase

The public API of the LGenBase can be found in the ApiDoc of the EPTF Core Load Library.

# References

1. ETSI ES 201 873-1 v3.2.1 (2007-02)  
   The Testing and Test Control Notation version 3. [Part 1: Core Language](http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/03.02.01_60/es_20187301v030201p.pdf)
2. 1/198 17-CRL 113 200/5 Uen  
   User Guide for the TITAN TTCN-3 Test Executor
3. 155 17-CNL 113 512 Uen   
   EPTF Core Library for CLL for TTCN-3 toolset with TITAN, Function Specification
4. EPTF Core Library for TTCN-3 toolset with TITAN, [Reference Guide](http://ttcn.ericsson.se/TCC_Releases/Libraries/EPTF_Core_Library_CNL113512/doc/apidoc/html/index.html)
5. 27/155 16-CNL 113 512   
   EPTF CLL StatHandler, Function Description
6. 16/155 16-CNL 113 512  
   EPTF CLL Statistics Measure, Function Description