SAE TECHNICAL PAPER SERIES

2007-01-0504

Automated Real-Time Testing of Electronic Control Units

Holger Krisp, Klaus Lamberg and Robert Leinfellner
dSPACE GmbH

Reprinted From: In-Vehicle Software & Hardware Systems, 2007 (SP-2126)

ISBN 0-7680-1633-9

**2007 World Congress
Detroit, Michigan
April 16-19, 2007**

SAE International™

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-0790 Web: www.sae.org

All SAE papers, standards, and selected books are abstracted and indexed in the Global Mobility Database.

ISSN 0148-7191

Printed in USA

2007-01-0504

Automated Real-Time Testing of Electronic Control Units

Holger Krisp, Klaus Lamberg and Robert Leinfellner
dSPACE GmbH

ABSTRACT

Today, hardware-in-the-loop (HIL) simulation is common practice as a testing methodology for electronic control units (ECUs). An essential criterion for the efficiency of an HIL system is the availability of powerful test automation having access to all of its hardware and software components (including I/O channels, failure insertion units, bus communication controllers and diagnostic interfaces). The growing complexity of vehicle embedded systems, which are interconnected by bus systems (like CAN, LIN or FlexRay), result in hundreds or even thousands of tests that have to be done to ensure the correct system functionality. This is best achieved by automated testing.

Automated testing usually is performed by executing tests on a standard PC, which is interconnected to the HIL system. However, higher demands regarding timing precision are hard to accomplish. As an example, ECU interaction has to be captured and responded to in the range of milliseconds. Such hard real-time constraints require the tests to be executed in real-time.

There is a need for a concept to execute tests in real-time on the processor board of the HIL simulator synchronously with the execution of the simulation model. The test description must be easy to learn and must provide a powerful means for software reuse, e.g. through libraries and object-orientation. Additionally, it must be possible to access all of the HIL components, such as hardware, real-time-model, communication buses, and diagnostic interfaces from within real-time tests.

In this paper we present a real-time testing concept meeting such demands. By applying this method, it is possible to implement sample-time-precise, highly reproducible tests with deterministic functional and timing behaviour. Such tests can be written using Python as a standard object-oriented scripting language, and executed in real-time without the need to modify and recompile the real-time model. Thus, the paper provides a new quality in automated testing of ECUs by introducing real-time execution as a core concept.

INTRODUCTION

The importance of electronics for the automotive industry is steadily increasing. Electronics and software are significant innovating factors in automotive technology. On the other hand, the amount of software and functionality is also increasing, and so is its complexity.

However, assuring the required quality of such complex systems is one of the main challenges in automotive development today. Testing plays a key role in quality assurance. In this paper, testing means to execute a system in order to find failures in the system. The most common testing method for electronic control units is hardware-in-the-loop (HIL) simulation. This means that one or more ECUs (units under test) are connected to a simulation system running models of the physical environment of the ECUs such as engine, transmission, vehicle dynamics etc. in real-time. This enables entire test scenarios for the ECUs to run in a software controlled manner, e.g. in a laboratory. Using hardware-in-the-loop simulation has several advantages. The main ones are as follows:

- Controller functions can be tested in the early stages of development, even before a test carrier (prototype vehicle) has been produced.
- Simultaneous development of ECU and vehicle, necessary to cut the overall development time, is possible with HIL simulation.
- Expensive field trials or experiments in borderline zones and hazardous situations can partly be replaced by laboratory or desktop experiments.
- Extreme or unusual ambient conditions can be adjusted very easily by varying the parameters in the model. Thus typical winter test drives on snow and ice can be carried out in summer, and cold-start tests can be performed repeatedly without having to wait for the engine to cool down.
- Failures and errors that could have devastating effects in a real vehicle can be simulated and tested without danger.
- Experiments performed on the HIL system can be automatically and precisely repeated as often as required.

Today, HIL simulation and testing is a well established testing method for automotive electronics at almost all automotive manufacturers and suppliers worldwide and has become a mandatory milestone in most automotive electronics development processes. At the same time, the demands for the simulation and testing quality in terms of simulation and timing precision as well as reactivity and repeatabilty of the automated tests has significantly increased over the years. Therefore, there is a clear need for a transition from today's way of automated testing, which is mostly done PC-based in non-real-time, to a new approach enabling test execution in real-time.

HARDWARE-IN-THE-LOOP-SIMULATION AND TESTING

HARDWARE-IN-THE-LOOP TEST SYSTEMS

Figure 1 shows the fundamental structure of HIL systems [1]. Instead of being connected to an actual vehicle, the ECU to be tested is connected to a simulation system. This runs models of the vehicle process and associated sensors and actuators on real-time simulation hardware. The real-time hardware is connected to the ECU's electrical interface via special I/O boards and suitable signal conditioning for level adjustment, using either simulated or real loads.

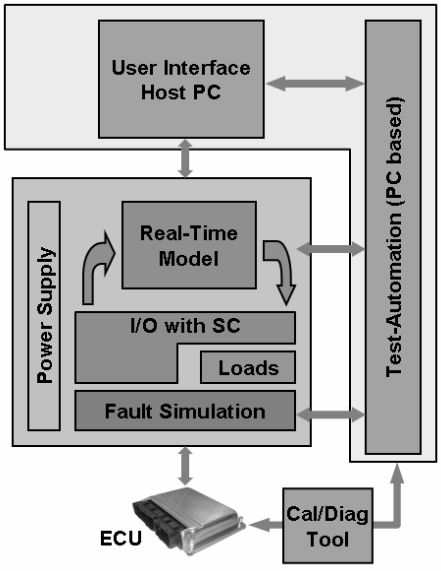

Figure 1: Typical hardware-in-the-loop system architecture

AUTOMATED TESTING

Automatic testing increases the benefit of HIL simulation significantly. For example, it allows testing overnight and on weekends ("lights-out-tests") and reduces the overall time needed for testing. In one case, automatically performing diagnostic tests for transmission ECUs using HIL technology decreased the testing efforts by as much as a factor of ten [2]. Additionally, automated testing leads to broader test coverage and a greater test depth, meaning that more tests can be done and more details can be tested.

TEST INTERFACES

To be effective, it is not enough to develop automation concepts only for simple procedures. There must be a wide range of functionality that the automated procedure is able to perform. This functionality is mainly determined by the interfaces that can be accessed from within a test. There are many interfaces that are typically used for testing ECUs; the following are the most important ones.

Real-Time Model Access

A fundamental element of model-based testing is the interface between the test program and the simulation model. Typically, this interface is specified by model variables. The test can access the models of the simulation model directly to read and write values from and to the respective model variables. As an example, a test can stimulate model variables such as accelerator pedal, steering angle and gearshift synchronously in order to let the real-time model perform a specific driving maneouver. At the same time, the test can capture model variables such as engine and vehicle speed to evaluate the test result.

Electrical Failure Simulation Access

Relays are used to generate real electrical short circuits on the ECU input and output pins. The ECU diagnostic software is required to detect such faults and to correctly handle the fault (for example, by limp home functions) and to store the information about the failure in the ECU internal diagnostic memory. Typical electrical faults to be simulated using relays are short circuits to battery voltage and to ground or broken wires. It is even possible to simulate transition resistances or bleeding resistances, or to enable multiple faults at the same time.

Due to its uniform procedure, testing diagnostic functions has very great potential for saving time and cost. Therefore automatic failure simulation is provided by most HIL systems by the ability to remotely and automatically control the failure simulation relays. This requires an automation interface for sending switching commands, for example via the serial interface (RS232) or via the CAN bus.

Diagnostic Interface Access

A very common use case of HIL systems is testing the diagnostic functions of the ECU (onboard diagnostics). This requires the ability to access and read the diagnostic memory of the ECU. Therefore, a test needs access to a standard diagnostic tool. Using a standardized automation interface (ASAM MCD-3) diagnostic tools can be controlled from within a test and serve as an interface to the ECU diagnostic functionality.

TEST IMPLEMENTATION AND EXECUTION

The test procedure itself is implemented by a test program. The test program accesses the above described HIL test system resources through the appropriate interfaces and implements the control flow of the test, as well as the rules to determine the test verdict (such as “passed” or “failed”).

Different ways of implementing test programs are possible.

Script based test implementation

A very common approach for automated testing is to implement test programs in the form of an executable script. Different programming languages can be used therefore, such as Visual Basic or Python. Python is an object-oriented, open-source scripting language (www.python.org), which is quite commonly used for such purposes. User can simply write Python scripts in a text editor or using a standard Python development environment, and execute the scripts using a standard Python interpreter. Since users can define their own library modules this provides a maximum in terms of flexibility and extendability.

To have access to the previously described test interfaces from within a test script, APIs (application programming interfaces) must be available. For Python there exist several library modules, e.g. to access a real-time model running on an HIL simulator, access diagnostic tools, read and write ECU calibration parameters, control the electrical failure simulation hardware, remote control third party tools and much more [4].

Graphical test implementation

Sometimes, a scripting language does not meet user requirements regarding transparency, simplicity, readability and traceability. In such cases, graphical test descriptions are used. Such test descriptions can for example be based on UML (Unified Modeling Language, www.uml.org) activity or sequence diagrams. An example is given in Figure 2.

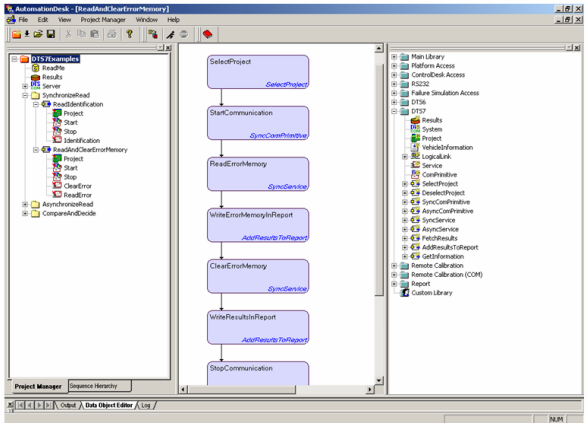

Figure 2: Graphical test representation in AutomationDesk [3]

The example shown in the screenshot consists of a series of different test step blocks being executed sequentially from top to bottom. Test interfaces in a graphical test automation environment are represented as blocks, as shown on the right hand side in Figure 2. There are different block libraries available representing function calls to the specific test interfaces.

Test Execution

As explained, test scripts can be simply executed on a PC using a script interpreter. For executing graphical test descriptions, there are generally two options available: the objects represented by the graphics can be interpreted and executed directly. Alternatively, it is possible to generate executable script code from the graphical representation.

However, executing scripts or graphical objects is performed on a standard PC using a conventional PC operating system. Since PC operating systems in general are not real-time capable, this leads to many limitations with respect to the timing precision, repeatability, timing resolution, and speed of the test program execution. To solve these problems, in conventional test automation environments, specific testing tasks such as stimulus signal generation and data acquisition are supported by service routines, running on the real-time processor of an HIL system. On the other hand this requires specific configuration instructions to instantiate such services, which are proprietary and lead to further limitations regarding flexibility and extendability.

Therefore, in the following a new approach is presented which combines the flexibility and powerfulness of a standard scripting language commonly known from PCs with the timing precision of a real-time system.

EMBEDDING A SCRIPT INTERPRETER ON A REAL-TIME PLATFORM

The new system architecture is shown in Figure 3. The Python interpreter is embedded in the real-time software environment. Thus, it has access to the real-time model variables and can read values from the model and write values to it. Also, it can be accessed from the test-automation environment hosted on the PC.

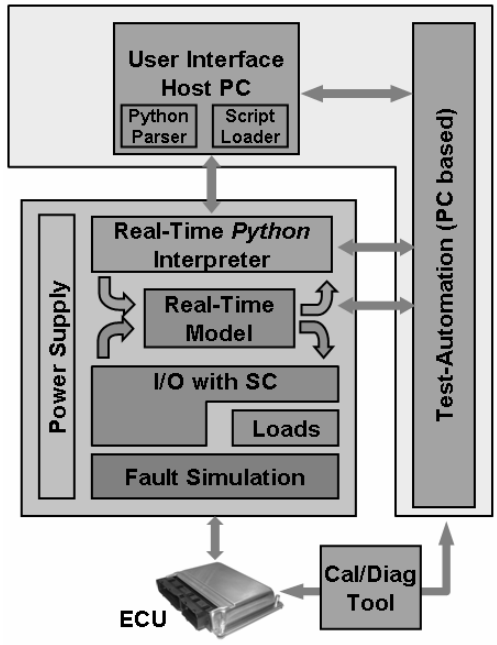

Figure 3: Hardware-in-the-loop system architecture with real-time script interpreter

The Concept of Time In Python Scripts

Real-time scripts have a different structure compared to model code. Since real-time model code is triggered and executed to completion as a whole in each sample step, a real-time script can contain very different sequential and concurrent tasks.

Therefore it is necessary to synchronize script execution with the sample steps of the real-time model, so that both use the same time base. This means, that for each sample step only those parts of the script which coincide with that sample step are executed. For this reason, the time dimension is introduced to the Python real-time scripts. This enables the real-time interpreter to determine, which script statement to execute in what sample step. This information is used for synchronizing the real-time script with the real-time model.

Function Hooks for Script Execution

To synchronize real-time scripts with real-time model execution, two function hooks (script service functions) are added to the control flow of the simulation base rate: one before the model outputs are calculated and one after. Figure 4 demonstrates this situation.

The first hook is the primary one, as reading and writing model variables (parameters and signals) are the primary operations at the interface between real-time model and real-time script. This hook allows scripts to stimulate parameters before new outputs are calculated. When multiple scripts are loaded, the hook functions for each script are subsequentially executed by the script service.

As shown in Figure 4 script execution takes place before the model outputs are calculated. In case of long scripts, the idle time before the end of a sample step can be exhausted, which leads to an overrun of the sample step. To solve this problem the execution time of real-time scripts is limited. When a script tries to execute beyond the limit, it is aborted with a user notification. This situation is called script overrun. A script overrun does not affect the execution of the real-time model.

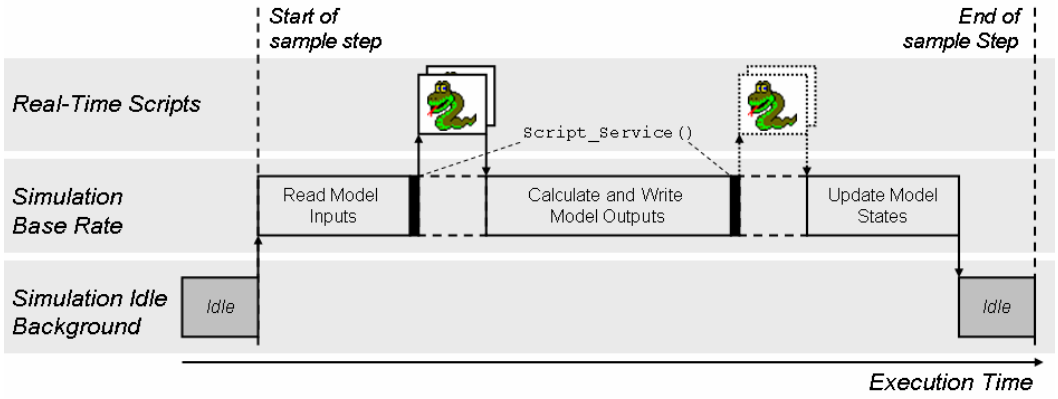

Figure 4: Real-time scheduling concept

IMPLEMENTING REAL-TIME SCRIPTS

With the implementation of the Python interpreter that synchronizes script execution with the real-time model execution, test actions can be described by the Python script and be performed on a real-time basis.

ACCESSING SIMULATOR VARIABLES

As described in a previous section, model variables of the real-time model have to be accessed by real-time scripts. This is necessary to stimulate and monitor the HIL simulator's hardware interfaces (like I/O channel, Electrical Failure Simulation, bus interfaces etc.) to test the connected ECU(s).

Access is done by a dedicated `variable` class contained in the module `rttlib`. Using this module, variable objects can be created by referencing the model path. The path is a unique reference to the element's position in the model hierarchy.

```
# import variable module
from rttlib import variable

# Create a variable object
Const1=variable.Variable(r'Model Root/Const1/Value')
```

Using a variable object, read and write can be done by means of the `Value` property of the variable object. If for example a test script has to increment the current value of a model variable by 1.0, the corresponding code fragment looks as follows:

```
# Increment the current value of Const1 by 1.0
Const1.Value += 1.0
```

Variable access is performed using information generated by the build process of the real-time model. Such information links the symbolic variables to the memory addresses of the processor board. Link data is compressed and downloaded to the real-time platform so that real-time scripts have access to all model variables without accessing the host PC during runtime.

When the user modifies the real-time model, he has to rebuild and download the new real-time application. When doing so, symbolic link information for test variable mapping is updated and transferred to the real-time platform automatically. The real-time scripts can remain unchanged because the scripts use only symbolic variable information. Of course, the user has to ensure that the variable references used within the script haven't changed.

SPECIFYING THE TIMING BEHAVIOUR IN TESTS

For real-time scripts, the capability to specify the exact timing behaviour is crucial. In the solution presented in this paper, the Python standard statement `yield` (officially available since Python 2.3) is used as a key element to time control real-time scripts.

Python Generator Functions

By Python programming convention, every function that contains at least one `yield` statement becomes a so-called generator function automatically. Generator functions can be executed stepwise: each time the interpreter reaches a `yield` statement within a generator function, it jumps back to the calling instance and returns a result object, which is at least an empty object when using `yield None`.

When the generator function is called again, the execution resumes exactly where it was suspended before with all local data maintained.

Use of Generator Functions for Real-Time Testing

In the approach presented here, all test scripts are embedded in an overall generator function, the `MainGenerator` function. It is called as top-level entry point first when executing a real-time test and must contain at least one `yield None` statement. So, an underlying real-time test scheduler can detect this top-level generator function and register it for execution.

Generator functions can be nested within a script so that generator functions can call other sub-generator functions. Doing so, the chain of commands has to be kept intact, meaning that every generator function has to be called by a leading `yield` statement. By this it is guaranteed that the top level real-time scheduler recognizes all nested generator functions for correct execution.

When the `yield None` statement is executed in a real-time script, the flow of control is given back to the real-time scheduler. The real-time scheduler resumes the interrupted generator function in the next simulation step. By this, it is easy to implement the timing behaviour of a test script because `yield None` can be used to split up execution of Python statements between different sample steps, as shown in the following:

```
from rttlib import variable

Const1 = variable.Variable(r'Model Root/Const1/Value')


def MainGenerator():

    # sample step n
    Const1.Value = 1.0

    yield None

    #sample step n+1
    Const1.Value = 2.0
```

When the script is executed, the variable `Const1` is written two times. By separating the assignments using the `yield None` statement, the first write command is executed exactly one sample step before the second write command.

If the `yield` statement is missing, `Const1` would have been written two times in the same sample step, with the last write access overwriting the first one immediately.

Example: Detecting model events

A common test case is the detection of an event being generated by the real-time model behaviour. An event is specified by a condition referencing certain real-time model variables. As soon as the event has been detected, some test action shall be triggered in the same sample step.

A code example is given below:

```
velocity    = variable.Variable(r'Model Root/Velocity/Out1')
brake_pedal = variable.Variable(r'Model Root/brake/Value')

while (velocity.Value <= 80.0):    # 80 kph
    yield None

brake_pedal.Value = 100.0          # 100 percent = full brake
```

In this example, the real-time script observes the speed of a simulated vehicle and initiates a full brake maneuver immediately when the vehicle exceeds 80 kph.

Example: Implementing a real-time wait function

Using the `yield` statement in a real-time script, it is very easy to implement a real-time wait function:

```
from rttlib import variable

Time = variable.Variable(r'currentTime')


def wait (seconds):
    start_time = Time.Value

    while( (start_time + seconds) > Time.Value):
        yield None
```

This wait function makes use of the model variable `currentTime` that holds the current simulation time in seconds and is automatically updated by the real-time model. On entry, the `wait` function saves the value of `currentTime` as start time and remains in the `while` statement until the requested waiting time (parameter `seconds`) has been elapsed.

By the use of the `yield` statement in the body of the `while` statement, the `wait` function is non-blocking because flow of control is given back to the real-time test scheduler in each sample step.

IMPORTING LIBRARY MODULES

For accessing real-time model variables as shown before, the class `variable` of the `rttlib` module can be used. In general, external Python modules can be used through the Python `import` statement. After importing, all (or selected) classes and functions of the respective module, they are available in the context of the real-time script. By this, standard Python modules, specific real-time scripting modules (like the `rttlib`) or user-defined modules can be integrated in the real-time script.

Using the import mechanism, user has access to various standard library modules coming with Python out-of-the-box. E.g., if a model variable has to be stimulated in real-time with a sine function, the Python `math` module can be used:

```
from math    import sin, pi
from rttlib  import variable

Time = variable.Variable(r'currentTime')


def generate_sin(var,amp,freq):
  start_time = Time.Value
  omega = 2.0 * pi * freq


  while(1):
    var.Value = amp * sin(omega*(Time.Value-start_time))
    yield None
```

After being called, the `generate_sin` function generates an endless loop. When it is invoked within a script to stimulate the real-time model variable `Const1` (with amp=2.5 and freq=1.0) at model time t=22s, this results in the the following plot:

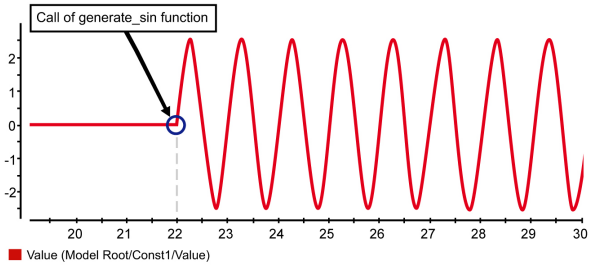

Figure 5: Stimulation of a model variable by the generate_sin function

IMPLEMENTING CONCURRENCIES

Various test scenarios need the ability to describe concurrent execution within a real-time script. Different test functions shall be combined easily and executed concurrently within a single sample step.

For concurrencies within a real-time script, two different concepts are available, `Parallel` and `ParallelRace`.

Parallel

User can execute several generator functions concurrently by referencing them within a `Parallel` generator function. Each generator function is executed once in a single simulation step.

```
from rttlib   import scheduler, variable
from userlib  import generate_sin

Const1 = variable.Variable(r'Model Root/Const1/Value')
Const2 = variable.Variable(r'Model Root/Const2/Value')

def MainGenerator():
  yield scheduler.Parallel(generate_sin(Const1, 2.5, 1.0),\
                           generate_sin(Const2, 1.0, 1.0))
```

If this script is started (again at model time t=22s), the Parallel statement invokes the `generate_sin` function simultaneously for the model variables `Const1` and `Const2` in each simulation step. This results in the following plot:

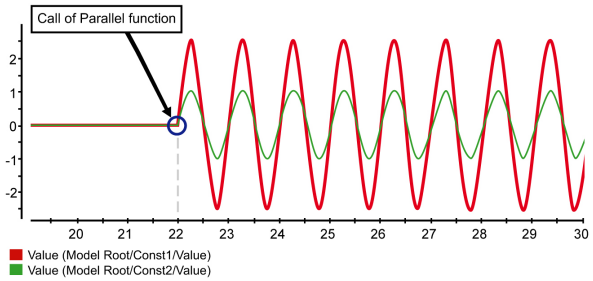

Figure 6: Stimulation of two model variables by use of Parallel

ParallelRace

In general, the `Parallel` generator function terminates, as soon as all generator functions included have terminated. So, the `Parallel` function of the example above will never terminate, because it includes two endless signal generators.

The `ParallelRace` generator function implements another termination condition. It finishes when at least one of the generator functions included terminates, and then also terminates all other generators that are still running.

With this behaviour, the `ParallelRace` function offers an elegant method to limit the stimulus generation time by combining the endless signal generators with the `wait` function introduced before:

```
from rttlib   import scheduler, variable
from userlib  import generate_sin, wait

Const1 = variable.Variable(r'Model Root/Const1/Value')
Const2 = variable.Variable(r'Model Root/Const2/Value')

def MainGenerator():
  yield scheduler.ParallelRace \
                (generate_sin(Const1, 2.5, 1.0),\
                 generate_sin(Const2, 1.0, 1.0),\
                 wait(5.0))
```

The `wait` function is executed for five seconds and finishes after that time. By this, the `ParallelRace` function is also finished and terminates the two endless signal generators:

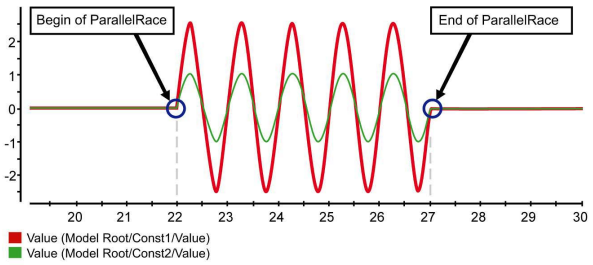

Figure 7: Timed stimulation by use of ParallelRace

Example: Observe a signal and measure a duration

Real-time scripts can be used for real-time measurements with sample time precision. Given a `check_signal_in_corridor` generator function checking whether a variable lies within a specific corridor, and another generator function `increase_timer` measuring time:

```
from rttlib   import variable

modelStepSize = variable.Variable(r'modelStepSize').Value

def check_signal_in_corridor(sig, min, max):
    while(sig.Value >= min and sig.Value <= max):
        yield None

def increase_timer(duration):
    duration[0] = 0.0

    while(1):
        yield None
        duration[0] = duration[0] + modelStepSize
```

Then, by using a `ParallelRace` it is very easy to combine those two functions in order to implement a new generator function such as an observer:

```
from userlib import check_signal_in_corridor,increase_timer
from rttlib  import variable, scheduler


velocity  = variable.Variable(r'Model Root/Velocity/Out1')
time      = [0]


def MainGenerator():
  yield scheduler.ParallelRace \
       (check_signal_in_corridor(velocity, 60, 80),\
        increase_timer(time))

  print "Velocity was ",time[0], " s between 60 and 80 kph"
```

For the model variable `velocity` it is checked how long it remains between 60 and 80kph. As long as this is the case, the `increase_timer` function executed in parallel measures the elapsed time. Afterwards, the result is available in the `time` variable (here implemented as Python list, because it has to be modified by reference using the `increase_timer` function) and printed on the host PC.

Example: Wait for an event using a timeout

The signal observation demonstrated above can be extended to cover another testing use case. Now, the time has to be measured until a model variable reaches a specified signal corridor. To implement this, only the `check_signal_in_corridor` function has to be extended slightly:

```
def check_signal_reaches_corridor(sig, min, max):
    while(not (sig.Value > min and sig.Value < max)
        yield None
```

This new function returns, when the signal value is greater than the `min` value and less than the `max` value. To avoid an endless measurement, a timeout can be implemented by the use of our valuable `wait` function. For this, only the `ParallelRace` statement has to be extended:

```
from userlib import check_signal_reaches_corridor, \
                    increase_timer, wait
from rttlib  import scheduler, variable

velocity  = variable.Variable(r'Model Root/Velocity/Out1')
time      = [0]
timeout   = 10.0

def MainGenerator():
  yield scheduler.ParallelRace \
       (check_signal_reaches_corridor(velocity, 60, 80),\
        increase_timer(time),
        wait(timeout))

  print "Evaluation finished after ",time[0], " s."
```

`ParallelRace` finishes, when the condition has become true or the timeout elapsed. So, the condition was fullfilled after `time[0]` seconds, when the measured time and given timeout are different.

Example: Data exchange between real-time scripts

Every real-time script uses a separate Python namespace. By this, unintentional naming conflicts between independent test scripts are avoided.

If a test script wants to publish data globally, this can be done by the `globalvariables` module:

```
from rttlib import globalvariables
globalvariables.mySharedData = 42.0

def MainGenerator():
    yield None
```

After this script has been executed, the `mySharedData` variable is accessible from within other real-time scripts:

```
from rttlib import globalvariables, variable

Const1 = variable.Variable(r'Model Root/Const1/Value')

def MainGenerator()
    Const1.Value = globalvariables.mySharedData
    yield None
```

The second script uses the published data to initialize the `Const1` variable.

COMMUNICATION BETWEEN HOST PC AND REAL-TIME PROCESSOR

Scripts or parts of them can be executed either on the real-time interpreter or on the host PC. Function calls are possible in both directions. This can be used for communication and synchronization purposes between the host PC and the real-time interpreter.

Function calls can also be used to exchange data between the host PC and the real-time interpreter. Since Python is used on the real-time processor as well as on the host PC, even complex Python objects (e.g. data structures, object instances) can be exchanged directly without data transformation.

Remote function calls from Host PC to real-time processor

After a test script has been downloaded, it can be parameterized before execution is started. This can be done by using a function call from the Host PC to the real-time processor. Doing so, parameters can be changed without modifying the real-time script itself. As an example, this can be used to perform parameter studies, where the same real-time script is executed repeatedly with varying parameters.

Remote function calls from real-time processor to PC

Host PC functions can also be called from the real-time interpreter. As an example, it is possible to upload test results to the host PC. By this, real-time scripts can gather the test results autonomously and transfer them to the host PC when script execution has been finished. Afterwards, the host PC can be used to post-process the data and to generate dedicated report documents (e.g. PDF, HTML).

Another example is the use of test interfaces that are only available on a host PC. A standard diagnostic tool can be called by a real-time test via ASAM MCD-3 interface to access an ECU fault memory entry. The result value can be given back to the real-time script.

Traceback handling

Due to the use of a Python interpreter mechanism, some errors can occur during script execution (like a division by zero). These run-time errors can be handled by Python's `try..except` construct also in a real-time script.

Error information is also shown in a Python traceback on the host PC. Users can then very easily identify problems, in the same way as when using a PC based interpreter.

ADVANCED APPLICATION EXAMPLES

Running multiple scripts concurrently

It is possible to download and start multiple scripts totally independent from each other. This includes downloading a script while another script is running. Starting and stopping of each script can be done separately from the others.

This can be used to implement „utility scripts“ to be run concurrently to the actual test, e.g. for data logging, observing certain system states (e.g. system time, battery voltage, or CAN bus signals), and invoking alarms or emergency-stops in cases where a system timeout has been reached, battery voltage exceeds a certain range or an ECU is in a critical state.

Since actual test script execution is independent from a utility script, such functionality does not have to be implemented directly in the test script. This makes test scripts modular and more reusable, and test script development becomes much easier and more flexible.

Real-Time Test Manager, a specific graphical user interface, can be used to manage concurrent real-time scripts tests running on the processor board, to download them and to start execution. Additionally, this can be done using Python scripts on the PC. By this, the users' real-time test framework can be set up and controlled fully automatically.

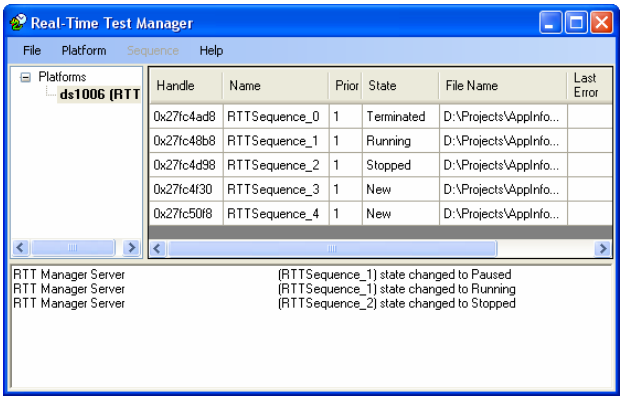

Figure 8: Real-Time Test Manager for handling real-time scripts

Dynamic modeling

The examples shown before were motivated mainly by performing specific testing tasks such as precise time measurement, signal stimulation and signal evaluation. More generally, real-time scripts can be seen as an extension of the real-time model. In cases where minor model extensions or modifications are required, real-time scripts can be used. This also enables model modifications during simulation run-time without the need to stop, re-compile, the simulation model. This can also be useful to dynamically create model variants.

As an example, the following script is used to calculate an electrical power P(t), as the product of voltage V(t) and electrical current I(t), and write the result value to a model parameter:

```
from rttlib import variable

Voltage = variable.Variable(r'Model Root/Voltage/Out1')
Current = variable.Variable(r'Model Root/Current/Out1')
Power   = variable.Variable(r'Model Root/Power/Value')

def MainGenerator():
    while(1):
        Power.Value = Voltage.Value * Current.Value
        yield None
```

The advantage of dynamic modeling via real-time scripts is, that temporarily needed model parts can be implemented "on demand". If the model part is no longer needed, the script can be stopped and the calculation time can be used for different tasks.

Dynamic rest bus simulation

ECUs are typically interconnected using bus systems like CAN, LIN, and FlexRay to communicate with each other and to perform distributed function. In HIL simulation and testing, the bus interaction of non-existing ECUs has to be emulated via rest bus simulation. Rest bus simulation includes symbolic coding and decoding according to the specific bus specifications and thus requires some calculation efforts on the real-time processor.

Real-time scripts can implement highly reactive and precise behaviour for restbus simulation. Use cases like 'Send Msg1 exactly 30 milliseconds after reception of Msg2' are easy to accomplish by means of real-time Python programming.

Therefore it is very useful to implement the static parts of the restbus simulation in the simulation model and to extend it in specific test cases using real-time scripts. By this, the overhead of messages that is only necessary for single test cases, but not for the whole life cycle of the HIL simulator can be minimized and execution time on the real-time processor can be saved.

EXECUTION TIME BENCHMARKS

In Table 1, benchmarks for some of the above mentioned examples are given. They show the turn-around time of the real-time processor which is used to run the respective script. The benchmarks have been done using a dSPACE DS1006 processor board with an AMD Athlon processor running at 2,6 GHz.

Script example	Execution time
Wait function	0.41 µsec
Single sine generator	1.30 µsec
Two sine generators in parallel	2.35 µsec
Two sine generators and a timeout function in parallel	2.75 µsec
Observe a signal and measure a duration in parallel	0.85 µsec
Wait for an event using a timeout	1.05 µsec

Table 1: Typical execution time for script examples

The execution time numbers given in Table 1 do not include the offset time required by the interpreter itself. The offset execution time of the Python interpreter is approx. 0.25 µsec.

In an HIL system, the real-time processor has to run the real-time model in parallel to the real-time script. Therefore, the model and the script are not totally independent from each other. The more execution time the model needs, the fewer time is available for the script, and vice versa. Therefore, it is of special interest, how much execution time is required for typical test scripts, and how much time is available for the real-time model. Experience with more complex examples than the ones above shows: 10-15% of the turn-around time of the real-time processor should be reserved for real-time scripts. Assuming a safety margin of 5-10% permits between 75% and 85% of the execution time to be used for the real-time model. This seems to be a reasonable portion for most of the HIL applications.

CONCLUSION

To summarize, this paper presents a new approach in ECU testing by combining the flexibility and power of a standard scripting language with the high timing precision of a real-time simulation system. By running and using a standard Python interpreter on the real-time processor, it is possible to implement test scripts which are executed in real-time as well as synchronized with the real-time model of the HIL simulator. Timing behavior of such test scripts is much more precise than when using conventional test script execution on a standard PC.

Compared with conventional approaches for real-time related testing, using a standard scripting language on a real-time simulator provides a number of advantages for the user:

- Real-time scripts can be downloaded and executed independently from the real-time model. I.e. the real-time model application does not have to be recompiled to run new tests.

- Since the same programming language is used on the real-time processor as on the host PC, users do not have to learn a new language or proprietary configuration instructions used to configure real-time services, but can directly reuse their programming know-how to use the new technology.

- Real-time scripts in a standard script language are easier to read and to understand than proprietary instruction languages used to configure a set of real-time services as in conventional approaches.

- Test scripts or parts of them, such as algorithms for signal evaluation, can be reused once developed for the host PC. However, attention has to be paid to the time-driven script execution, since the scheduling on a real-time processor can be different from that on the host PC.

- Users can extend the functionality very easily by implementing new libraries and using them on the real-time system instantly.

As a result, the solution presented in this paper provides a significant improvement in HIL technology and an increase in the quality of ECU testing in general.

REFERENCES

1. Lamberg, K.; Wältermann, P.: Using HIL-Simulation to test Mechatronic Components in Automotive Engineering. 22. Tagung Mechatronik im Automobil, Munich, Germany, 2000
2. Gühmann, C.; Riese, J.: Testautomatisierung in der Hardware-in-the-Loop Simulation. VDI-Berichte Nr. 1672, Germany 2002
3. Lamberg, K., Richert, J.; Rasche, R.: A New Environment for Integrated Development and Management of ECU Tests. SAE 2003-01-1024, 2003.
4. dSPACE product information, http://www.dspace.de.

CONTACT

Holger Krisp is Product Manager for Test and Experiment Software at dSPACE GmbH, Paderborn, Germany.

E-mail: hkrisp@dspace.de
Web: http://www.dspaceinc.com