Chapter

37

Adaptation of Generated Code

This chapter contains information about making an adaptation of the generated code to make it communicate with the generated code.

The first part of this chapter describes the interface to be used when writing the adaptation, the GCI Interface. The different parts of the interface are explained in detail with recommendations and examples of use.

The second part of this chapter is a C implementation of GCI, this can be used as a reference documentation.

The third part describes the EGci (extended GCI) value construction and functions.

The fourth section introduces the Adaptation Framework that can be used to implement GCI functions in a straightforward way with ready-made low-level protocol implementations such as TCP/IP.

The fifth and final section describes the means and measures to complete the adaptation. It describes the requirements on the adaptation in terms of the necessary functions to implement.

The GCI Interface

The Generic Compiler Interpreter (GCI) interface standardizes the communication between a TTCN component supplied by a vendor and other test system components supplied by the customer, together forming a MOT, Method Of Test.

The GCI interface focuses on what an ATS needs in order to execute in term of functionality, and on what is needed in order to integrate TTCN into a larger system. This chapter contains a natural language description of the interface and a GCI C Reference.

The GCI Interface Model

The main purpose of the GCI interface is to separate TTCN behavior from protocol and test equipment specific code. The GCI interface shall be a standardized set of functions. Communication shall be done by calling functions, passing arguments to the functions and return values from the functions, and in no other way. The interface is bidirectional which means that both parties (the TTCN Runtime Behavior and the Test Adaptation) must provide services to each other. The TTCN Runtime Behavior shall at least provide services for handling values and managing tests (evaluating test cases etc.) and the Test Adaptation shall provide the protocol/test equipment specific parts of an executable (send, snapshot, timer functions etc.). The implementation of the functions in the TTCN Runtime Behavior may only depend on the ATS and 9646-3, no extra information shall be needed to implement them.

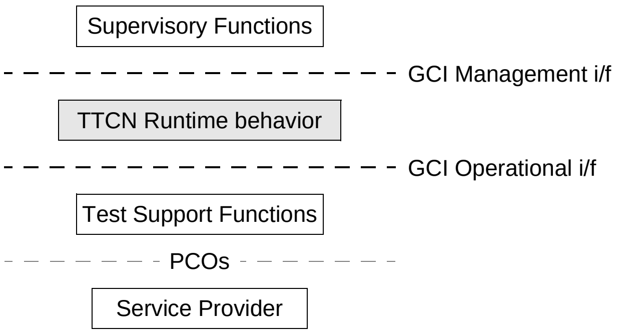

Figure 265: The GCI model

Informal Description of the Test Run Model

This section contains an informal description of how a test run might look like using the GCI interface. A test run is defined as a complete test session, where a selection of test cases are run to ensure a specific behavior of the IUT.

A test run begins when the user decides to run a test case or a collection of test cases. He will then use the Supervisory functions to start test cases and monitor their verdicts. The TTCN Runtime Behavior then executes TTCN, occasionally using the operational interface (send, snapshot, etc.) to gather information or to request some service. The TTCN Runtime Behavior handles verdicts internally during a test case and returns the verdict at the end of a test case.

Before using the TTCN Runtime Behavior, it must be initialized. After that the user chooses which test case to run using the supervisory functions. The test case returns a verdict which the user can use to form reports or stop test runs. The TTCN Runtime Behavior or the GCI interface does not impose any restrictions on this part. Any number of test cases can be run and each is commanded using the Supervisory functions.

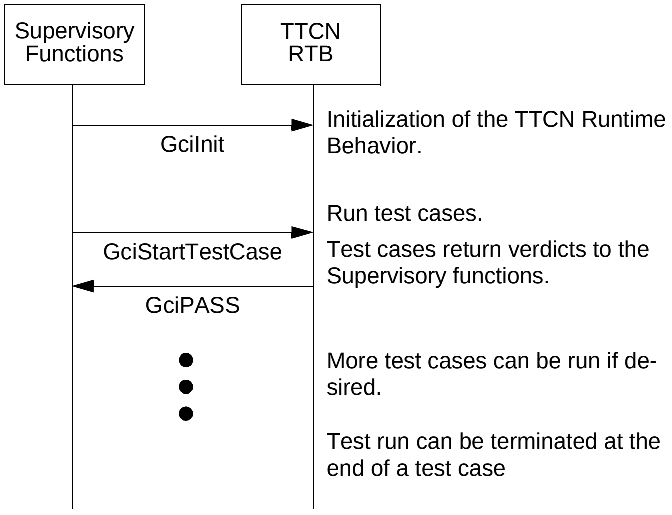
Supervisory Functions
TTCN RTB
GciInit
Initialization of the TTCN Runtime Behavior.
Run test cases.
GciStartTestCase
Test cases return verdicts to the Supervisory functions.
GciPASS
More test cases can be run if desired.
Test run can be terminated at the end of a test case

Figure 266: Start of test run

The test case at some point will try to send a message on some PCO. The TTCN Runtime Behavior then passes information to the Test Adaptation about the message to send and on which PCO to send it on. It is the responsibility of the Test Adaptation to properly encode the message and actually send it on some media (e.g. sockets, screen, printer port, pipe). Note that control is in the Test Adaptation until it returns to the TTCN Runtime Behavior. When control returns to the TTCN Runtime Behavior it assumes that the message was sent correctly and continues execution of the test case.

Eventually the test case will have emptied its possibilities to act and needs input from the environment. It therefore passes control to the Test Adaptation in order to take a snapshot of the IUT. Within the snapshot, the Test Adaptation then checks all PCO's for incoming messages and all timers for time-outs. If a message has arrived on a PCO, the Test Adaptation must decode the message and translate it into a proper GCI value. If a timer has timed out, the Test Adaptation must record which timer. The Test Adaptation acts as a filter between the IUT and the TTCN Runtime Behavior. Note that the actual reception of a message or time-out could be handled somewhere else (e.g. in an interrupt routine), only the official communication that the event has taken place must be done in SNAPSHOT.

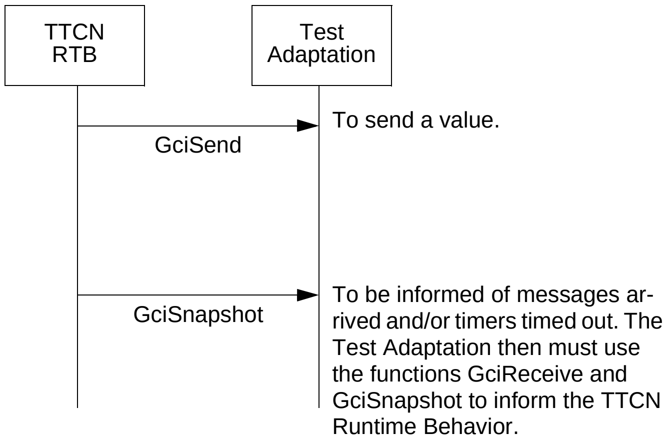
TTCN RTB
Test Adaptation
To send a value.
GciSend
GciSnapshot
To be informed of messages arrived and/or timers timed out. The Test Adaptation then must use the functions GciReceive and GciSnapshot to inform the TTCN Runtime Behavior.

Figure 267: Send and Snapshot

Preliminary verdicts may be set during the test execution and when a final verdict is set, the test case returns to the Test Adaptation immediately. The Test Adaptation then has freedom to decide to continue the test run or not. A final verdict has meaning only in the context of the current test purpose. The meaning of a final verdict is not specified in the context of an entire collection of test cases. For example, when doing regression tests, all test cases should be run regardless of how many failed in order to get a complete picture of the status of the IUT.

Which Does What?

The table below summarizes the responsibilities of the two GCI parties, the TTCN Runtime Behavior and the Test Adaptation. It is meant to describe the model and basic assumptions being made.

TTCN Runtime Behavior	**Test Adaptation**
SEND: Builds the object to be sent and requests the actual sending from the Test Adaptation.	SEND: Receives the object that shall be sent and transfers this to the IUT. This might include encoding in some form.
RECEIVE: When a value is put on a PCO queue by the Test Adaptation, it remains there until a matching receive line is found in TTCN.	RECEIVE: Builds the received object (from the IUT) into a GCI value, which may include some form of decoding, and notifies the TTCN Runtime Behavior that the message has arrived.
Sets the verdict (based on the test case).	Treats the verdict (i.e. decides if the test run should continue or not).
Impl. of value representation.	Encoders/decoders.
Uses the value repr.	Uses the value repr.
Provides test cases. The test cases are sensitive to test case selection references.	Determines which test cases to run and in which order.
Uses send on messages.	Provides the send functionality.

TTCN Runtime Behavior	Test Adaptation
Uses SNAPSHOT to fix a view of the status of the IUT. Expects all changes in system status to be recorded in SNAPSHOT.	Reads system status in SNAPSHOT and records this through GCI interface for the TTCN Runtime Behavior.
Uses the LOG function to log TTCN Runtime Behavior.	Provides the LOG functionality, and also decides on what level logging should be done.

Case Studies

This section contains case studies of a send and a receive. For the case studies, we use an example PDU shown in Figure 268.

TTCN PDU Type Definition		
PDU Name: pdu1		
Field Name	**Field Type**	**Comment**
a	INTEGER	w value 19
b	BOOLEAN	w value FALSE
Comment: Example pdu for GCI case studies		

Figure 268: Example table

Case Study: SEND

Nr	Lbl	Statement Line	Cref	Comment
		`L ! CR`	`CR_c`	

The semantic of the TTCN send statement is as follows:

1. Build the SendObject using the constraint reference.
2. Execute the assignments.
3. Send the SendObject on the PCO.
4. Execute the timer operations.
5. LOG, at least the PCO and the SendObject must be logged.

The SendObject above is a temporary object containing the object to be sent. All steps above are initiated by the TTCN Runtime Behavior.

Steps 3 to 5 require communication with the environment (IUT). The TTCN Runtime Behavior will build a SendObject using the constraint CR_c above. Then it will call the Test Adaptation function Send. By doing this, the TTCN Runtime Behavior has ensured that the message gets sent on whatever PCO was stated. The Test Adaptation function Send then encodes the message using the transfer syntax of the protocol and then sends the message on the medium that represents the PCO. The arguments passed *to* Send is the PCO and the message in the GCI representation, and the *side effect* of Send is to send the message in protocol representation on the PCO in test system representation. Note that the encoding and sending on a PCO might very well be represented as a function call.

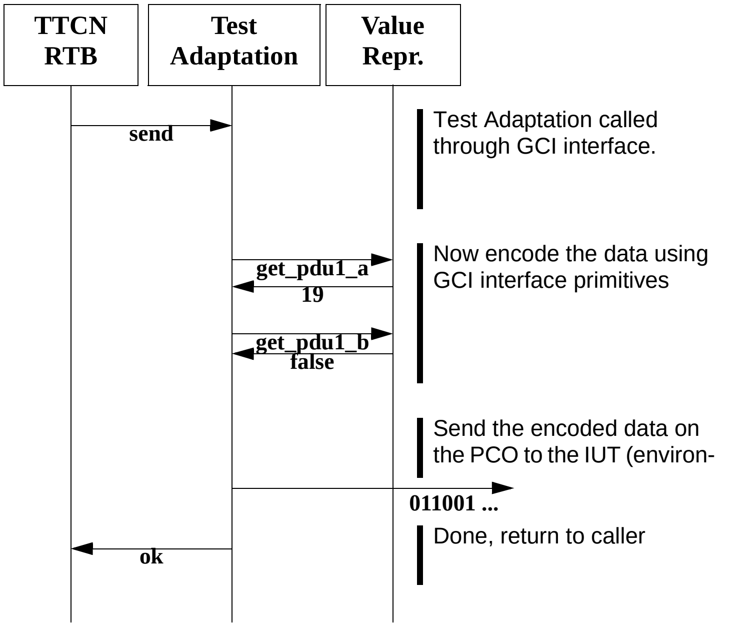
TTCN RTB
Test Adaptation
Value Repr.
send
Test Adaptation called through GCI interface.
get_pdu1_a
19
Now encode the data using GCI interface primitives
get_pdu1_b
false
Send the encoded data on the PCO to the IUT (environ-
011001 ...
ok
Done, return to caller

Figure 269: The send event

Case Study: RECEIVE

Nr	Lbl	Statement Line	Cref	Comment
		`L ? CC`	`CC_c`	

The receive is a little more complicated than the send. Send is initiated by the TTCN Runtime Behavior while receive is an internal event in the TTCN Runtime Behavior. The actual reception of messages is done in the Test Adaptation and these actions are communicated to the TTCN Runtime Behavior in SNAPSHOT. Note that there may be a difference between the reception of a message and the notification to the TTCN Runtime Behavior that the message has arrived. The TTCN Runtime Behavior calls the function SNAPSHOT and when it returns, the TTCN Runtime Behavior expects all time-outs to have been recorded and all received messages to be put on the correct PCO queues in the correct format. In this case we study the message event only and not the time-out.

> A message has been received from the IUT (not necessarily in SNAPSHOT) and the TTCN Runtime Behavior must be told that the message has been received. This shall be done in SNAPSHOT using the GCI function provided. The message and the PCO must be presented in a representation understood by the TTCN Runtime Behavior. Note that the message probably needs to be translated into the GCI value representation somewhere but again not necessarily in SNAPSHOT.

If SNAPSHOT does what is stated above, the TTCN Runtime Behavior will do the following on the receive line in the test case:

> The PCO queue is checked and if there is a message there, that message is matched against the constraint reference. If the message matches, it is removed from the PCO queue and the receive statement is considered to be TRUE, otherwise, the next alternative is checked.

In Figure 270, we assume that decoding is done in SNAPSHOT. The message is decoded using the value manipulating primitives of the GCI interface. When the message has been decoded into something known to the TTCN Runtime Behavior, it is easy to assign it to a PCO using the GCI function receive.

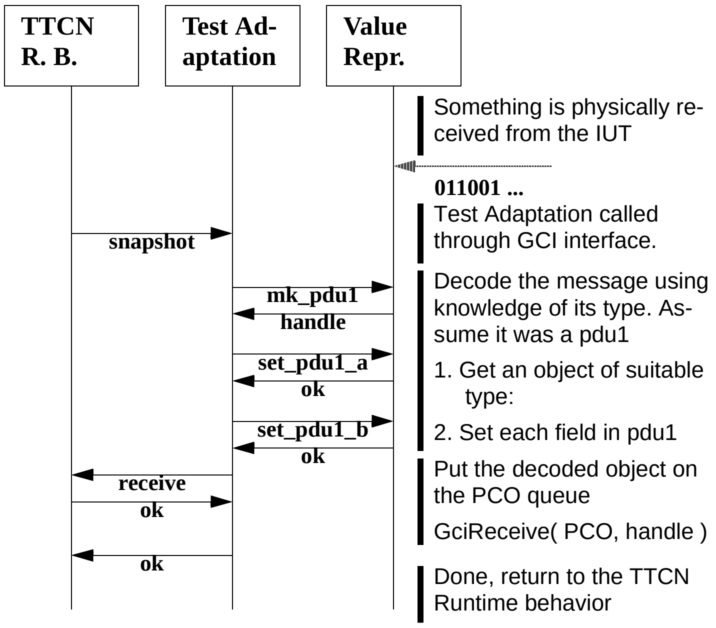
TTCN R. B.
Test Adaptation
Value Repr.
Something is physically received from the IUT
011001 ...
Test Adaptation called through GCI interface.
snapshot
Decode the message using knowledge of its type. Assume it was a pdu1
1. Get an object of suitable type:
2. Set each field in pdu1
mk_pdu1
handle
set_pdu1_a
ok
set_pdu1_b
ok
Put the decoded object on the PCO queue
GciReceive(PCO, handle)
receive
ok
ok
Done, return to the TTCN Runtime behavior

Figure 270: The receive event

Methods Used

This section contains general thoughts on methods used for designing the GCI interface. Choices have been made such as to provide reasonable trade-offs between speed and ease of use. An efficient implementation in C should be possible with the choices made.

Identifying Global Objects in TTCN/ASN.1

One problem is how to identify global objects in TTCN, PCOs, CPs, ASPs etc. There are a number of requirements that should be met:

1. Execution time:

   The TTCN behavior, operational and value interface functions are likely to be called many times in a time sensitive context which requires them to be fast. The management functions does not have this requirement.

2. Compilation time:

   The Test Adaptation and the TTCN Runtime Behavior are likely to be used together with a graphical interface (GUI) or integrated into a larger system. This integration should be possible without re-compiling the GUI or the larger system every time a change has been made in the ATS. This requires that dependencies between the ATS and the management interface implementation must be kept to a minimum.

There are at least three alternative solutions to be considered:

1. Straight name reference using target language semantics, i.e start test case “TC1” by writing the function call “TC1()” in C, and the value of the variable “TCVAR1” is expressed simply as TCVAR1
2. Assign an integer to each object. Symbolic names could then be used. The examples from above would then be:

   ```
   #define TC1 192
   #define TCVAR1 1211
   Verdict = GciStartTestCase( TC1 );
   Display( GciGetValue( TCVAR1 ));
   ```
3. Use string references.

   ```
   Verdict = GciStartTestCase( TC1 );
   Display( GciGetValue( TCVAR1 ));
   ```

Solution 1 gives the fastest execution but is very unsuitable for integration into a larger application. Solution 2 is at least possible to integrate into a GUI, and almost as fast as solution 1. Solution 3 is excellent for integration but gives slow execution.

In the GCI interface, we choose to use solution 3 (string references) in the management functions and solution 2 in the TTCN behavior functions. It would then be very easy to integrate an ETS into an already existing test system since the interface is string based while we still maintain speed in the execution of test cases.

A similar problem is how to identify fields in structured types. The simplest solution is to address them using integers field 0, field 1, etc., but then the Test Adaptation writer would get very little support from the compiler. From the Test Adaptation writers point of view, he might want to address the fields using names and also get type checking aid from the compiler. The Test Adaptation writer also wants to write generic functions for encode/decode. The GCI interface supports both views, one view where substructures are identified by numbers, and one which in effect use name addressing.

Test Suite Operation Definitions

The TTCN Runtime Behavior requires that all test suite operations are defined as functions at run-time. Arguments to and return values from the test suite operation functions shall use the GciValue* format. The name of the test suite operation function shall be the same as the name of the test suite operation in TTCN.

Logging

The log will provide a view of a test run. Most things happening in the system will need to be logged. We have identified a number of requirements on the log functionality:

1. It should be possible to only log every event visible to the TTCN Runtime Behavior. I.e. the sending of a message should leave a note in the log while the internal behavior of the TTCN Runtime Behavior should not.
2. For debug purposes, it must be possible to report every event in the system.
3. The granularity of the log must be easy to change by the Test Adaptation writer. It should be possible to be quite specific when logging, i.e. log only sends and receives.

The method that the GCI interface suggests is massive logging. Everything is logged at any time and every type of event (send, receive etc.) is assigned a number. That way it is easy for the Test Adaptation writer to write a log function that only logs interesting log messages.

Value Representation

The GCI interface must provide a stable way of manipulating values. This is most important for the Test Adaptation writer to be able to access and build values in his encode and decode functions. The GCI interface proposes not to specify the value representation but rather the **methods** which can be used on it. The GCI interface specifies what the Test Adaptation writer is able to do with values. The reason for this is that most vendors supporting the GCI interface would like to have their own value representation within their TTCN Runtime Behavior, and the representation used within TTCN has different requirements than the representation needed for the Test Adaptation.

Introduction to the GCI Interface

This section contains the description of the GCI interface in natural language. The interface functionality is sorted by responsibility. The management, behavior and value interfaces are interfaces to the TTCN Runtime Behavior. The operational interface is the interface to the Test Adaptation.

Management Interface

The management interface consists of those functions necessary for initiating and managing test runs. They provide an API to the ATS. These functions are used by the Supervisory functions to govern test runs.

Initiation of the TTCN Runtime Behavior

Must be called before the TTCN Runtime Behavior is used in any way. It will read test suite variables (using primitives in the operational i/f), set up test case selection references, test suite constants and any other initialization necessary. It should only be called once.

Start Test Case

Shall run a test case according to the dynamic behavior in 9646-3. Input shall be a test case name or test case group name. The test case or all the test cases in the test group shall then be run in the order which they were listed in the ATS. If the selection reference of a test case or test case group is false, that test case or test case group shall not be executed. Verdict shall be set and communicated back to the caller. In case several test cases have been run, the verdict shall be the most negative of the verdicts from each test case (if one test case is FAIL, the verdict would be FAIL, if all are PASS, the verdict would be PASS).

Start Test Component

This function works just line the function to start test cases, but starts test components for example when concurrent TTCN is used.

Test Case List

A list of names of test cases.

Test Case Group List

A list of names of test case groups.

Number of Timers and PCOs

Two functions are needed to return information about the amount of timers and PCOs in the system.

Information About Timers and PCOs

Help functions are needed to get some extra information about timers and PCOs. (For example name and type.)

Configuration Information

A set of functions is needed to retrieve information about configurations when running concurrent TTCN.

Accessing TTCN Values

Values of TTCN (and ASN.1) objects must be accessible from the Test Adaptation. A general access function will be provided. Information passed shall be an object name and the information passed back shall be the value of the object. Valid objects shall be test suite parameters, test case selection expressions, test suite constants, test suite variables and test case variables.

Behavior Interface

The behavior interface consists of functions used to notify the TTCN Runtime Behavior of events in the IUT. The functions are a formalism of what the TTCN Runtime Behavior needs to know about the status of the IUT. The functions are implemented in the TTCN Runtime Behavior and must be used in Snapshot.

Receive

Whenever a message has been received and decoded it must be passed to the TTCN Runtime Behavior in SNAPSHOT in order for it to match it with the constraint. The information passed shall be the PCO descriptor and the value. Note that the value passed in receive must be a GciValue*.

Time Out

Whenever a timer has timed out the TTCN Runtime Behavior must be notified of it in SNAPSHOT. The information passed shall be the timer descriptor.

Done

Whenever a test component has terminated its execution the TTCN Runtime Behavior needs to know about this. The information passed is the descriptor for the given test component.

Operational Interface

The operational interface consists of those primitives necessary for the TTCN Runtime Behavior to implement TTCN, i.e. Send, Snapshot, StartTimer etc. This part of GCI can be compared to POSIX. TTCN Runtime Behavior requires that the functions are defined.

Test Suite Parameters

The TTCN Runtime Behavior shall call the Test Adaptation once for each test suite parameter during the initialization of the TTCN Runtime Behavior. Information passed shall be a handle for the value, the parameter name and the PICS/PIXIT reference.

Test Suite Operations

Each test suite operation must be defined in the Test Adaptation. The name of the test suite operation function shall be the same as in the ATS, and the order of the parameters shall also be the same.

Create

When a test component needs to be created this function will be called from the TTCN Runtime Behavior.

Configuration

When different tests are run, different configurations might be needed. This function is called from the TTCN Runtime behavior to set up a configuration before the test goes on.

Snapshot

The status of the IUT must be read. Events in the IUT must be translated for the TTCN Runtime Behavior to be aware of them. The communicating interface between the TTCN Runtime Behavior and the IUT (environment) is made up of messages that has arrived and timers that have timed out. Therefore there are only two requirements on SNAPSHOT: 1) any received messages shall be recorded and 2) any timers that have timed out shall be recorded. The recording shall be done by using the provided GCI functions described in the Behavior i/f.

Send

This function is used by the TTCN Runtime Behavior to ensure that messages get sent. Send probably involves encoding and physical sending but might as well translate to encoding of some parts of the message and use of a lower layer service. The information passed to the function shall be a PCO descriptor and the value to be sent.

Start Timer

When the TTCN Runtime Behavior needs to start a timer. Information passed shall be the timer descriptor, the integer timeout duration and the timer unit.

Read Timer

Shall pass the current timer value back to the caller. Information passed to the function shall be a timer descriptor.

Cancel Timer

Shall stop a timer identified by a timer descriptor.

Log

Shall log events on an appropriate format. An integer specifying log message type and a log string shall be passed as parameters.

Value Interface

The interface to values is a simple API in which the user is allowed to build and access GciValues in an ordered way. The actual values used are not specified, only the methods that can be used on them are included. This way will allow vendors to have their own value representation within their TTCN Runtime Behavior and still conform to GCI.

Three types of value primitives are needed. Primitives to access structured values:

- Primitives to build and access components of structured values
- Primitives to find and set the type of values
- Primitives to convert base values to and from the value domain of the target language

Value primitives can be further divided in the groups:

- SEQUENCE values, which include SET values and also TTCN ASP, PDU and Struct types
- SEQUENCE OF values, which include SET OF values
- CHOICE values
- OBJECT IDENTIFIER values
- BASE values, integers, booleans, strings and reals

Please note that SET and SET OF values are treated as SEQUENCE and SEQUENCE OF values because there is no semantic difference at this level of abstraction. The unordered behavior of SETs is considered a decoder problem and left to the decoder writer.

GCI C Code Reference

This section is a code description of GCI. It describes what each function does and why it is used. Some of the functions must be implemented by you in your adaptation. See section “Completing the Adaptation” on page 1502.

Predefined Types

These are the GCI value types with their respective value set:

GciStatus = { GciNotOk, GciOk }

GciVerdict = { GciFail, GciInConc, GciPass, GciNone}

GciTimeUnit ={ Gcips, Gcins, Gcius, Gcims, Gcis, Gcimin }

GciTimerStatus = { GCIRUNNING, GCISTOPPED, GCIEXPIRED }

GciPCOID = INTEGER

GciTimerID = INTEGER

GciPosition = <internal position value>

These are structured GCI value types with their respective members:

The general communication address type

```
typedef struct GciAddress {
  int     type;        /* Address type 0 - 100 is
                          reserved by Telelogic */
  char*   buffer;      /* The address stored in a
                          buffer */
} GciAddress;
```

The general time value

```
typedef struct GciTime {
  unsigned long time_val;
  GciTimeUnit   unit;
} GciTime;
```

Management Interface

The management interface consists of the functions necessary for initiating and managing test runs. They provide an API to the TTCN runtime behavior.

`GciStatus GciInit()`

Initiates the TTCN Runtime Behavior. Must be called before any test cases are started.

`GciStatus GciExit()`

Shuts down the TTCN Runtime Behavior system. Usually called last before closing a test session.

`GciVerdict GciStartTestCase( const char* TCorTGName )`

Calling this function with a test case name will start and run the given test case. The verdict returned is the verdict set by the test case.

If the function is called with a test group name, it will start and run the given test group. The verdict returned is the product of the verdicts set in the test cases or test groups contained in the given test group. The individual verdicts from the different test cases or test groups have to be extracted from the log. The verdict algorithm for test groups is defined as follows:

1. The verdict is PASS if the verdict of every test case is PASS.
2. The verdict is INCONC if the verdict of at least one test case is INCONC and all others are PASS.
3. The verdict is FAIL if the verdict of at least one test case is FAIL.

If the given name is not a valid test case nor test group, the function will return `GciNotOk` and log the following message:

```
No such Test Case or Test Group as <name> to run!
```

`GciVerdict GciStartTestComponent( char* TSName, GciValue* args )`

This function works just like GciStartTestCase but is used to start a parallel test component (PTC) when using concurrent TTCN. TSName is the name of the Test Step to run in the PTC.

`GciTCList* GciGetTestCaseList()`

This function returns a list of all test case names (including test cases in test groups).

`GciTCList GciGetTestCaseGroupList()`

This function returns a list of all test group names (including test groups in test groups). List of character strings.

`GciValue* GciGetValue(char* TTCNObjectName)`

This function returns the TTCN value object given the object's name. If the object name does not exist, the function will return NULL and log the following message:

```
GciGetValue: Could not find <name>! NULL returned
```

If the object name is a TTCN name, but not an object (variable, constant, etc...) the function will return NULL and log the following message:

```
GciGetValue: <name> not an instance! NULL returned
```

`int GciGetNoOfTimers( )`

Returns the number of timers in the system.

`int GciGetTimer( int index )`

Returns the identifier of the given timer index.

`char* GciGetTimerName( int timer )`

Returns the name of the given timer.

`int GciGetTimerIndex( int desc )`

Returns the index of the given timer descriptor.

`int GciGetNoOfPCOs( )`

Returns the number of PCOs in the system.

`int GciGetPCO( int index )`

Returns the descriptor of the given PCO index.

`char* GciGetPCOName( int pco )`

Returns the name of the given PCO.

`int GciGetPCOIndex( int desc )`

Returns the index of the given PCO descriptor.

`int GciGetPCOType( pco )`

Returns the type of a given PCO.

`int GciGetNoOfComponents( GciConf conf )`

Returns the number of components in the given configuration.

`GciComponent* GciGetComponent( GciConf conf, int index )`

Returns the indexed component in the given configuration.

`char* GciGetComponentName( GciComponent* conp )`

Returns the name of the given component.

`int GciGetComponentType( GciComponent* conp )`

Returns the type of the given component (GciMTC or GciPTC).

`int GciGetComponentDescriptor( GciComponent* conp )`

Returns the descriptor of the given component.

`char** GciGetComponentCPs( GciComponent* conp )`

Returns the CPs used by the given component.

`char** GciGetComponentPCOs( GciComponent* conp )`

Returns the PCOs used by the given component.

Behavior Interface

The behavior interface consists of the functions used to notify the TTCN Runtime Behavior of events in the IUT. The functions are a formalism of what the TTCN Runtime Behavior needs to know about the status of the IUT. The functions are implemented in the TTCN Runtime Behavior and must be used in Snapshot.

`GciStatus GciTimeout( int timerd )`

This function will change the status of the internal timer indexed by the timer descriptor `timerd`. The timer will be marked as timed out.

`GciStatus GciReceive( int pcod, GciValue* value )`

This function will insert the received object into the internal PCO queue indexed by the PCO descriptor `pcod`.

Operational Interface

The operational interface consists of the functions necessary for the TTCN Runtime Behavior to communicate with the IUT, i.e. the dependencies on the protocol and test environment. It implements functionality for Send, Snapshot, StartTimer, etc. The TTCN Runtime Behavior requires that the functions are defined in the adaptation.

`GciValue* GciReadTSPar( char* name, char* pRef )`

The user has to implement how to read the test suite parameters. This code will be called one time for each test suite parameter in the system. Here `name` is the name of the test suite parameter and `pRef` is the PICS/PIXIT field (ex. file). The value returned by this function must be a pointer to a valid `GciValue` structure which can be successfully used by the TTCN runtime behavior. If for any reason this could not be done, NULL should be returned.

`GciStatus GciConfig( GciConf conf )`

When using concurrent TTCN the user is responsible for creating the configurations to be used for a given test. The creation of those configurations is done in this function using a set of help functions in the management interface to traverse the information.

`GciStatus GciSnapshot()`

This is a very important function that the user has to implement. This function is called from the run-time behavior when it needs input from the test environment (timers and/or PCO's).

- Communication lines (PCOs) must be checked. If something has been received, we must decode the message, build a receive object and insert the object into the correct, internal PCO queue by using the `GciReceive` function with appropriate PCO descriptor. The GCI Value Interface is used to build the object(s).
- Timers must be checked and their status must be reported to the TTCN run-time system. If the user uses the simple adaptation template he/she must self keep track of time and use `GciTime-`

`out` to mark a timer as timed out. If the user uses the timers adaptation template he/she can use the `AdSetTime` function (see below) to have this work done automatically.

`GciStatus GciSend(int pcod, GciValue* value)`

The user is responsible for implementing the send functionality. This involves identifying a given PCO and encoding the value before sending the message on that PCO.

`GciStatus GciStartTimer(int timerd, long duration, int unit)`

To start a timer the TTCN runtime behavior will call this function. Note that timer duration value is optional in TTCN but is always present here.

`GciStatus GciCancelTimer(int timerd)`

This function is called to stop the given timer. It should have the effect that the appropriate timer representation in the adaptation is stopped.

`long GciReadTimer(int timerd)`

This function is called when a timer is read. It must return the current timer value.

`GciStatus GciCreate(int ptc, char* tree, GciValue* args)`

This functions is called when a PTC is to be created to run a given dynamic behavior with the given arguments (concurrent TTCN).

`GciValue*` ***`<TS_Op>`*** **`(`** ***`<Arguments>`*** **`)`**

Every test suite operation must have this function defined. The name shall be the same as in the ATS and arguments must match the parameters in the ATS.

`GciStatus GciImplicitSend( GciValue* value )`

This function is called by the run-time behavior when an Implicit Send Event occurs in the test suite. It is up to the adaptation writer to define its exact implementation.

`GciStatus GciLog(int logId, char* LogString)`

The user has to implement parts of the logging facility. As the GCI document describes, some log identifiers are already implemented. The log identifiers (identifies the type of log message) in this function should NOT be changed as they follow the values listed in the GCI document. The TTCN Runtime Behavior uses this function for logging.

logId	**Description**	**The logString must contain:**
`Msg`	General message	
`StartTC`	Start test case	The name of the test case
`StopTC`	Stop test case	The name of the test case
`StartTS`	Start test step	The test step name
`StopTS`	Stop test step	The test step name
`StartDEF`	Start default	The default name
`StopDEF`	Stop default	The default name
`Verdict`	Final verdict set	The verdict set
`PVerdict`	Prel. verdict set	The verdict set
`Match`	Line tried matches	Line number
`NoMatch`	Line tried does not match	Line number
`SendE`	Send Event	The PCO name The type of message sent Constraint name
`RecE`	Receive Event[a]	The PCO name The type of message received Constraint name
`OtherE`	Otherwise Event	The PCO name
`TimeoutE`	Timeout Event	The timer name
`Assign`	Assignment	The Left hand side The right hand side (textually)

logId	Description	The logString must contain:
`StartT`	Start timer	The timer name The timer duration
`StopT`	Stop timer	The timer name
`CancelT`	Cancel timer	The timer name
`ReadT`	Read timer	The timer name The timer value
`Attach`	Attachment	The name of the attached test step
`ImplSend`	Implicit send	The PCO name The message type
`Goto`	Goto	The line number to which the jump will be made
`Rec`	Receive[b]	The PCO descriptor number
`Timeout`	Timeout[c]	The timer descriptor number

a. Logging that a receive line matches.

b. Note this is low level logging done when the Test Adaptation writer calls Gci-Receive (or GciTimeout)

c. Note this is low level logging done when the Test Adaptation writer calls Gci-Receive (or GciTimeout)

Value Interface

The interface to values is a simple API in which the user is allowed to build and access GciValues in an ordered way. The actual values used are not specified, only the methods that can be used on them are included. This way will allow vendors to have their own value representation within their TTCN Runtime Behavior and still conform to GCI.

Base Types/Values

These functions are used to transform actual values into the GCI representation. The interface is on base types only, so TTCN simple types are not visible in the interface.

`GciValue* GciMkINTEGER( int num )`
`int GciGetINTEGER( GciValue* value )`

Used for integer values, and simple type values with base type integer.

`GciValue* GciMkBOOLEAN( int bool )`
`int GciGetBOOLEAN( GciValue* value)`

Used for Boolean values, and simple type values with base type Boolean.

`GciValue* GciMkREAL(int mantissa, int base, int exponent)`
`GciReal GciGetREAL( GciValue* value)`

Used for Real values, and simple type values with base type Real.

`GciValue* GciMkBIT_STRING( const char* str )`
`char* GciGetBIT_STRING( GciValue* value)`

Used for bit string values, and simple type values with base type bit string.

`GciValue* GciMkHEXSTRING( const char* str )`
`char* GciGetHEXSTRING( GciValue* value)`

Used for hex string values, and simple type values with base type hex string.

`GciValue* GciMkOCTET_STRING( const char* str )`
`char* GciGetOCTET_STRING( GciValue* value)`

Used for octet string values, and simple type values with base type octet string.

`GciValue* GciMkNumericString( const char* str )`
`char* GciGetNumericString( GciValue* value)`

Used for values numerical strings, and simple type values with base type numerical string.

`GciValue* GciMkPrintableString( const char* str )`
`char* GciGetPrintableString( GciValue* value)`

Used for Printable string values, and simple type values with base type Printable string.

```
GciValue* GciMkTeletexString( const char* str )
char* GciGetTeletexString( GciValue* value)
```

Used for Teletex string values, and simple type values with base type Teletex string.

```
GciValue* GciMkVideotexString( const char* str )
char* GciGetVideotexString( GciValue* value)
```

Used for Videotex string values, and simple type values with base type Videotex string.

```
GciValue* GciMkVisibleString( const char* str )
char* GciGetVisibleString( GciValue* value)
```

Used for Visible string values, and simple type values with base type Visible string.

```
GciValue* GciMkIA5String( const char* str )
char* GciGetIA5String( GciValue* value)
```

Used for IA5string values, and simple type values with base type IA5string.

```
GciValue* GciMkT61String( const char* str )
char* GciGetT61String( GciValue* value)
```

Used for T61string values, and simple type values with base type T61string.

```
GciValue* GciMkISO646String( const char* str )
char* GciGetISO646String( GciValue* value)
```

Used for ISO646string values, and simple type values with base type ISO646string.

```
GciValue* GciMkGraphicalString( const char* str )
char* GciGetGraphicalString( GciValue* value)
```

Used for Graphical string values, and simple type values with base type Graphical string.

```
GciValue* GciMkGeneralString( const char* str )
char* GciGetGeneralString( GciValue* value)
```

Used for General string values, and simple type values with base type General string.

```
GciValue* GciMkENUMERATED( int value )
int GciGetENUMERATED( GciValue* value)
```

Used for Enumerated values, and simple type values with base type Enumerated.

```
GciValue* GciMkCHOICE(const char* name, GciValue* value)
GciValue* GciGetCHOICE( GciValue* value)
char* GciGetCHOICEName( GciValue* value)
```

Used for Choice values, and simple type values with base type Choice.

```
GciValue* GciMkOBJECT_IDENTIFIER()
int GciOBJECT_IDENTIFIERSize( GciValue* value)
GciValue* GciAddOBJECT_IDENTIFIERComponent(
   GciValue* value, int comp)
int GciGetOBJECT_IDENTIFIERComponent( GciValue* value,
   int index)
```

Used for Object identifier values, and simple type values with base type Object identifier.

```
GciValue* GciMkObjectDescriptor( const char* str )
char* GciGetObjectDescriptor( GciValue* value)
```

Used for ObjectDescriptor values, and simple type values with base type ObjectDescriptor.

```
GciValue* GciMkNULL( )
int GciGetNULL( GciValue* value)
```

Used for Null type values, and simple type values with base type Null type.

```
GciValue* GciMkANY( GciValue* value )
GciValue* GciGetANY( GciValue* value)
```

Used for ANY values, and simple type values with base type ANY.

```
GciValue* GciMkR_TYPE( int value )
int GciGetR_TYPE( GciValue* value)
```

Used for R_Type values, and simple type values with base type R_Type.

```
GciValue* GciMkPDU( GciValue* value )
GciValue* GciGetPDU( GciValue* value)
```

Used for PDU values, and simple type values with base type PDU.

Base Functions

```
int GciGetType( GciValue* val )
```

Shall return the type of a value. The type is represented as the number used to identify global objects, see “Identifying Global Objects in TTCN/ASN.1” on page 1457. The function is valid for all values, but some values may be untyped in which case the function returns 0.

```
GciValue* GciSetType( int type, GciValue* val )
```

Shall set the type of a value. Uses the same number as above. Valid for all values. Returns the input value with type set.

Value Management

The functions are divided into two meta sets derived from ASN.1: The sequence, and the sequence of, corresponding to struct and array in C. The choice type of ASN.1 is not represented as a meta class because all values in the GCI value representation can be typed and therefore is the choice value implicit in GCI.

The functions only works within their intended set (e.g. GciSetField(GciMkSEQUENCE(2), 1, GciMkINTEGER()) is well defined but GciSetField(GciMkSEQUENCEOF(), 0, GciMkINTEGER()) is not defined and certainly is unpredictable.

Sequence and Set Types/Values

```
GciValue* GciMkSEQUENCE( int size )
```

Creates a sequence value with size children.

```
GciValue* GciSetField( GciValue* seq, int index,
   GciValue* fld )
```

Sets the field identified by index to the given value. Indices start at 1. The function is undefined for indices greater than the size of the sequence.

`GciValue* GciGetField( GciValue* seq, int index )`

Returns the field identified by index. Indices start at 1. The function is undefined for indices greater than the size of the sequence.

`int GciSeqSize( GciValue* seq )`

Returns the number of fields declared in the sequence *seq*.

Sequence of and Set of Types/Values

`GciValue* GciMkSEQUENCEOF()`

Creates a sequence of value.

`GciValue* GciAddElem( GciValue* seqOf, GciValue* elem )`

Adds the element *elem* to the end of sequence of *seqOf*.

`GciValue* GciGetElem( GciValue* seqOf, int index )`

Returns the element number *index* of the sequence of *seqOf*. Indices start at 0. The function is undefined for indices greater than current size.

`int GciSeqOfSize( GciValue* seqOf )`

Returns the current number of elements in *seqOf*.

Examples

This section contains examples of how the mapping between TTCN/ASN.1 and their GCI representation could be done. The examples use a conceptual model for encoding/decoding, no error handling is done and no attention is paid to the fact that some functions would use pointers because of allocation issues.

Global objects (PCOs and types) are identified with an unique number. This number is given a symbolic reference which is its TTCN name with a *D* for Descriptor appended to the end of it. The number for the PCO *L* is therefore referenced as *LD*.

Encoding/Decoding Examples

Each example consists of a table and the corresponding encode (and/or decode) functions. The examples focus on the value representation so the transfer syntax is simple: Each value is preceded by a header. The header contains the type of the value (as a number) and in some cases its length (element count, not size in bytes), in real ASN.1, the header would be built using tags. The header is written/read using primitives BufWriteType, BufWriteSeqOfSize, etc. They are not encode/decode primitives but rather buffer primitives.

An encode function is a function that translates the GCI value onto a buffer, and a decode function is one that reads a value from a buffer and builds a value in the GCI representation. Encoding functions are called from SEND and decoding functions are called from SNAPSHOT. The buffer has type *Buffer* and could be anything behaving as a sequence of bytes. The primitives BufReadInt, BufReadBool are used to read and write GCI basic values.

ASN.1 SEQUENCE Type

ASN.1 PDU Type Definition
PDU Name: T1
Comment: Example PDU for GCI examples
Type Definition

`SEQUENCE {`
`  a INTEGER,`
`  b BOOLEAN`
`}`

Figure 271: Example

Encode

```
void EncodeT1( Buffer buf, GciValue* v)
{
  BufWriteType( buf, T1d );
  BufWriteInt( buf, GciGetField( v, 1 ));
  BufWriteBool( buf, GciGetField( v, 2 ));
}
```

Decode

```
void DecodeT1( Buffer buf, GciValue* v)
{
  int i;

  v = GciMkSEQUENCE( 2 );
```

```
    if ( BufReadType( buf ) != T1d )
      Error();
    GciSetType( T1d, v );

    i = BufReadInt( buf );
    GciSetField( v, 1 , GciMkINTEGER( i ) );

    i = BufReadBool( buf );
    GciSetField( v, 2 , GciMkBool( i ) );
  }
```

ASN.1 SEQUENCE OF Type

ASN.1 PDU Type Definition
PDU Name: T2
Comment: Example PDU for GCI examples
Type Definition
`SEQUENCE OF T1`

Figure 272: Example

Encode

```
  void EncodeT2( Buffer buf, GciValue* v )
  {
    int i;

    BufWriteType( buf, T2d );
    BufWriteSeqOfSize( buf, GciSize( v ) );
    for (i = 1 ; i <= GciSize( v ) ; i++ )
      {
        EncodeT1( buf, GciGetElem( v, i ) );
      }
  }
```

Decode

```
  void DecodeT2( Buffer buf, GciValue* v )
  {
    int i, seqOfSize;
    GciValue* elem;

    if ( BufReadType( buf ) != T2d )
      Error();
    GciSetType( T2d, v );

    seqOfSize = BufReadSeqOfSize( buf );
    for ( i = 1 ; i <= seqOfSize ; i++ )
      {
        DecodeT1( buf, elem );
        GciAddElem( v, elem );
      }
  }
```

ASN.1 CHOICE

ASN.1 PDU Type Definition
PDU Name: T3
Comment: Example PDU for GCI examples
Type Definition

`CHOICE {`
`  c1 T1,`
`  c2 T2`
`}`

Figure 273: Example

Encode

```
  void EncodeT3( Buffer buf, GciValue* v )
  {
    switch ( GciGetType( v ))
      {
      case T1d:
        EncodeT1( buf, v );
        break;
      case T2d:
        EncodeT2( buf, v );
        break;
      default:
        Error();
      }
  }
```

Decode

```
  void DecodeT3( Buffer buf, GciValue* v )
  {
    switch ( BufGetType( v ))
      {
```

```
    case T1d:
      DecodeT1( buf, v );
      break;
    case T2d:
      DecodeT2( buf, v );
      break;
    default:
      Error();
    }
}
```

In-Line ASN.1 Type

ASN.1 PDU Type Definition
PDU Name: T4
Comment: Example PDU for GCI examples
Type Definition

`SEQUENCE {`
`  c1 T1,`
`  c2 SEQUENCE {`
`    c1 INTEGER,`
`    c2 T2`
`  }`
`}`

Figure 274: Example

Encode

```
void EncodeT4( Buffer buf, GciValue* v )
{
  GciValue* tmp;

  BufWriteType( T4d );
  /* Encode first field */
  EncodeT1( GciGetField( v, 1 ));
  /* Now encode inline type definition */
  tmp = GciGetField( v, 2 );
  BufWriteInt( buf, GciGetField( tmp, 0 ));
  EncodeT2( buf, GciGetField( tmp, 1 ));
}
```

Decode

```
void DecodeT4( Buffer buf, GciValue* v)
{
  int i;
  GciValue* tmp;

  v = GciMkSEQUENCE( 2 );
```

```
    if ( BufReadType( buf ) != T4d )
      Error();
    GciSetType( T4d, v );
    DecodeT1( buf, tmp );
    GciSetField( v, 1, tmp );
    tmpseq = GciMkSEQUENCE( 2 );
    i = BufReadInt( buf );
    GciSetField( tmpseq, 1, GciMkINTEGER( i ) );
    DecodeT2( buf, tmp );
    GciSetField( tmpseq, 2, tmp );
  }
```

TTCN Examples

`myQueue` below is the Test Adaptation writers own representation of the PCO queue(s). In this example there is only one PCO, called `L` (`Ld` for its number). Note that the number `Ld` can be any number (most certainly not zero).

Snapshot Example

Snapshot must read the status of the IUT and tell this to the TTCN Runtime Behavior.

```
void GciSnapshot()
{
  GciValue* v;

  /* Check if anything has arrived,  */
  /* This would be a while statement */
  /* in a blocking context           */
  if ( ! myQueue[ 0 ].empty )
    {
      switch ( BufPeekType( myQueue[0].buf ))
        {
        case T1d:
          DecodeT1( myQueue[0].buf, v );
          break;
        case T2d:
          DecodeT2( myQueue[0].buf, v );
          break;
        default:
          Error();
        }

      GciReceive( Ld, v );
    }

  /* Nothing has happened. */
}
```

Send Example

For the send example the following ASP table is used to show an API view of encode.

ASN.1 ASP Type Definition
ASP Name: TCONreq
Comment: Example ASP for GCI examples
Type Definition

`SEQUENCE {`
`  num INTEGER,     -- sequence number`
`  pdu T1           -- Embedded pdu`
`}`

Figure 275: Example

```
/* Two examples of send functionality */
GciStatus GciSend( int pcoDescr, GciValue* msg )
{
  if ( pcoDescr == PcoToSocket )
    {
      /* Encode first */
      if ( GciGetType( msg ) == T1d )
        {
          EncodeT1( buf, msg );
          BufSocketSend( buf );
        }
    }
  else if ( pcoDescr == PcoToAPI )
    {
      if ( GciGetType( msg ) == TCONreqd )
        {
          Buffer pdu;
          EncodeT1( pdu, GciGetField( msg, 1 ));
          /* Call to lower layer */
          T_CONreq( GciGetField( msg, 0 ), pdu );
       }
    }
}
```

EGci Value Construction and Functions

The GCI interface has been defined as a base for executable test suite adaptation. One part of this interface covers values, how to build them, access them, etc. While this interface was not enough, a set of functions has been added with the EGci prefix so that the GCI functions (a set defined by a standard) was not changed. The purpose of this section is to describe this paradigm of value construction and the current EGci functions available.

Value Construction

The major difference in value handling is how values are constructed. Values are either created by the GciMk-prefixed functions, followed by a GciSetType call, or value are created by a call to the EGciMkValue function. The run time system is compatible with both alternatives but the latter is preferred. The difference is that the GciMk-function creates untyped values of a given predefined type (SEQUENCE, SET, INTEGER, etc.) while the EGciMkValue takes the type descriptor of the user-defined type and creates the complete value structure with correct types throughout the whole value structure. Values of type SEQUENCE OF, SET OF and CHOICE cannot be fully instatiated since the size/choice of the value is not known at code generation time. Elements of the SEQUENCE/SET type are appended explicitly and CHOICE values are also selected explicitly.

The procedure is to create the value structure (with uninitialized leaf values) and then extract the proper value (container) through ordinary field access and extraction functions. Then the value is assigned using the EGciAssign function. When available, for simple non-structured values, EGciSet/GciSet-functions can also be used on the extracted values.

Below you will find two alternatives for creating a value given the following type:

```
T ::= SEQUENCE
{
  a SEQUENCE {
     b MyINTEGER
  },
  c MyINTEGER
}

MyINTEGER ::= INTEGER
```

```
v T := { a { b 17 }, c 42 }
```

Alternative 1 – Using GciMkSEQUENCE and GciSetType

The individual values through out the structured value have to be created and the correct type set. It is also impossible to set the correct type for the inlined types since they are unknown.

```
...
value = GciMkSEQUENCE(1);
GciSetType(value, GcTD);

inner_seq = GciMkSEQUENCE(1);
/* GciSetType(inner_seq, ?); This is not possible
since the inlined type is not available to the user.
A generated no-name type exists but is not known. So
the SEQUENCE will be untyped (only a SEQUENCE). This
is not a problem with alternative 2. */

b = GciMkINTEGER(17);
GciSetType(b, GcMyINTEGERD);
GciSetField(inner_sequence, 1, b);
c = GciMkINTEGER(42);
GciSetType(c, GcMyINTEGERD);
GciSetField(value, 1, inner_seq);
GciSetField(value, 2, c);
...
```

Alternative 2 – Using EGciMkValue (Recommended Approach)

Using the EGciMkValue function makes it somewhat more readable, but what is more important is that the final value has the correct type throughout the structure (all done by the generated type information used in the call to EGciMkValue). All field names are also correct.

```
...
value = EGciMkValue(GcTD);
/* With this call the whole structure is created and
the basic leaf values are only extracted and set
("filled in"). */

tmp = GciGetField(value, 1);
tmp = GciGetField(tmp, 1);
EGciSetINTEGER(tmp, 17);

tmp = GciGetField(value, 2);
EGciSetINTEGER(tmp, 42);
...
```

Available Functions

`GciValue * EGciMkValue(int)`

Takes the generated type identifier constant `Gc<type>D` and creates a proper value using all available type information generated. This function is faster than the `EGciMkValueFromTypeName` but the type identifier constants can potentially change their numeric value if new types are added to the test suite. Any depending encoder/decoders have to be recompiled if the generated type information changes. If generation dependent information is required, the `EGciMkValueFromTypeName` should be used.

Example 275 ––

Given the type `MyType`, the call for constructing a value would be `EGciMkValue(GcMyTypeD)`.

–––

`GciValue * EGciMkValueFromTypeName(const char* tname)`

Same as `EGciMkValue` but it takes the (bare) name of the type.

Example 276 ––

Given the type `MyType` the call for constructing a value would be `EGciMkValueFromTypeName("MyType")`.

–––

`void EGciRmValue(GciValue *value)`

This function deletes a constructed value.

`GciStatus EGciAssign(GciValue *destination, GciValue *source)`

Given the two values, which must be instantiated values of the same type, the source is assigned to the destination value.

`GciStatus EGciSetMemberByName(GciValue *val, const char* name, GciValue *mem_val)`

Sets the named member of a SEQUENCE, SET or CHOICE value to the given value (mem_value). This is the same as doing an extraction followed by an assignment.

`GciStatus EGciSetEmptySET_OF(GciValue *value)`

Sets the value to the defined state. That is, it is an empty SET OF value.

`GciStatus EGciSetEmptySEQUENCE_OF(GciValue *value)`

Sets the value to the defined state. That is, it is an empty SEQUENCE OF value.

`GciStatus EGciSetOMIT(GciValue *value)`

Sets the given optional value as omitted.

`GciStatus EGciSetBOOLEAN(GciValue *value, Bool v)`

Sets a constructed value of (base) type BOOLEAN to the given boolean value.

`GciStatus EGciSetSTRING(GciValue *value, const char *)`

Sets a constructed value of any string base type (BITSTRING, OCTETSTRING, IA5String, etc.) to the given string value.

`GciStatus EGciSetINTEGER(GciValue *value, int number)`

Sets a constructed value of (base) type INTEGER to the given integer value.

`GciStatus EGciSetNULL(GciValue *value)`

Sets a constructed value of (base type) NULL to the NULL value.

`GciValue * EGciSetENUMERATEDByName(GciValue *val, const char* enum_name)`

Sets the actual value of the ENUMERATED value to one of its declared named numbers. For example, EGciSetENUMERATEDByName(gcivalue, “foo”), where “foo” is one of the named numbers in the type.

Example 277 ———————————————————————————

```
EType ::= ENUMERATED { first, second, third }

GciValue *eval = EGciMkValue(GcETypeD);
EGciSetENUMERATEDByName(eval, “second”);
```

———————————————————————————————

`const char * EGciGetENUMERATEDName(GciValue *value)`

Retrieves the name of the current value.

Example 278 ––

Continuing Example 277, a call to `EGciGetENUMERATED-Name(eval)` would return the string `“second”`.

––

`GciValue * EGciGetChoiceMemberByName(GciValue *value, const char *name)`

CHOICE values are not constructed all the way down to its leaf value since the information about the selected field is not available in the type information. When this function is called, the value construction is continued and proper value will be constructed for the selected CHOICE field (name) and the value can then be assigned properly.

`GciValue * EGciGetChoiceMemberByTag(GciValue *value, GciTagClass tag_class, int tag)`

Equivalent to `EGciGetChoiceMemberByName` but the unique tag of the choice element instead of the name is used.

`GciValue * EGciGetSEQUENCE_OFElement(GciValue *value)`

Constructs a new value given the underlying/contained type of the SEQUENCE_OF type. The newly constructed value must be appended using the `GciAddElem` function.

`GciValue * EGciGetSET_OFElement(GciValue *value)`

Constructs a new value given the underlying/contained type of the SET_OF type. The newly constructed value must be appended using the `GciAddElem` function.

`Bool EGciGetAnyValue(GciValue *value)`

Predicate to check if the value is ANY value attribute. Note that this has nothing to do with the `GciGetANY` function since that function operates on values of the type `ANY`.

`Bool EGciGetAnyOrOmit(GciValue *value)`

Predicate to check if the value is ANYOROMIT value attribute.

`Bool EGciGetIfPresent(GciValue *value)`

Predicate to check if the IF_PRESENT value attribute is specified for this value.

`Bool EGciGetDefault(GciValue *value)`

Predicate to check if the instatiated value comes from the default value of the type or not.

Error Handling

If an error occurs during execution, the error state is propagated up through the call stack and error information is added. An error has occurred when either the EGciGetErrorCount function returns a number greater than zero or a Gci/EGci-function call has returned GciNotOk. The error message is retrieved by EGciGetLastErrorMessage and reset by EGciClearError.

`unsigned int EGciGetErrorCount()`

Retrieve the current number of errors detected by the run time system.

`const char* EGciGetLastErrorMessage(void)`

Retrieve the last error message in text form.

`void EGciClearError()`

Clear the error state of the run time system.

Miscellaneous

`void EGciDumpValue(FILE *stream, const char *prefix, GciValue *value, const char *suffix)`

Prints the given value on stream. The prefix and suffix are printed before and after the value dump respectively.

`void EGciSetDebugStream(FILE *stream)`

Set the debug stream where logging will be made. It is set to `stderr` by default.

Examples

```
MyString         ::= IA5String
EmailAddress     ::= MyString
SnailmailAddress ::= SET { street MyString, number INTEGER }
AddressKind      ::= ENUMERATED { email, snailmail }

-- This represents a contact with name and an address.

Contact ::= SEQUENCE {
  name MyString,
  address_kind AddressKind
  address CHOICE {
    email     EmailAddress,
    snailmail SnailmailAddress
  }
}
```

A value would be constructed in the following way:

```
GciValue *tmp;
GciValue *address;
CciValue *contact;

contact = EGciMkValue( GcContactD);

tmp = EGciGetFieldByName(contact, “name”);
EGciSetSTRING(tmp, “Elvis”);

tmp = EGciGetFieldByName(contact, “address_kind”);
EGciSetENUMERATEDByName(tmp, “snailmail”);

 address = EGciGetFieldByName(contact, “address”);
 address = EGciGetChoiceMemberByName(address, “snailmail”);
 tmp = EGciGetFieldByName(address, “street”);
 EGciSetSTRING(tmp, “High road”);
 tmp = EGciGetFieldByName(address, “number”);
 EGciSetINTEGER(tmp, 42);
```

The Adaptation Framework

Introduction to the Adaptation Framework

The Adaptation Framework, technically referred to as “ACM”, is a collection of the functions that are needed to write a full-fledged adaptation implemented by plug-in libraries that can be used “as is”. Currently, the following plug-in modules are included in the TTCN suite distribution:

- TCP/IP socket communication implementation.
- Standard system-time timer implementation.

This means that if the target system uses TCP/IP to communicate, the adaptation writer can simply use the Adaptation Framework right from the box to get the IUT connected to generated code with minimal amount of work.

Note that the Adaptation Framework is provided as a standardized way to implement the GCI interface. This means that the Adaptation Framework *does not replace GCI, but rather complements it.* To build the adaptation, simply implement the needed GCI functions by calling the counterpart framework functions that use the compiled-in implementations for the low level code; in this distribution this would be the TCP/IP protocol for communication and a standard system time timer implementation. On the other hand, this means that one can continue to implement adaptations without using the framework altogether, the old way, and old adaptations will continue to work as before.

Examples of usage

The included adaptation residing under the ACM/ subdirectory contains an implementation of the GCI layer by framework functions and is instructive in displaying the way to work with framework functions.

Function reference

Communication data types

Defined constant values

Defined constant	Meaning
ACM_ADDRESS_TCPIP	The address type specifier used with the TCP/IP implementation of the Adaptation Framework for the `GciAddress` type.

More values will be defined in the file *acm.h* once more plug-in packages are available. The user can also specify new values for his/her own communication modules.

Note:

Values 0-100 are reserved for use by Telelogic.

Communication role of the executable test suite

```
typedef enum
{
  ACMClient,
  ACMServer
} ACMConnectionType;
```

The enumerated type *ACMConnectionType* is used when calling `ACMConnect()` to select whether the generated code should act as a server or client to the IUT.

Communication primitives

These functions are used to implement the GCI communication functions.

Initializing the communication and timer package

`GciStatus ACMInit(const GciTime* max_timeout, GciTimeUnit time_tick_unit)`

Must be called after GCIInit() but prior to any use of the ACM communication or timer primitives. The first argument, *max_timeout* defines the maximum amount of time the system will wait in blocked state for any timer. Setting this to a low value enables the possibility to write a polling snapshot function. A defined constant exists by the name of **ACM_ONE_YEAR_IN_MINUTES** that can be used as a default “big enough” number argument.

The second argument, *time_tick_unit*, depicts the unit of time (in GCI time units) used internally in the run-time system. Since all internal timer data is converted to and from this unit, it is important to set this to a value that encompasses approximately the value range of the time-out values used in the test suite. This parameter also restricts the maximum time that is available for a time-out value, as depicted below.

Internal unit	Maximum length of time before timer wraps
Gcips	4.3 seconds
Gcins	71 minutes
Gcius	49 days
Gcims	136 years
...	...

Resetting the Run-time System

```
GciStatus ACMReset()
```

This function cancels all timers, and resets all timer queues and PCO buffers. It is typically called before starting a new test case to ensure that all old data is removed, and to put the run-time system into an initial mode.

Registering the GciTimeout Function

```
GciStatus ACMRegisterTimeoutHandler(ACMTimeoutHandler
                                    timeouthandler)
```

This function needs to be supplied with a pointer to the *GciTimeout()* function before using the *ACMSnapshot()* function, since *ACMSnapshot()* calls the timeout function automatically for every timer that has expired.

Registering the GciReceive Function

```
GciStatus ACMRegisterReceiveHandler(GciReceiveHandler
                                    receive_callback)
```

This function needs to be supplied with a pointer to the *GciReceive()* function before using the *ACMSnapshot()* function, since *ACMSnapshot()* calls the receive function automatically for every data packet that has arrived.

Registering the Active Decode Function

```
GciStatus
ACMRegisterDefaultDecodeHandler(ACMDecodeHandler
                                decode_handler)
```

The function *ACMSnapshot()* calls *GciReceive()* automatically for every data packet that are ready to be received. In order to do this, however, a decode function needs to be registered that can decode the incoming data from its transfer syntax to the internal GCI value representation. This is easily done by the above call. Note that this opens the possibility to switch decoding during runtime, since any decode function that fulfills the requirements are eligible to be registered as the default decoder at any time. See "Encoding, and in Particular Decoding within ACM and GCI" on page 1512 in chapter 37, *Adaptation of Generated Code* for details on how to construct the decode function(s).

Registering a Specific Decode Function for a PCO

```
GciStatus ACMRegisterDecodeHandler(ACMDecodeHandler
                                   decode_handler,
                                   GciPCOID pco_id)
```

This function is similar to the previous function, but it enables a certain decode function to be connected with a certain PCO. This means that the function *ACMSnapshot()* will be able to use different decoding mechanisms depending on which PCO the message was received from. See "Encoding, and in Particular Decoding within ACM and GCI" on page 1512 in chapter 37, *Adaptation of Generated Code* for details on how to construct the decode functions.

Connecting to a Communication Port

```
GciStatus ACMConnect(GciPCOID pco_id,
                     GciAddress* address,
                      ACMConnectionType type,
                      unsigned int buffer_size)
```

This function maps the PCO identifier value internally to an external communication port and makes the connection. This makes it possible to use the PCO identifier in all subsequent calls, which makes the code quite clear.

In addition, we need to supply what type of role the ETS will have in the communication with the IUT - as a server (ACMServer) or as a client (ACMClient).

The *buffer_size* argument defines the largest possible buffer (in bytes) that can be received in one chunk. This should usually be set to the largest buffer needed to encode a value in the current value representation.

Disconnecting a Communication Port

```
void ACMDisconnect(GciPCOID pco_id)
```

This function disconnects a previously opened communication link.

Sending a Message

```
GciStatus ACMSend(GciPCOID pco_id, GciBuffer* buffer)
```

This function sends an encoded buffer on the designated PCO. When sending, the function will block until the entire buffer have been sent, or an error has occurred. This is simply because it is preferred to see the send operation as an atomic operation without possible race conditions.

Receiving a Message

```
GciStatus ACMReceive(GciPcoID pco_id, GciBuffer* buffer)
```

This function tries to receive a message from the designated PCO. If successful, the received encoded byte stream is inserted to the provided buffer, which is allocated inside the function.

Retrieving the Position of the First PCO with Received Data

`GciPosition ACMGetReceivedPCOPos()`

This function returns the position of the first PCO to have data ready for receiving. The return value is simply meant to be used as an iterator argument to the `ACMGetNextReceivedPCO()` function. If the returned value is zero, there are at this time no PCO that has received data.

Retrieving the position of the next PCO with received data

`GciPCOID ACMGetNextReceivedPCO(GciPosition* position)`

This function returns the numeric identifier of the PCO that has data ready to receive, and updates the *position* variable to point to the next PCO with data to receive. If the *position* is zero, no further PCO has data to receive.

A simple loop that checks all PCO's for receive can be written in a similar manner to the timer iteration described below in "A Loop That Fetches All Expired Timers and Prints Their ID Numbers" on page 1498 in chapter 37, *Adaptation of Generated Code*.

Timer Primitives

Timer handling can be implemented in various ways. The package makes it easy for the adaptation writer to use timers, by encapsulating functionality that used to be implemented in a fairly standard way in GciSnapshot() implementations over and over again. Instead of using cryptic timer structures the user can now use the GCI identifier for the needed timer in a number of timer access functions. Time itself is fetched from the system clock, normalized to zero at the time of starting the execution.

Starting a Timer

`GciStatus ACMStartTimer(GciTimerID timer_id,`
`GciTime timeout)`

This function starts the designated timer with the given expiration time. If the timer has not yet been created, this call will create it. If the timer was running or expired it is simply restarted.

Cancelling a Timer

`GciStatus ACMCancelTimer(GciTimerID timer_id)`

This function transfers the designated timer from the active list to the list of stopped timers. This call needs to be performed once a timer has been noted as expired (via `GciTimeout()` in `GciSnapshot()` for instance)

Cancelling All Timers

`GciStatus ACMCancelAllTimers()`

This function cancels all timers, even if they are already expired.

Reading the Value of a Timer

`GciStatus ACMReadTimer(GciTimerID timer_id, GciTime* timer_before_timeout)`

This function places the current time of a given timer in the provided GciTime object.

Checking the Current Status of a Timer

`GciStatus ACMTimerStatus(GciTimerID timer_id, GciTimerStatus* timer_status)`

This function places the current status of the given timer in the status variable. Please refer to the declaration of GciTimerStatus for the various states of a timer.

Retrieving the Key to the First Timer That Has Expired

`GciPosition ACMGetTimedOutPos()`

This function retrieves the position to the first timer that has expired from a chronological list of expired timers. The return value is simply meant to be used as an iterator argument to the `ACMGetNextTimedOut()` function. If the returned value is zero, there are no currently expired timers.

Getting the Next Expired Timer

`GciTimerID ACMGetNextTimedOut(GciPosition* position)`

Given a position index (the index to the first timer is fetched with the `ACMGetTimedOutPos()` function), this function returns an expired timer and updates the position variable to point to the next timer in the list of expired timers. If the position value is equal to zero after this call, no more expired timers exist in the list.

Example 279 A Loop That Fetches All Expired Timers and Prints Their ID Numbers

```
. . .
GciPosition pos = ACMGetTimedOutPos()

while (pos != NULL) {
  printf("Timer %d has expired!\n",
         ACMGetNextTimedOut(&pos) );
}
```

Time Left Before the Next Timer Expires

`GciStatus ACMTimeLeft(GciTime *time_before_timeout)`

This function retrieves the actual time left before the next timer is due to expire. If one or more timers have already expired, the function returns zero as the time left. If no timers exist, or all timers have been stopped, the function returns GciNotOk.

Timer and PCO Handling with a Single Call

`GciStatus ACMSnapshot()`

For the hardened writer of many adaptations, this function offers a relief. Calling this function will wrap most GciSnapshot functionality into one call. Briefly, what the function performs internally is the following:

1. Check all PCO/CP channels for received data.
2. For each queue that contains fresh data, receive, decode it and call GciReceive() with the result.
3. For each expired timer, call GciTimeout() with the expired timer’s ID number, and move the timer to the list of stopped timers.
4. Return status of the operations.

In effect, the simplest snapshot implementation can now look like this:

Example 280 The Simplest Snapshot in the World!

```
GciStatus GciSnapshot()
{
  return ACMSnapshot();
}
```

Somewhere in the code, before calling ACMSnapshot() for the first time, it is also necessary to register three functions with ACM:

- GciReceive()
- GciTimeout()
- A decode function.

This is due to the fact that ACMSnapshot() will possibly try to call these functions depending on the internal state. A good place to register these functions is close to the ACMInit() call:

```
.
.
ACMInit(max_timeout, ... );
.
.
ACMRegisterTimeoutHandler(&GciTimeout);
ACMRegisterReceiveHandler(&GciReceive);
ACMRegisterDefaultDecodeHandler(&DecoderFunction);
```

Waiting for the Next Event

`GciStatus ACMWaitForEvent()`

If more functionality is needed in the snapshot function, a mechanism for waiting for the correct amount of time is needed. This function sleeps until the next timer is due, or until data arrives on a PCO, whichever comes first, and returns with status GciOk, at which point the snapshot function can simply poll the input queues for data, and/or take the now arrived timeout. If no timers are running, the function returns after the maximum time to wait as defined in **ACMInit(...)**. If this time is set to a small amount, we have a polling snapshot in effect.

Example 281: The Same Snapshot but with Written Out Code ––––––

```
GciStatus GciSnapshot()
{
  GciPosition expired_pos;
  GciTimerID  expired_timer;
  GciPosition received_pos;
  GciPCOID    received_pco;
  GciBuffer   buffer;
  GciValue*   value;

/* Sleep until either a timer has expired
 * or data is received. For a polling
 * snapshot, remove this. */
  if (ACMWaitForEvent() != GciOk) {
    return GciNotOk;
  }

/* First, check all timers. */
  if (ACMSynchronizeTimers() != GciNotOk) {
    return GciNotOk;
  }
  expired_pos = ACMGetTimedOutPos();

  while(expired_pos != NULL) {
    expired_timer =
       ACMGetNextTimedOut(&expired_pos);
    GciTimeout(expired_timer);
    if (ACMCancelTimer(expired_timer) != GciOk) {
      return GciNotOk;
    }
  }
/* Next, take care of the PCO’s */

  buffer.buffer =
   (char *)malloc(sizeof(char) *
                 MAX_ENCODING_BUFFER+1);
  buffer.current_length = 0;
  buffer.max_length = MAX_ENCODING_BUFFER;

  received_pos = ACMGetReceivedPCOPos();

  while(received_pos != NULL) {
    received_pco =
      ACMGetNextReceivedPCO(&received_pos);
    if (ACMReceive(received_pco, &buffer) != GciOk)
    {
      return GciNotOk;
    }
    value = GeneralDecode(&buffer);
    if (value != NULL) {
      if (GciReceive(received_pco, value)!= GciOk) {
        return GciNotOk;
      }
    }
```

```
        else {
          fprintf(stderr,
                  "Transfer syntax error on PCO %s\n",
                  GciGetPCOName(received_pco) );
          return GciNotOk;
        }
      }
      return GciOk;
    }
  -----------------------------------------------------------
```

Retrieving the Current System Time

`GciStatus ACMCurrentTime(GciTime* current_time)`

This function retrieves the time of the system clock in the instance of the function call.

Error Handling

The error handling of the Adaptation Framework will be straightforward to those familiar with the *err_no* paradigm of C. Every time an error occurs, the function sets an internal error code and returns *GciNotOk*. The error code can then be retrieved either as an enumerated value, or as a descriptive string.

The error codes can be found in the file *acm.h*.

Setting the Error Code

`ACMSetLastError(ACM_Error_Number errornumber)`

Sets the error number to the given value.

Retrieving the Error Code of the Last ACM Error

`ACM_Error_Number ACMGetLastError()`

Returns the last error code as an enumerated value.

Retrieving the Error String of the Last ACM Error

`const char* ACMGetLastErrorMessage()`

Returns the last error code as a printable string.

Completing the Adaptation

The generated code has to be extended before it is complete in order to test the intended implementation. This section describes how to make these extensions.

Figure 276 displays in a simple way the anatomy of an executable test suite.

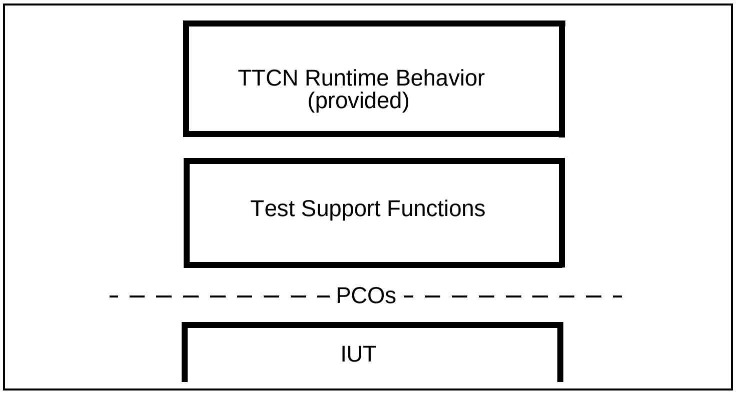

Figure 276: The anatomy of an executable test suite

When code is generated by the code generator, it does not know anything about the system it is about to test. It assumes that it will have access to certain functions, implemented by the user. This is the "Test Support Functions" module in Figure 276.

One important thing to remember is that the previously defined interface called GCI, is a standardized set of functions.See "The GCI Interface" on page 1450 in chapter 37, *Adaptation of Generated Code*. **Adaptation should be made using the functions and data types defined by that interface**.

The Test Support Functions

This is the "glue" between the TTCN Runtime Behavior and the IUT. It is a set of functionality (functions) that is adaptation specific and should be provided by the user.

The following areas have to be covered and they are described in more detail in the sections they refer to.

Implementation and Handling of Timers

The timers defined in the Test Suite must have a real representation in the test environment. The TTCN Runtime Behavior will, when necessary, ask the adaptation about the status of a timer. See “Representation and Handling of PCO and CP Queues” on page 1506 in chapter 37, *Adaptation of Generated Code*.

Representation and Handling of PCOs [and CPs]

Messages are communicated through *Points of Control and Observation* and *Connection Points*. The actual buffering of incoming IUT messages must be supplied by the adaptation. See “Representation and Handling of PCO and CP Queues” on page 1506 in chapter 37, *Adaptation of Generated Code*.

Communication with Test Equipment [and PTCs]

To be able to test the IUT at all, there must be actual communication channels to and from it. The actual communication is of course totally depending on the system environment. See “IUT Communication” on page 1505 in chapter 37, *Adaptation of Generated Code*.

Encoding and Decoding

Values in the test suite (constraints, variables, constants, etc.) must be properly encoded and decoded for sending them to the IUT. The actual protocol for encoding decoding is up to the user. See “Encoding and Decoding” on page 1507 in chapter 37, *Adaptation of Generated Code* and “Encoding and Decoding Using BER” on page 1514 in chapter 37, *Adaptation of Generated Code* for more details.

Implementing the Adaptor the Efficient Way – the Adaptation Framework

The implementation of timer handling and communication primitives can be made by calls to the Adaptation Framework API, which permits the underlying “hard” protocol and timer implementation to change without having to change the high-level adaptation code. See “The Adaptation Framework” on page 1491 in chapter 37, *Adaptation of Generated Code* for a thorough explanation.

Timers

Timers need some special attention. As timers are implemented differently on different systems the implementation of the timers might differ. See the *timers* adaptation template in the installation for ideas.

Timers are simply a number of constructions to keep track of each of the test suite timers. In the generated code a timer is represented by an integer descriptor which uniquely identifies it. The timer implementation supplied with the Adaptation Framework implements the timeout functionality by maintaining an ordered list of the running timers, which means that retrieving the time left to timeout and the next timer to expire are quick operations.

To refresh the current status of timers, the timer lists need to be *synchronized* to the system time. This is made automatically by the snapshot function `ACMWaitForEvent()` (more on this function later) or can be made explicitly by a call to the function `ACMSynchronizeTimers()`. This function simply checks all running timers with the system clock, moving timers that are due to the list of expired timers.

There are four functions to consider regarding timers:

```
GciStartTimer

GciCancelTimer

GciReadTimer

GciSnaphot
```

The interfaces to timers that can (and should) be called in the TTCN Runtime Behavior are:

```
GciTimeout

GciGetNoOfTimers

GetTimer

GciGetTimerName

GetTimerIndex
```

For details concerning these functions, see “GCI C Code Reference” on page 1465 in chapter 37, *Adaptation of Generated Code*.

The Adaptation Framework interfaces to timer functionality that can be used to implement the GCI functions are:

```
ACMStartTimer

ACMCancelTimer

ACMCurrentTime

ACMGetTimedOutPos

ACMGetNextTimedOut

ACMReadTimer

ACMTimeLeft

ACMTimerStatus

ACMWaitForEvent
```

These functions map to a real-time timer implementation that is provided with the installation of TTCN suite and can be used "as-is". See Release Notes for details on which platforms are supported by this timer package.

Timer Adaptation Example

The *timers* adaptation template is the simple adaptation template with timers implemented. It demonstrates in general how to use the Adaptation Framework to implement timer functionality, and in particular the use of the call `ACMSynchronizeTimers()` that must be used in snapshot when `ACMWaitForEvent()` can't be used.

IUT Communication

From an abstract point of view, sending and receiving is done over PCOs (Points of Control and Observation). The physical representation of these PCOs has to be defined by the user. It can be shared memory, serial communication, sockets, etc. This, of course, is only done on the controlling side. The PCOs have to be connected somewhere to the test equipment and the responsibility for this is put upon the user.

The GCI functions that must perform the communication is (at least):

```
GciSend
```

```
GciReceive
```

For details concerning these functions, see "GCI C Code Reference" on page 1465 in chapter 37, *Adaptation of Generated Code*.

Representation and Handling of PCO and CP Queues

PCOs are constructions to handle the PCO queues. Each PCO should have buffers for sending and receiving, a method for retrieving the status of the receive buffer and additional information such as channels and ports must be provided for the physical channels.

If the ETS is to run within the test equipment, i.e. the communication between the ETS and the IUT resides within the test equipment, the ETS has to be moved (cross compiling or if possible compiling within the test equipment).

This queue initialization should of course be made before any test case are run. The function that concerns the PCO's and CPs are:

```
GciSnapshot
GciSend
```

`GciSnapshot()` can be implemented using for instance `ACMWaitForEvent()`, and `GciSend()` can be implemented using `ACMSend()`. Today, the ACM layer builds upon a TCP/IP socket implementation but other protocol implementations can be easily added.

The interface that should be used when a message has been received (after proper decoding) in the TTCN Runtime Behavior is:

```
GciReceive
```

Note that if the ACM layer function `ACMWaitForEvent()` is used in the snapshot function, the user never has to call `GciReceive()` since this is done automatically from within `ACMWaitForEvent()`!

For details concerning these functions, see "GCI C Code Reference" on page 1465 in chapter 37, *Adaptation of Generated Code*.

Encoding and Decoding

tBuffer is the subsystem of UCF that is used to represent values of ASN.1 types in the transmitted bit pattern. This section describes the interface for tBuffer.

Functions

To invoke functions of tBuffer, the following macros are to be called:

Initialization of buffer

You should initialize tBuffer before using it.

```
void BufInitBuf(tBuffer buf, tDirectionType dirtype,
UCFStatus status)
```

tBuffer buf	Buffer for data storage
tDirectionType dirtype	Data writing order (DIRECT or REVERSE).
UCFStatus status	Status of initialization. Variable “status” is equal to UCF_Ok if no errors were encountered during initialization or an error code otherwise.

Note: Buffer direction mode

`REVERSE` modes is not supported in the buffers anymore, although the prototype of the `BufInitBuf` function still includes the `dirtype` parameter for backwards compatibility. The direction mode parameter is always ignored when calling the `BufInitBuf` function.

Closing of buffer

You should close tBuffer after working with it.

```
void BufCloseBuf(tBuffer buf)
```

tBuffer buf	Buffer

Initialization in write mode

This function initializes tBuffer in write mode. You can write data to the buffer after calling this function.

```
UCFStatus BufInitWriteMode(tBuffer buf)
```

Initialization in read mode

This function initializes tBuffer in read data. You can read data from the buffer after calling this function.

```
UCFStatus BufInitReadMode(tBuffer buf)
```

Closing write mode

This function closes write mode of tBuffer. You should call this function after the ‘write to buffer’ operation is completed.

```
void BufCloseWriteMode(tBuffer buf)
```

Closing read mode

This function closes read mode of tBuffer. You should call this function after reading from the buffer is finished.

```
void BufCloseReadMode(tBuffer buf)
```

Getting direction type of the buffer

This function returns the direction type of tBuffer.

```
tDirectionType BufGetDirType(tBuffer buf)
```

Note: No REVERSE mode

`REVERSE` mode is not supported in the buffers anymore, although the `BufGetDirType` function is still present in the buffer interface for backwards compatibility. It will always return `DIRECT` mode.

Creating copy of buffer

This function makes a copy of the buffer.

```
UCFStatus  BufCopyBuf(tBuffer dst, tBuffer srs)
```

tBuffer dst	destination buffer
tBuffer src	source buffer

Getting the data byte length

This function returns the buffer’s data length in bytes.

```
tLength  BufGetDataLen(tBuffer buf)
```

Getting the data bit length

This function returns the buffer’s data length in bits.

```
tLength  BufGetDataBitLen(tBuffer buf)
```

Getting the available length

This function returns the maximal data length you can write into the buffer in one piece (in bytes).

```
tLength  BufGetAvailableLen(tBuffer buf)
```

Getting byte

This function reads one byte from the buffer. The buffer should be initialized in read mode.

```
unsigned char  BufGetByte(tBuffer buf)
```

Getting data segment

This function reads a data segment from the buffer. The buffer should be initialized in read mode.

```
unsigned char*  BufGetSeg(tBuffer buf, tLength
seglen)
```

tBuffer buf	Buffer to read from
tLength seglen	Requested data length

Peeking byte

This function peeks (reads, but does not remove) one byte from the buffer. The buffer should be initialized in read mode.

```
unsigned char  BufPeekByte(tBuffer buf)
```

Peeking data segment

This function peeks (reads, but does not remove) a data segment from the buffer. The buffer should be initialized in read mode.

```
unsigned char*  BufPeekSeg(tBuffer buf, tLength
seglen)
```

tBuffer buf	Buffer to read from
tLength seglen	Requested data length

Putting byte

This function writes one byte into the buffer. The buffer should be initialized in write mode.

```
void  BufPutByte(tBuffer buf, unsigned char byte)
```

tBuffer buf	Buffer to write into
unsigned char byte	Byte to write into buffer

Putting data segment

This function writes a data segment into the buffer. The buffer should be initialized in write mode.

```
void  BufPutSeg(tBuffer buf, unsigned char* data,
tLength seglen)
```

Putting bit

This function writes one bit into the buffer. The buffer should be initialized in write mode.

```
void BufPutBit(tBuffer bud, unsigned char bit)
```

tBuffer buf	Buffer to write into
unsigned char bit	Value of bit (0 or 1)

Getting bit

This function reads one bit from the buffer. The buffer should be initialized in read mode.

```
unsigned char BufGetBit(tBuffer buf)
```

Putting bits

This function writes bits into the buffer. The buffer should be initialized in write mode.

```
void BufPutBits(tBuffer buf, unsigned char bits,
unsigned char num)
```

tBuffer buf	Buffer to write into
unsigned char bits	Sequence of bits

unsigned char num	Number of bits in sequence (num <= 8)

Getting bits

This function reads bits from the buffer. The buffer should be initialized in read mode.

```
unsigned char BufGetBits(tBuffer buf, unsigned char
num)
```

tBuffer buf	Buffer to read from
unsigned char	Number of bits (num <= 8)

Putting padding bits

This function writes padding bits in PER ALIGN variant of encoding. The buffer should be initialized in write mode.

```
void BufPutAlign(tBuffer buf)
```

Getting padding bits

This function reads padding bits in PER ALIGN variant of encoding. The buffer should be initialized in read mode.

```
void BufGetAlign(tBuffer buf)
```

Set encoding variant (PER only)

This function sets the encoding variant.

```
void BufSetEncVar(tBuffer buf, UCFEncVariant encVar)
```

UCFEncVariant encVar	Encoding variant (UCF_Align, UCF_Unalign or UCF_NoEndPad)

Get encoding variant (PER Only)

This function returns the encoding variant.

```
UCFEncVariant  BufGetEncVar(tBuffer buf)
```

Setting up error catcher

This function sets up error catcher. Returns 0 or code of error if any.

```
unsigned int BufSetCatcher(tBuffer buf)
```

Getting error mode

This function gets error mode (bem_Off or bem_On).

```
tBufferErrorMode BufGetErrorMode(tBuffer buf)
```

Supported ASN.1 Types

- BOOLEAN
- INTEGER
- ENUMERATED
- REAL
- OBJECT IDENTIFIER
- NULL
- BIT STRING
- OCTET STRING
- IA5String
- NumericString
- PrintableString
- VisibleString
- UTCTime
- GeneralizedTime
- SEQUENCE
- SET
- SEQUENCE OF
- SET OF
- CHOICE
- Tagged types
- Open types

Encoding, and in Particular Decoding within ACM and GCI

The standard way to implement the encoder and decoder functions are to put at least one function of each in the file *encoder.c* and declare them in the file *encoder.h*. With every adaptation that is provided with the TTCN suite installation are those files included, often with empty encode/decode functions where the user is supposed to enter the appropriate functionality. The functions should be of the following structure:

The Encoding Function

```
GciStatus Encoder_<name>(const GciValue*,
                          GciBuffer** )
```

If encoding was successful, the GciBuffer should be loaded with the encoded data and *GciOk* returned by the function. Otherwise, the function should return *GciNotOk*.

The Decoding Function

```
GciStatus Decoder_<name>(const GciBuffer*,
                         int*,
                          GciValue**)
```

If decoding was successful, the GciValue should be loaded with the decoded GCI value, the integer should contain the number of bytes that were consumed in the decoding process and the return status *GciOk*. Upon failure, the function should simply return *GciNotOk*.

Note that several encode/decode functions can exist, and the user can switch between them at will, either by calling different functions from *GciSnapshot()* or by registering different functions prior to calling *ACMSnapshot()*. See *“Registering the Active Decode Function” on page 1494 in chapter 37, Adaptation of Generated Code* for details.

General

A test suite has no knowledge of the encoding and decoding rules of the actual application protocol. The definition of signal components and the description of the signal flows are done in an abstract and high-level manner. The physical representation of the signal components and the definition of the actual transfer syntax is not defined within the test suite.

The encoding and decoding rules (functions) simply define a common transfer syntax between the test equipment and the executable test suite.

It is up to the user to write his/her own encoding and decoding rules using the GCI value representation. Even if the TTCN to C Compiler comes with an adaptation template that includes a general encoder and decoder, these rules can not be used at all times.

For test applications that need to send the same type of messages back and forth through a communication channel, the encoding and decoding functions must be related to each other in such way that the decoding function is the inverse function of the encoding function. This gives the following simple rule:

```
Message = Decode( Encode ( Message ) )
```

This simply states, that if you decode an encoded message, you will get the original message back.

For applications that send and receive messages of different types (for example an application sending commands to an interface one way and receiving command results the other way), the encoding and decoding rules might not be related at all.

It is up to the user to identify how he/she needs to encode and decode messages to successfully be able to communicate with his/her test equipment.

Encoding and Decoding Using BER

TTCN Suite can generate encode/decode function definitions, that give an opportunity for representing values of each ASN.1 type as a string of eight-bits octets and transfer them between the environment and the ETS. TTCN Suite now have support for BER (Basic Encoding Rules) standard, defined in X.690 for the subset of ASN.1 that is defined by TTCN.

BER Encoder/Decoder Support Library

The BER support comes in the form of a static library that supports encoding/decoding functionality of ASN.1. Profiles of all the functions that a user can use are located in the `ucf.h` in the static files directory. This file should be included in the adaptation and the library should be linked together with the adaptation.

The library contains two basic functions that provide functions for encoding/decoding types, computing length of values and buffer handling procedures: `BEREncode(...)` and `BERDecode(...)`.

How to Use BER Support in the TTCN Suite?

First, select *Generate BER encoders/decoders* in the Make options dialog. A file will be generated called `asn1ende.h` in your target directory. This file defines the encoding and decoding functions for the ASN.1 definitions in the test suite.

Second, you should implement the usage of the encoding functions in your `adaptor.c`. This is done in the `GciSend` and `GciSnapshot` functions. Also, you should define buffers and initialize them (this can be done in `main`.

The generated file <ModuleName_of_ASN.1-specification>_asn1icoder.h contains the encoding/decoding function definitions for ASN.1 objects in the Test Suite. For example, if you have ASN.1 ASP Type definition by name of `ASPType1`, two functions will be generated:

```
UCFStatus Encode_ASPType1(tBuffer, GciValue*);
```

```
UCFStatus Decode_ASPType1(tBuffer, tLength*, GciValue**);
```

The first parameter of the encode function is the buffer where the encoded data is placed. The decoding is made from the buffer to a GCI value.

The encoding/decoding functions are specific for the ASP/PDU types in question so when calling them in `GciSend` and `GciSnapshot`, you need to make sure that they are called with a value of the correct ASP/PDU type. One way to get around this problem is to define a single ASP/PDU type that is a CHOICE of all types that you are sending or receiving in your test suite.

Example

For example, we have a test suite with ASN.1 ASP Type

```
ASPType1 ::= INTEGER
```

And ASN.1 ASP constraint of ASPType1 named

```
ASPCon1 ::= 5
```

We have PCO named PCO1 and dynamic behavior like this:

```
PCO1 ! ASPType1     | Constraint: ASPCon1
PCO1 ? ASPType1     | Constraint: ASPCon1
```

And we want to use BER for encoding/decoding. After code generation we have a <ModuleName_of_ASN.1-specification>_asn1icoder.h file with the following definitions:

```
tLength Encode_ASPType1(tBuffer, GciValue*);
tLength Decode_ASPType1(tBuffer, GciValue**);
```

So in our adaptation file we need to declare, initialize and close BER buffers:

```
tBuffer InBuffer;
tBuffer OutBuffer;
.
. (rest of the adaptation)
.
int main(int argc, char* argv[])
{
   BufInitBuf(InBuffer, DIRECT, status);
   BufInitBuf(OutBuffer, DIRECT, status);
   .
   . (ETS control)
   .
  BufCloseBuf(InBuffer);
  BufCloseBuf(OutBuffer);
  return 0;
}
```

In `GciSend()` we should initialize buffer in `WriteMode` and encode value:

```
GciStatus GciSend( int pcod, GciValue* object )
{
     UCFStatus status;
   BufInitWriteMode(OutBuffer);
   .
   .
  Encode_ASPType1(OutBuffer, object);
  status = Encode_ASPType1(OutBuffer, object);
  if(status != UCF_Ok)
  {
   UCFPrintErrorMessage(stderr,status);
   /* warning or exit */
  }
  .
  .
  BufCloseWriteMode(OutBuffer);
  return GciOK;
}
```

In `GciSnapshot()` we should initialize buffer in `ReadMode` and decode value:

```
GciStatus GciSnapshot( )
{
  UCFStatus status;
  tLength DecLength;
```

```
     BufInitReadMode(InBuffer);
     .
     .
     Decode_ASPType1(InBuffer, &result);
    status = Decode_ASPType1(InBuffer, &DecLength,
   &result);
    if(status != UCF_Ok)
    {
     UCFPrintErrorMessage(stderr,status);
     /* warning or exit */
    }
     .
     .
     BufCloseReadMode(InBuffer);
     return GciOK;
   }
```

How to Use the PER Support in the TTCN Suite

The PER support is used almost exactly like the BER support, so read above first to learn how to use the BER support. The PER Encoding/Decoding functions have been changed and now return a UCFStatus value:

```
    UCFStatus Encode_ASPType1(tBuffer, GciValue*)

    UCFStatus Decode_ASPType1(tBuffer, tLength*,
   GciValue**)
```

The primary difference to the BER support is that the PER support includes supporting different variants of PER. The variants supported are Unalign, Align and NoEndPad with the Unalign variant being the default. The Unalign and Align variants are, just like BER, eight-bit octet-oriented, but the NoEndPad variant is bit oriented which necessitates use of the bit-oriented access to the buffer instead of the byte-oriented access.

The mechanism to select PER encoding variants is based on the use of pre-processor symbols. By defining the symbol `UCF_PER_DEFAULT_ENCODING_VARIANT` to the value associated with the selected encoding variant before including the `asn1ende.h` file, all types will get the selected encoding variant unless explicitly overridden.

Overriding the default encoding variant can be done on a per type basis by defining the pre-processor symbol `EncodingVariant_<type name>` to the selected variant value before including the `asn1ende.h` file.

The following example shows a snippet of the `adaptor.c` file in a situation where all types but the type `Type1` is to be encoded/decoded with the Align variant and `Type1` is to use the Unalign variant.

Example 282: Selecting PER encoding variant

```
#define UCF_PER_DEFAULT_ENCODING_VARIANT UCF_Align
#define EncodingVariant_Type1 UCF_Unalign
#include <asn1ende.h>
```

The Adaptation Framework

To make it even easier to connect the generated code to “the real world”, the Adaptation Framework (ACM) is introduced, a platform-independent API that encapsulates communication and timer handling provided by *plug-in components.* These components can be communication protocol implementations, simulated timer modules, etc. and can easily be replaced *without having to rewrite the adaptation* – provided the Adaptation Framework has been used to implement the GCI functions in the adaptor.

Currently, the TTCN suite is delivered with the following plug-in modules for the framework:

- A TCP/IP socket communication implementation.
- A system-time timer package.

This means that if the adaptation is written using the Adaptation Framework, the generated code can instantly be used with TCP/IP communication. As the framework also hides all internal mechanisms of the runtime system, the resulting adaptation will be easier to maintain, more general and much smaller in size.

Example 283: An Implementation of GciSend with ACM

```
GciStatus GciSend (int pcod, GciValue* msg)
{
  GciBuffer  encoded_value;
  GciStatus  status;

  encoded_value.buffer = (char *) malloc(
sizeof(char) * MAX_ENCODING_BUFFER + 1);
  encoded_value.current_length = 0;
  encoded_value.max_length = MAX_ENCODING_BUFFER;
```

```
    status = Encode(&encoded_value, msg);
    if (status != GciOk) {
      fprintf(stderr, "%s\n",
              ACMGetErrorMessage(ACMGetLastError()));
      return GciNotOk;
    }
    status = ACMSend(pcod, &encoded_value);
    free(&encoded_value);

    if (status != GciOk) {
      fprintf(stderr, "%s\n",
              ACMGetErrorMessage(ACMGetLastError()));
      return GciNotOk;
    }
    return status;
  }
```

Adaptation Templates

An empty adaptation is copied to the code directory if the user has no prior adaptation.

It is up to the user to implement the functions in the GCI operational interface. These functions are called from the TTCN runtime behavior and should not be removed even if they are empty.

The function bodies and declarations are found in the empty adaptation files, but the function bodies are empty. They only contain a print statement about the function not being implemented.

The *ACM* adaptor also uses the Adaptation Framework for communication and timer handling and is a good example to study on how to use the Adaptation Framework to implement GCI functions.

Auxiliary Adaptation Functionality

This section describes some extra functionality which is included in the adaptation templates.

`FILE* logStream;`

This is the stream to where log messages are written. The default value is **stdout** and is set in the `main` function, but can be set to whatever stream the user wishes to use.

`extern const int Gc<tablename>D = {int}`

Constant numbers for all tables (test case table, PCO table, timer table, etc...) in TTCN. Used to search in the IcSymTab array (see symbol table below). An example is for **`GcTestCaseD = 517`**.

The simple adaptation template (in the installation) includes all empty functions for the previously described GCI operational functions to be defined by the user.