Hitex Germany

– Head Quarters –
Greschbachstr. 12
76229 Karlsruhe
Germany

☎ +049-721-9628-0
Fax +049-721-9628-149
E-mail: Sales@hitex.de
WEB: www.hitex.de

Hitex UK

Warwick University
Science Park
Coventry CV47EZ
United Kingdom

☎ +44-24-7669-2066
Fax +44-24-7669-2131
E-mail: Info@hitex.co.uk
WEB: www.hitex.co.uk

Hitex USA

2062 Business Center Drive
Suite 230
Irvine, CA 92612
U.S.A.

☎ 800-45-HITEX (US only)
☎ +1-949-863-0320
Fax +1-949-863-0331
E-mail: Info@hitex.com
WEB: www.hitex.com

Tessy Tutorial

Component / Integration Testing With Tessy

Tessy is a tool to automate unit / module / integration testing of embedded software.

This tutorial describes the component testing feature of Tessy. **This feature was introduced in Tessy V2.9.**

The paper at hand is illustrated by many screen shots. This should allow studying this paper independently of a Tessy installation.

Some familiarity with Tessy is prerequisite.

Architecture:	TESSY
Author:	Frank Buechner / Hitex GmbH
Revision:	06/2009 – 001

© Copyright 2009 - Hitex Development Tools GmbH

All rights reserved. No part of this document may be copied or reproduced in any form or by any means without prior written consent of Hitex Development Tools. Hitex Development Tools retains the right to make changes to these specifications at any time, without notice. Hitex Development Tools makes no commitment to update nor to keep current the information contained in this document. Hitex Development Tools makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hitex Development Tools assumes no responsibility for any errors that may appear in this document. DProbe, Hitex, HiTOP, Tanto, and Tantino are trademarks of Hitex Development Tools. All trademarks of other companies used in this document refer exclusively to the products of these companies.

ApplicationNote.dot - 11/2007 - 005

Contents

1 Terms: From Unit to Component 3
1.1 Types of Test Objects 3
1.1.1 Single Function 3
1.1.2 Hierarchy of Functions 3
1.1.3 Interacting Functions, Non-temporal Approach 3
1.1.4 Interacting Functions, Temporal Approach 5

2 Example: Interior Light 6
2.1 Specification of Interior Light 6
2.2 Implementation of Interior Light 6
2.3 Prerequisites 8
2.3.1 Prerequisites in the File System 8
2.3.2 Prerequisites in Tessy 8
2.4 Testing Interior-Light with Tessy 9
2.4.1 Create Project and Module 9
2.4.2 Select Component Testing 10
2.4.3 Select the Test Object 11
2.4.4 Open the Module 11
2.4.5 Adjust the Interface 12
2.4.6 Define Test Cases (Scenarios) 17
2.4.7 Execute the Scenarios 32
2.4.8 View the Results 34

3 Variations of the Example 39
3.1 Check initialization 39
3.1.1 Data from the TDE 39
3.1.2 Data from init() 39
3.2 Check Final State 39
3.3 Check Intermediate State 39
3.4 Check That Event Is Not Happening 39
3.5 Unite LightOn() and LightOff() 39
3.6 Use an Second Component 40
3.7 Change the Time Base 40
3.8 Several Handler Functions 40
3.9 Specify Scenarios in the CTE 40

4 Versions Used 41

5 The Author 41

© Copyright 2009 Hitex Development Tools GmbH

1 Terms: From Unit to Component

1.1 Types of Test Objects

We discuss the terms unit testing, module testing, integration testing, and component testing based on the type of the test object. Then we will introduce temporal component testing. We assume software written in C.

1.1.1 Single Function

A single function is the smallest reasonable test object of a C program and is usually considered to be a unit. Unit testing is functional black-box testing, i.e. based on the interface of the function (i.e. input and output variables). Unit testing is dynamic, i.e. the test object is executed.

If the function under test calls other functions, during pure unit testing, the calls to these functions are replaced by calls to stub functions.

Testing of a single function (i.e. unit testing) is the hitherto functionality of Tessy and not discussed here further.

1.1.2 Hierarchy of Functions

A hierarchy of functions can be tested in a manner very similar to the test of a single function by taking the top-level function as the test object, and considering the called function as to be inside the test object.

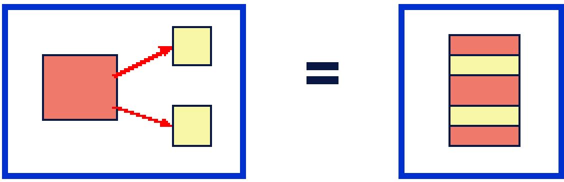


Fig. 1 A hierarchy of functions takes the top-level function as test object

This is technically achieved by not replacing the called functions by stub functions. You may still consider this as unit testing as above, but for bigger units. It can also be considered as integration testing for the functions in the hierarchy, because they have to work together correctly to pass the tests, at least to some extent. Such a hierarchy of functions could also be called a module, but I am reluctant to use this term, because of the connotation to the C source modules of a C program. (A C source module is not automatically an appropriate test object for module testing, because it is defined syntactically, whereas a module for module testing is defined semantically.)

The testing of a hierarchy of functions is technically very similar to the testing of a single function. Testing of a hierarchy of functions is the hitherto functionality of Tessy and not discussed here further.

1.1.3 Interacting Functions, Non-temporal Approach

In contrast to a hierarchy of functions, in the section at hand, we consider functions that do not necessarily have a calling relation. However, we assume that the functions work together though, e.g. operate on common data, with the objective to achieve a common goal. The well-known stack data

© Copyright 2009 Hitex Development Tools GmbH

type with its push() and pop() operations is a very simple example for that. The term module may be appropriate for such a collection of cooperating functions, but I would prefer the term component, still because of the inappropriate connotation to C source modules.

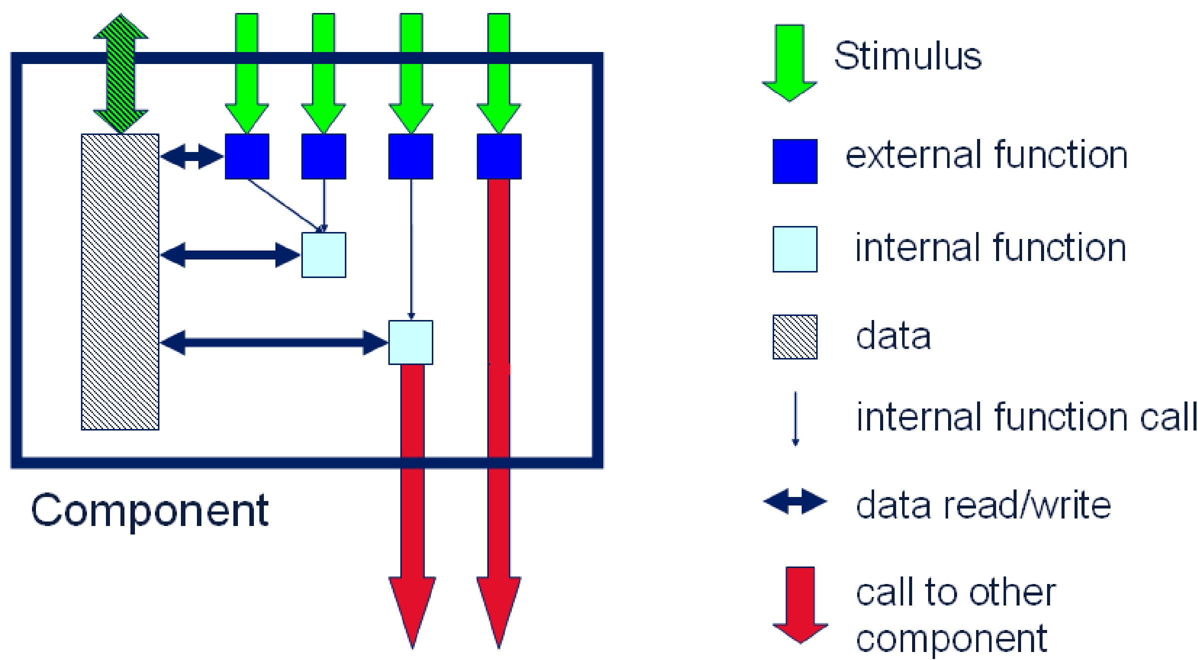


Fig. 2 The structure of a component and its interface to the outside world

At least one of the functions combined in a component must be callable from the outside of the component to stimulate the functionality of the component. Normally several function are callable from external. We call these functions “component functions”. A test for a component is not longer a single function call (as in the two sections above) but a sequence of calls to the component functions. The calls to the component functions stimulate the component. Like testing of a single function, a test case for a component also comprises of input and output data (variables of the component and parameters of the called external functions). A component may have internal functions that cannot be called from the outside of the component, but only from functions inside the component. If internal functions exist and if so, what they do exactly, is not relevant for component testing, because component testing takes the component as a black box. However, relevant for the result of a component test is the sequence of the calls from within the component to (callable) functions in other components. This is with respect to the number of the calls, their order, and the parameters passed by the function calls to other components.

Obviously, the functionality of the functions in a component and the interfaces between them are tested by component testing, at least to a certain extent. Hence, component testing can well be considered as integration testing for the functions in the component.

Component testing = integration testing for the functions in the component

© Copyright 2009 Hitex Development Tools GmbH

1.1.4 Interacting Functions, Temporal Approach

In the above section, time was not an issue, neither for the timely distance of the stimulating calls to a component, nor for the timely distance between a stimulating call to a component and the resulting call to another component. However, it could be an important test to check if such a resulting call happens "fast enough" after a stimulating call, i.e. within a given time frame.

To be able to test the temporal behaviour of a component in a simulated environment, a simulated time base needs to be available. This is usually the case if the component is executed under the control of a Real-time Operating System (RTOS) with time-slicing, e.g. OSEK. Then a certain function inside the component is called in known equidistant times (e.g. every 10 ms) by the operating system. The calls to this function represent the "heartbeat" of the component. They provide a time reference for the testing of the temporal behaviour of the component. (The heartbeat function is usually called "handler function" or "work task" or simply "tick").

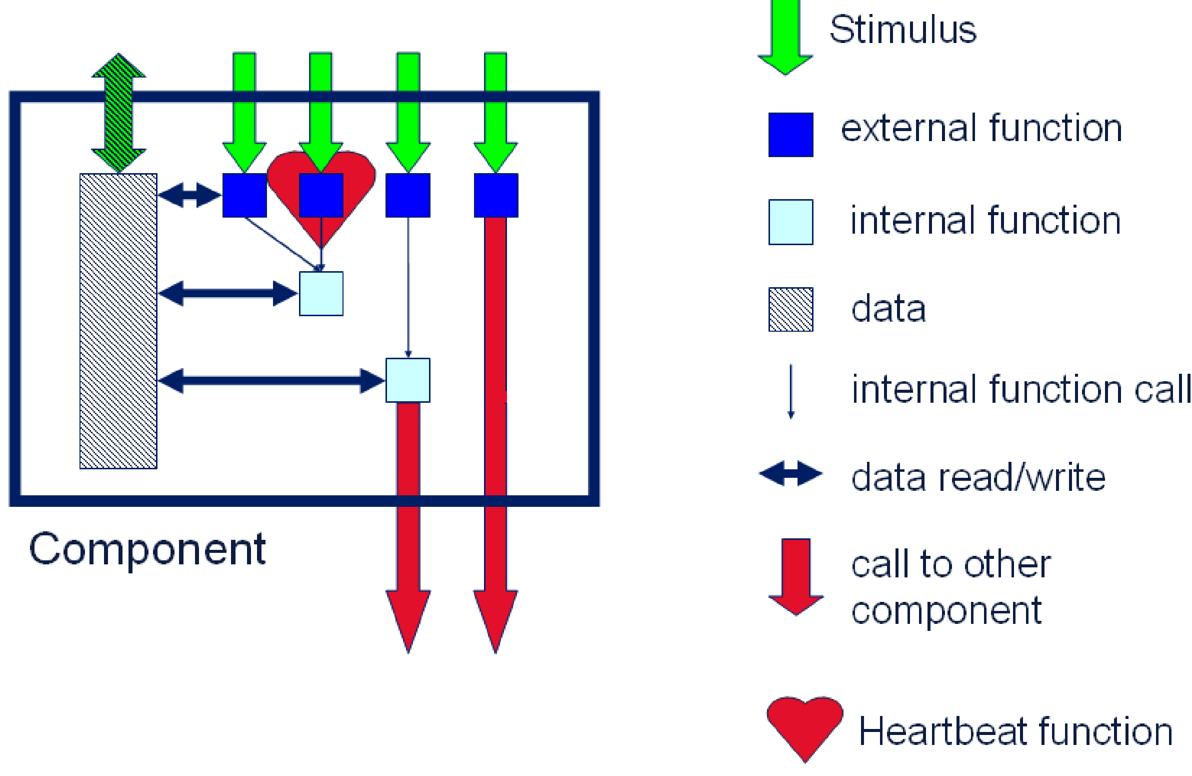


Fig. 3 If a heartbeat function exists, timely behavior can be tested

© Copyright 2009 Hitex Development Tools GmbH

2 Example: Interior Light

To introduce temporal component testing with Tessy, we use a simple example: “Interior Light”.

2.1 Specification of Interior Light

The interior light of a car shall be controlled by two inputs: The door and the ignition of the car.

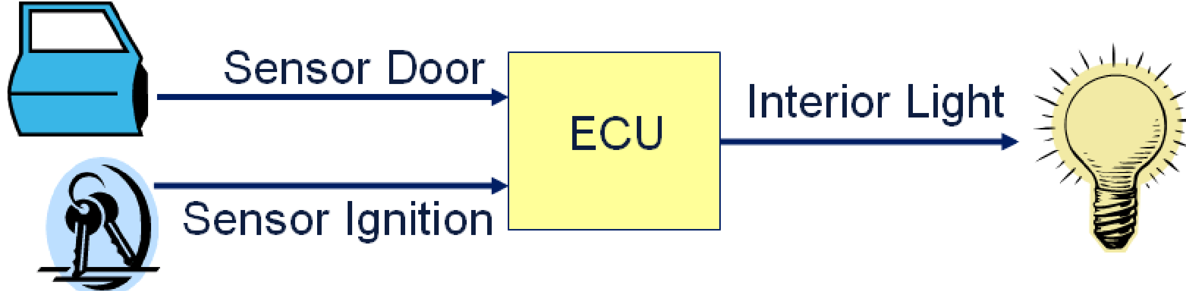


Fig. 4 Scheme of Interior Light

The functional specification comprises of three simple requirements:

1. If the door is closed, the interior light shall go on
2. The interior light shall go off after 5 seconds at the latest
3. If the ignition is switched on, the interior light shall go off immediately

The specification above is obviously not complete. Especially the initial state of door, ignition, and light is not given. Also it is not specified what shall happen e.g. if the ignition is switched off (after it was switched on), etc. But this simple specification is sufficient to demonstrate temporal component testing with Tessy.

For simplicity, we assume that the initial state shall be

- Door = open
- Ignition = off
- Light = off

2.2 Implementation of Interior Light

Below is a possible implementation in C for Interior Light.

```
/**************************************************************************
 Br, 2009-02-20
 Inspired by an example of Roman Pitschinetz, from the former
 Software Research Center of DaimlerChrysler in Berlin.
(c) Hitex Development Tools 2009
*************************************************************************/

typedef enum
{
        closed, open
} co_states;
```

© Copyright 2009 Hitex Development Tools GmbH

```
typedef enum
{
        off, on
} oo_states;

co_states sensor_door, state_door;
oo_states sensor_ignition, state_light;

extern void LightOn(void);
extern void LightOff(void);

void init(void)
{
        sensor_door = open;
        sensor_ignition = off;
        state_door = open;
        state_light = off;
}

void set_sensor_ignition(oo_states i)
{
        sensor_ignition = i;
}

void set_sensor_door(co_states d)
{
        sensor_door = d;
}

static void iLightOn(void)
{
        if(on == state_light)
                return;
        else
        {
                state_light = on;
                LightOn();
        }
}

static void iLightOff(void)
{
        if(off == state_light)
                return;
        else
        {
                state_light = off;
                LightOff();
        }
}
// heartbeat function
/* ********************************************************************/
void tick()
{
```

```
        static int Timer = 0;

        if((state_door == open) && (sensor_door == closed))
        { // Door was closed ---> start timer!
                Timer = 500;            // 5 seconds = 500 cycles of 10 ms
                iLightOn();
        }
        else
                if(on == sensor_ignition)
                {
                        iLightOff();
                }

        if(Timer > 0)
                Timer--;

        if(0 == Timer)
                iLightOff();

        // Store current value for next tick
        state_door = sensor_door;
}
```

Fig. 5 The contents of interior_light.c

This implementation features a heartbeat function, the function tick(). The implementation assumes that tick() is called every 10 ms. Based on this assumption the value “500” for the timer is calculated.

2.3 Prerequisites

2.3.1 Prerequisites in the File System

We assume that the implementation of Interior-Light is present in the file

C:\Tessy-Projects\Hitex_examples\Source\Interior_Light\interior_light.c

2.3.2 Prerequisites in Tessy

We assume an existing Tessy database (*.pdb, any database will do). If you do not dispose of a existing database, see Tessy’s online help how to create one.

2.4 Testing Interior-Light with Tessy

2.4.1 Create Project and Module

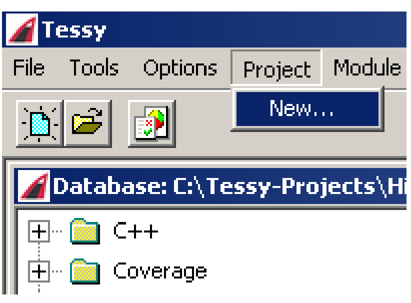


Fig. 6 Create a new project

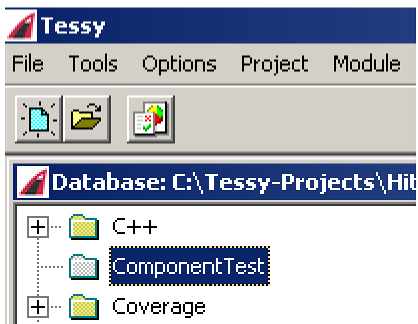


Fig. 7 Rename the project to “ComponentTest”

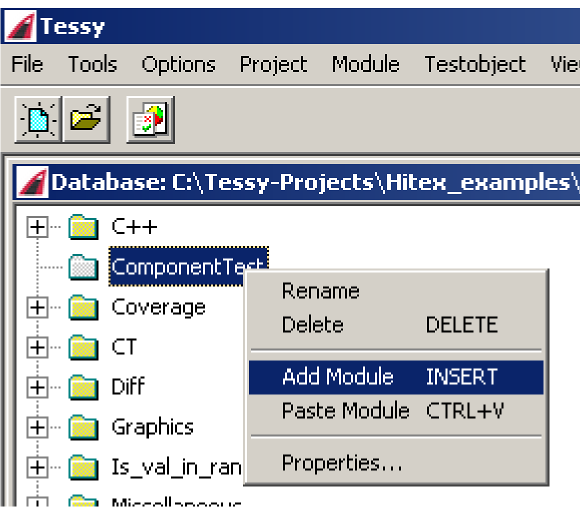


Fig. 8 Add a module to the project “ComponentTest”

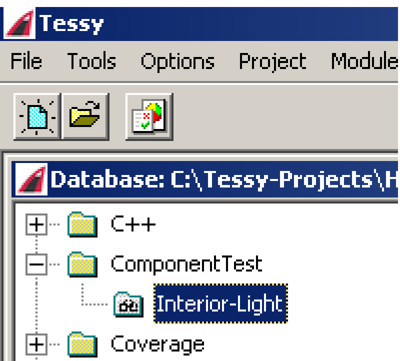


Fig. 9 Rename the module to “Interior-Light”

2.4.2 Select Component Testing

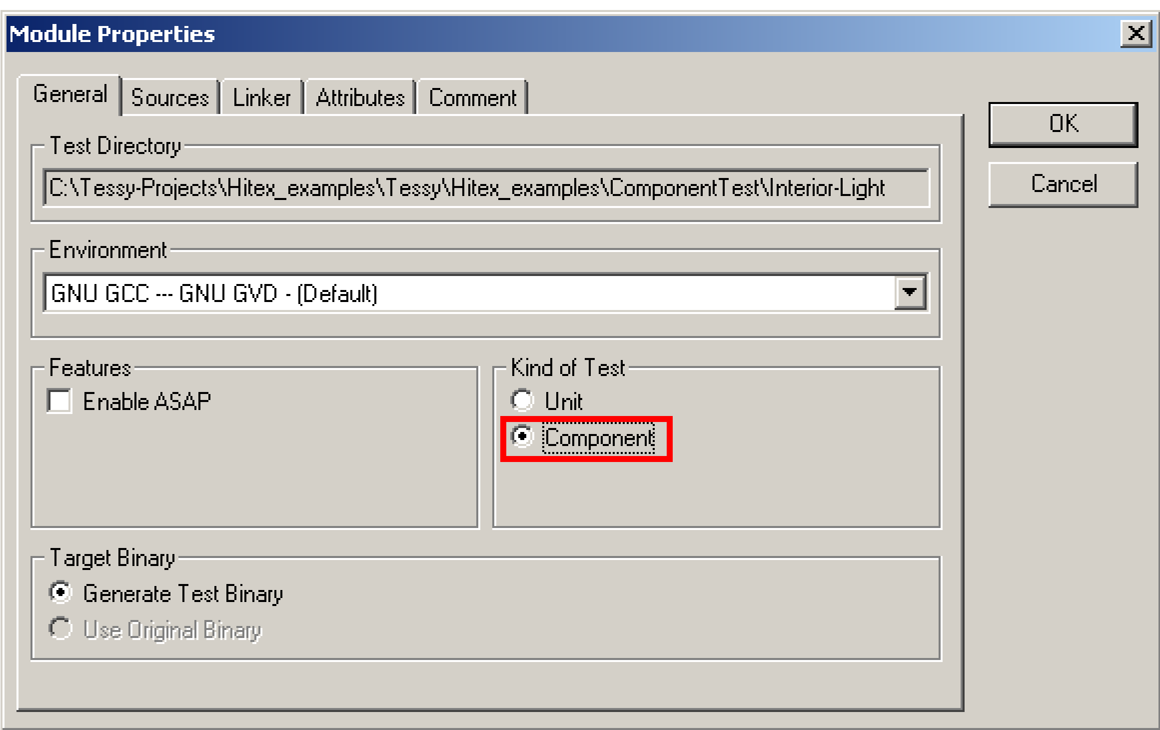


Fig. 10 Select “Component” in the Module Properties

By default, “Unit” is selected. This is the point where unit testing and component testing begin to part.

Please note:
As environment the default GNU gcc compiler is used.

© Copyright 2009 Hitex Development Tools GmbH

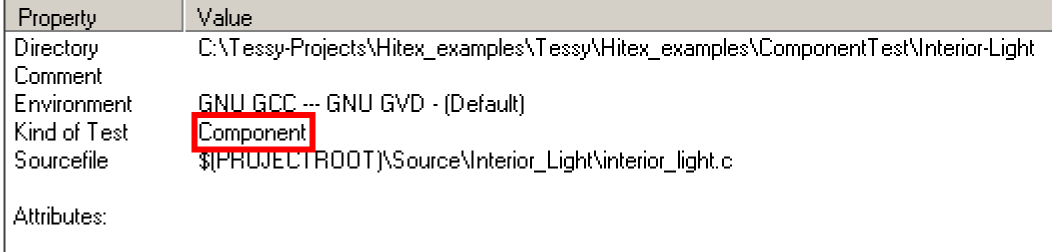


Fig. 11 Module Properties for component testing

2.4.3 Select the Test Object

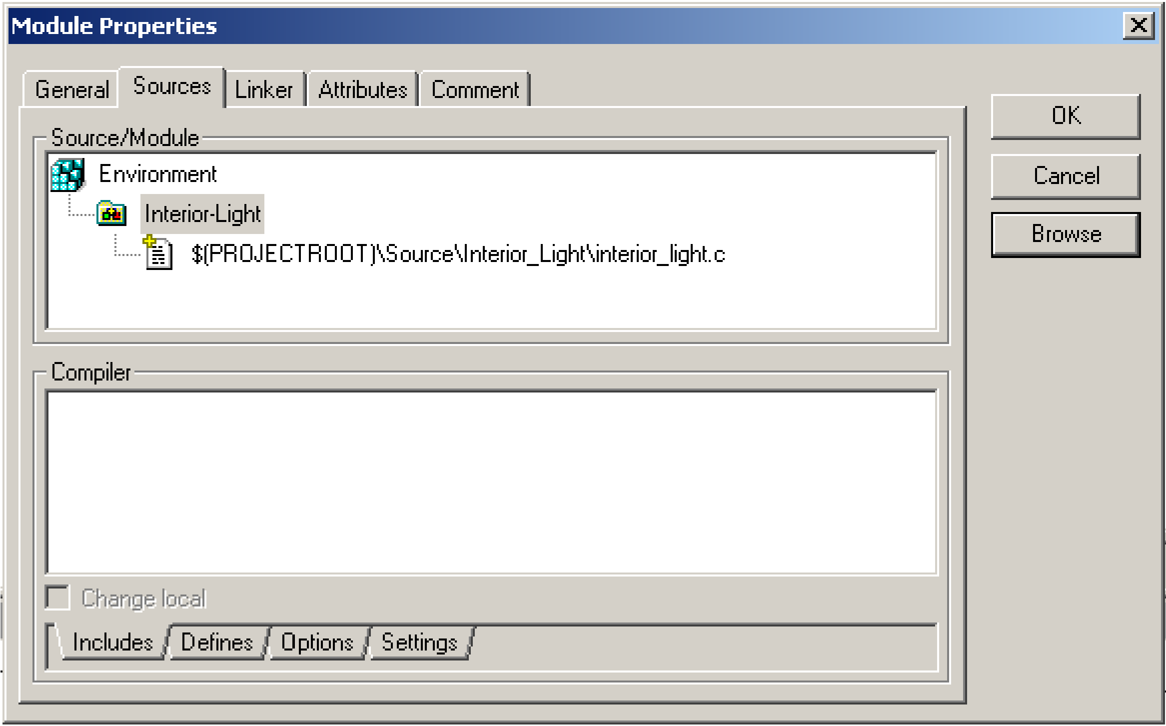


Fig. 12 Select “interior_light.c” in the source tab of the Module Properties

2.4.4 Open the Module

By clicking on the plus sign in front of the module name, the module opens.

© Copyright 2009 Hitex Development Tools GmbH

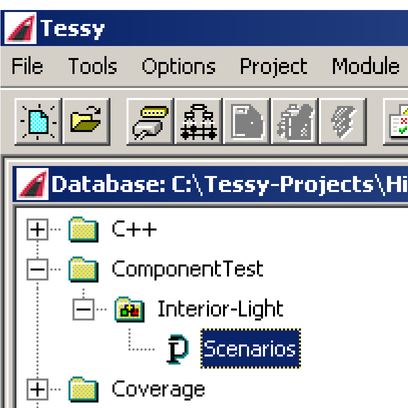


Fig. 13 The module "Interior-Light" is open

The only "test object" displayed has the default name "Scenarios". This is different to unit testing with Tessy, where the names of the possible test objects in interior_light.c (i.e. the functions) would be listed instead.

2.4.5 Adjust the Interface

We open the Test Interface Editor (TIE) for "Scenarios" as usual.

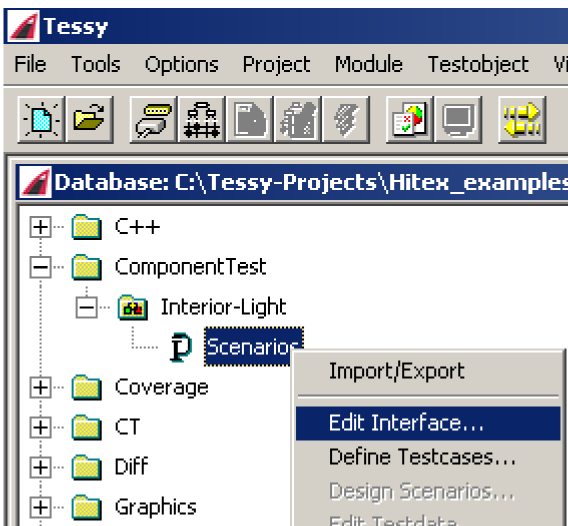


Fig. 14 Open the TIE

© Copyright 2009 Hitex Development Tools GmbH

TIE - 32 [Interior-Light:Scenarios]

File Search Options Help

Scenarios

- External Functions
  - LightOff
  - LightOn
- Component Functions
  - iLightOff
  - iLightOn
  - init
  - set_sensor_door
  - set_sensor_ignition
  - tick
- External Variables
- Global Variables
  - enum co_states sensor_door — Irrelevant
  - enum oo_states sensor_ignition — Irrelevant
  - enum co_states state_door — Irrelevant
  - enum oo_states state_light — Irrelevant
  - long tick::Timer — Irrelevant
- Parameter
- Return
- Enums
  - enum oo_states
    - off — 0
    - on — 1
  - enum co_states
    - closed — 0
    - open — 1
- Declarations (Unused)

Fig. 15 The initial interface

2.4.5.1 External Functions in the Interface

In the section “External Functions” of the interface, the two functions LightOff() and LightOn() are listed. These two functions are used (called) by the component Interior-Light; but these two functions are not implemented in interior_light.c. The component Interior-Light expects these two functions to be provided by another component of the application. However, we want to test the component Interior-Light without that other component. Therefore, we direct Tessy to provide replacements, i.e. (user) stub functions for these two functions.

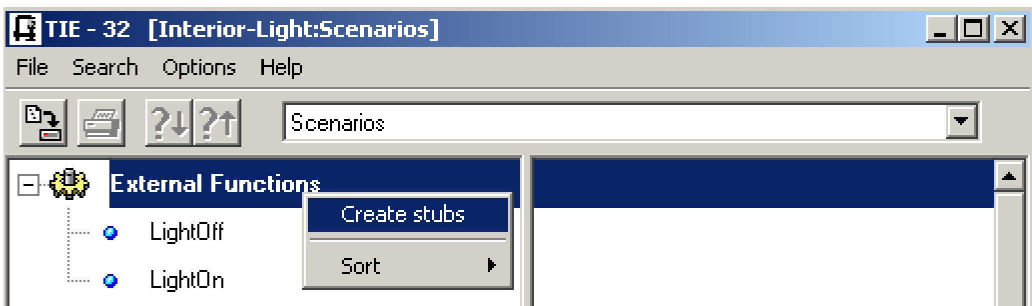


Fig. 16 Creating stubs for LightOff() and LightOn()

The fact that Tessy provides stub functions for external functions is indicated by a red tick mark in front of the function name.

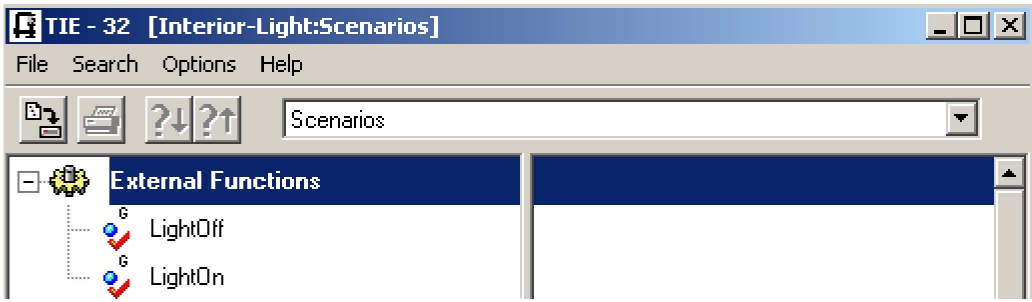


Fig. 17 Stubs are provided for LightOff() and LightOn()

2.4.5.2 Component Functions in the Interface

In the section “Component Functions” of the interface, all functions of the component Interior-Light were listed. Astonishingly, this includes also static functions like iLightOn(), which are not callable from outside the component.

Furthermore, to all functions a special symbol is attached (a crossed-out eye). This symbol indicates that the variables used by this function are not visible in the interface. This is the reason that all variables in the “Global Variables” section of the interface are “irrelevant”.

Background: Tessy assumes, that a typical component features much more variables that are “really” internal to the component, i.e. variables that you don’t want to see in the interface of the component, compared to variables, that you want to set or observe from the outside of the component during testing. Therefore, Tessy hides initially all variables.

© Copyright 2009 Hitex Development Tools GmbH

You can make all variables used by a component function visible in the interface by selecting the component function and using the context menu. In case a variable that you want to make visible in the interface is only used in a static function, you need this static function to be listed in the "Component Functions" section of the interface.

If you make any variable visible, the available passing direction will be the resulting passing direction of the usage of this variable within all component functions. You may adjust the passing direction to your needs.

2.4.5.3 Global Variables in the Interface

Because the function init() of the component Interior-Light uses all the variables we might be interested in, it is convenient for us to make the variables used by init() visible.

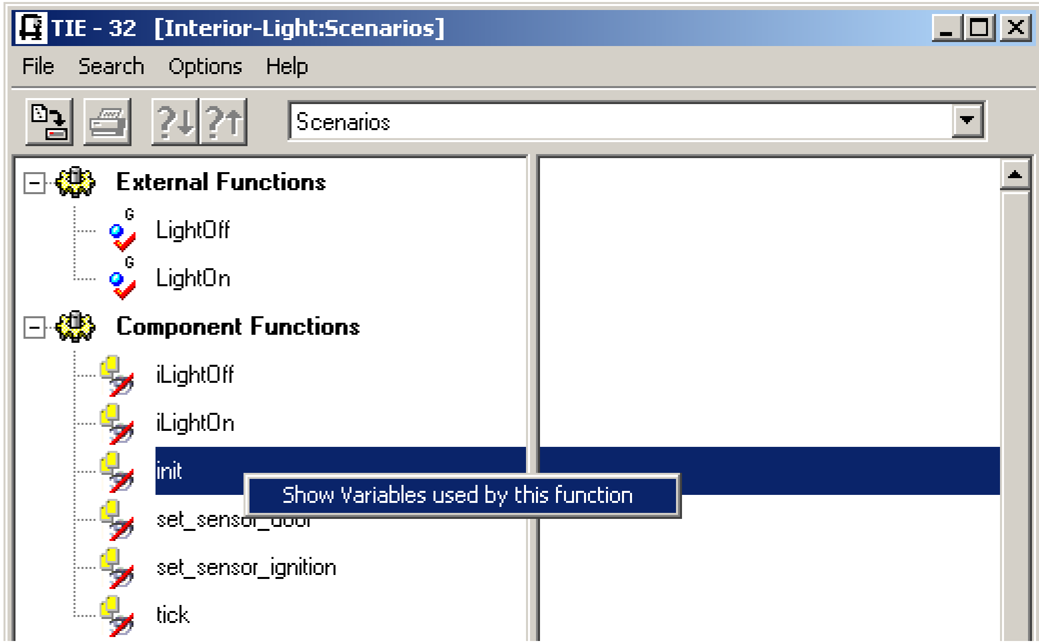


Fig. 18 Making the variables used by init() visible

It is possible to reverse the effect.

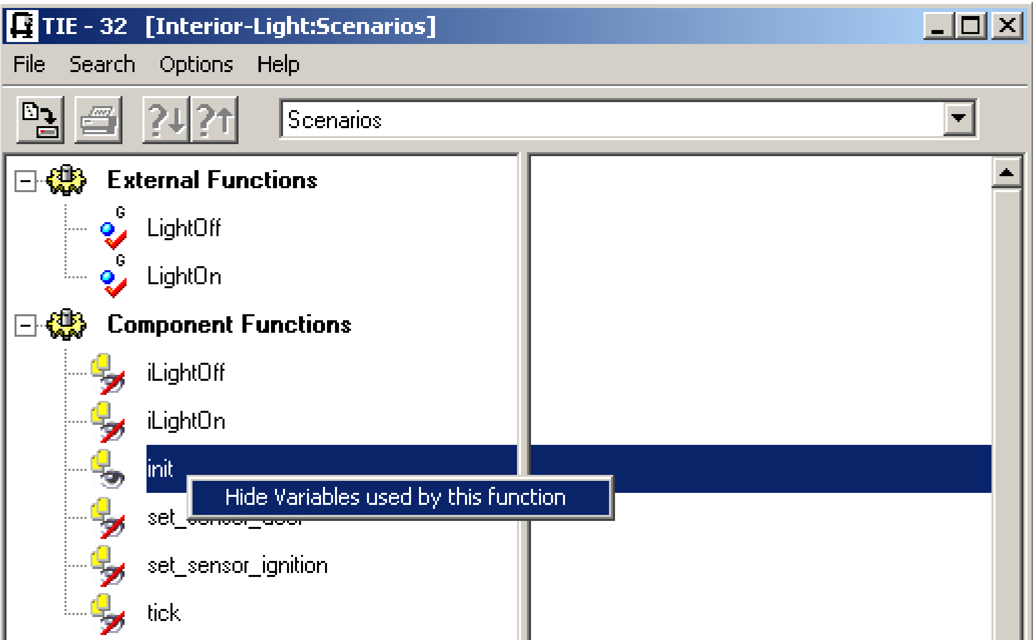


Fig. 19 Hide the variables used by init(). This reverses the effect of “Show Variables”!

With all the variables used by init() visible, the “Global Variables” section of the interface looks like below.

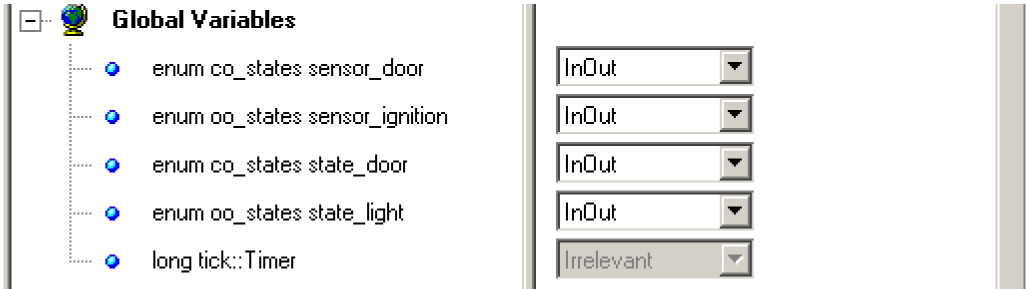


Fig. 20 The initial passing directions of variables used by init()

Tessy has set the initial passing directions of all variables used by init() to “InOut”, because all variables are both read and written in the component.

The variable timer is still “irrelevant”, because it is not used by init().

Because we are only interested in the variables sensor_door, sensor_ignition, and state_door as input, we set the passing direction for these variables to “In” manually.

© Copyright 2009 Hitex Development Tools GmbH

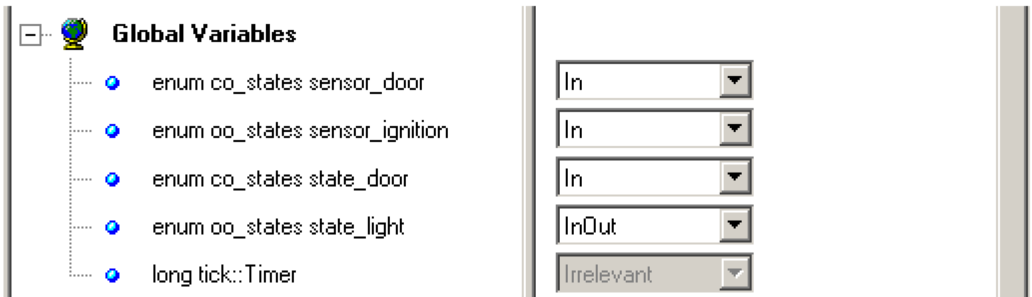


Fig. 21 The final passing directions of variables used by init()

We save and close the TIE.

2.4.6 Define Test Cases (Scenarios)

2.4.6.1 Add Scenarios

Now we want to define scenarios (i.e. test cases) for the component Interior-Light. For the moment, we content us with two scenarios:

1. Door closed: Light goes off after 5 seconds
2. Door closed + ignition on: Light goes off immediately

Please note:
It should be clear that these two scenarios are not sufficient for a serious test of the component.

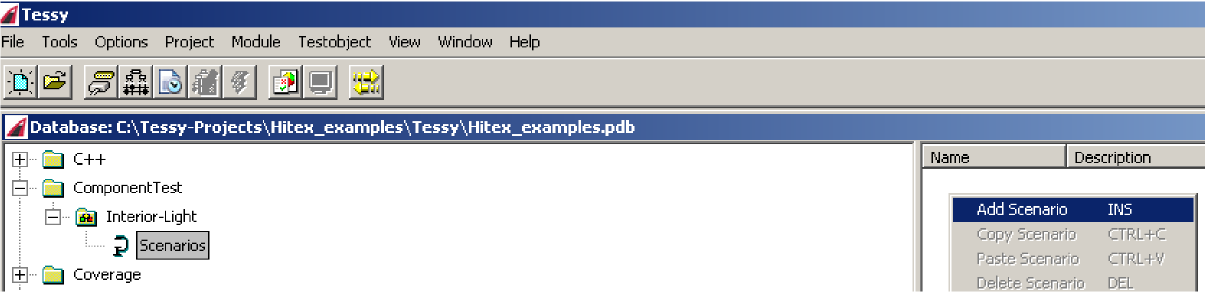


Fig. 22 Adding a scenario

We add two scenarios in the component tab on the right hand side of Tessy's main window.

Please note:
It is also possible (and recommended) to use the Classification Tree Editor CTE to specify scenarios and to import these scenarios to Tessy. But systematic test case specification is not the emphasis of this tutorial.

© Copyright 2009 Hitex Development Tools GmbH

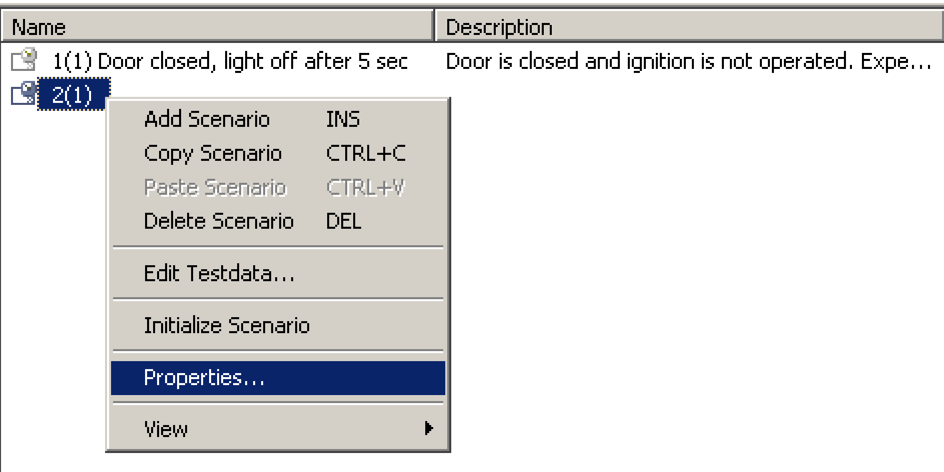


Fig. 23 Specifying name and description for the scenarios by using the context menu

2.4.6.2 Edit Test Data

By double clicking on the scenario icon, the Test Data Editor (TDE) opens.

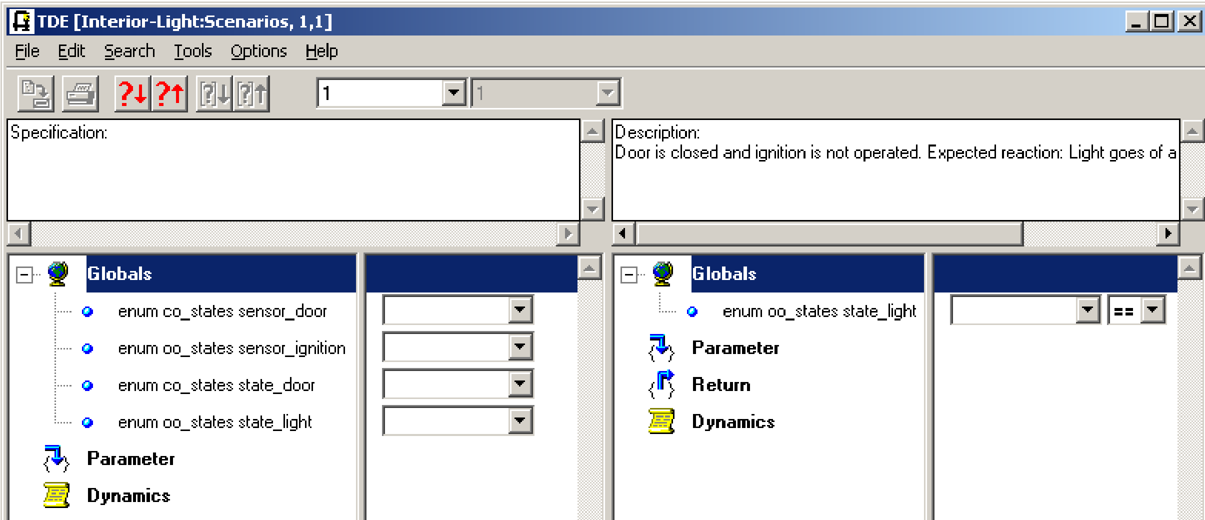


Fig. 24 Initial contents of the TDE for the first scenario

This is similar to unit testing. However, with component testing, the left hand side of the TDE allows specifying input data prior to the execution of a scenario; and the right hand side allows specifying expected result after the execution of the scenario.

Therefore, on the left hand side, we specify the test input data that represents the initial state as described above, i.e.

- Door = open
- Ignition = off
- Light = off

This state is also achieved by executing the init() function of the component.

© Copyright 2009 Hitex Development Tools GmbH

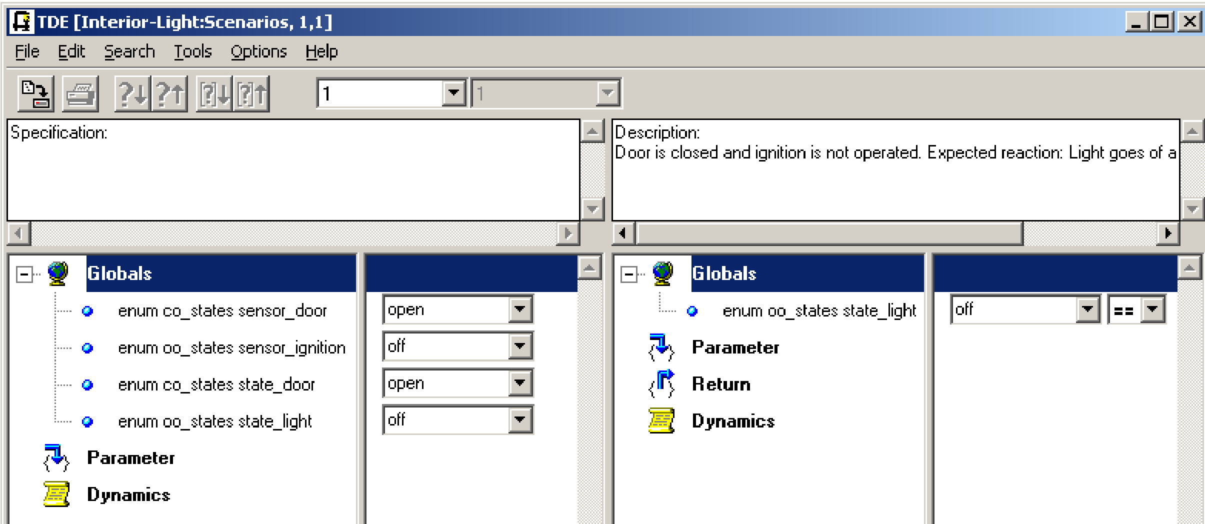


Fig. 25 Test data for the first scenario in the TDE

We use the same data also for the second scenario.

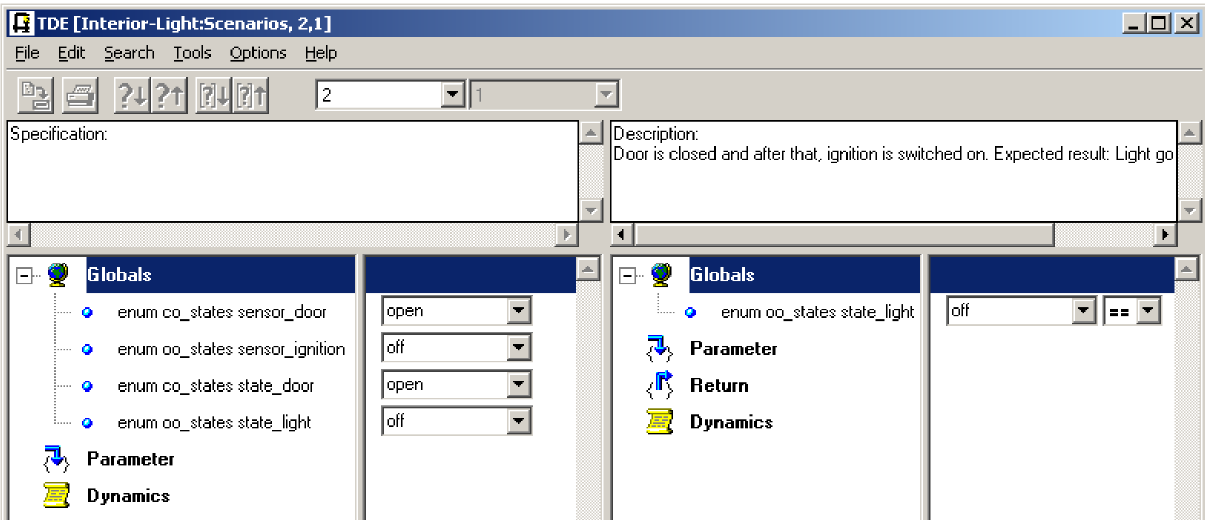
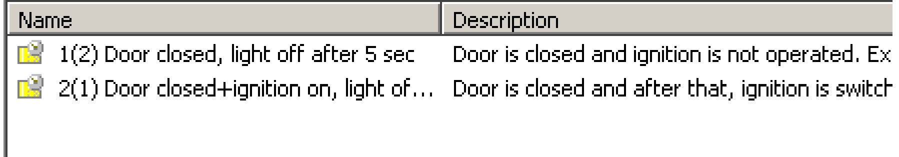


Fig. 26 Test data for the second scenario in the TDE

The scenarios are now yellow, because test data is present. The little clock icon is still grey because we did not yet specify any stimulation of the component within a test scenario. This will be the next step.



Name	Description
1(2) Door closed, light off after 5 sec	Door is closed and ignition is not operated. Ex
2(1) Door closed+ignition on, light of…	Door is closed and after that, ignition is switch

Fig. 27 Two scenarios with test data in Tessy's component window

© Copyright 2009 Hitex Development Tools GmbH

2.4.6.3 The Scenario Editor (SCE)

To design scenarios, i.e. test cases for components, the so-called Scenario Editor (SCE) is used.

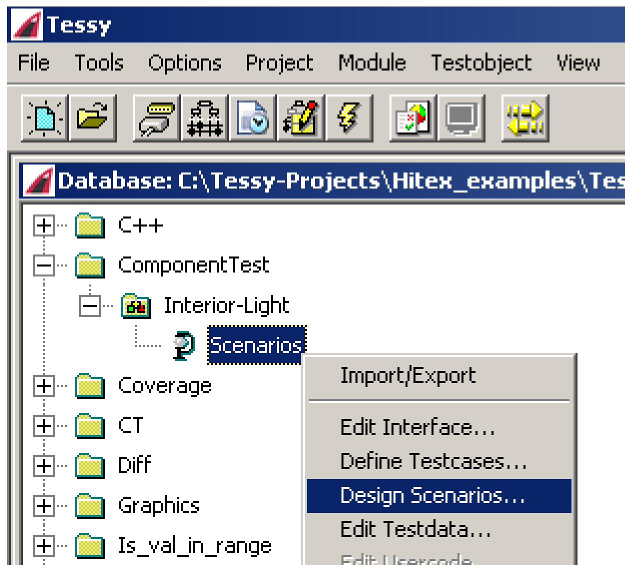


Fig. 28 Opening the Scenario Editor (SCE) by using the context menu

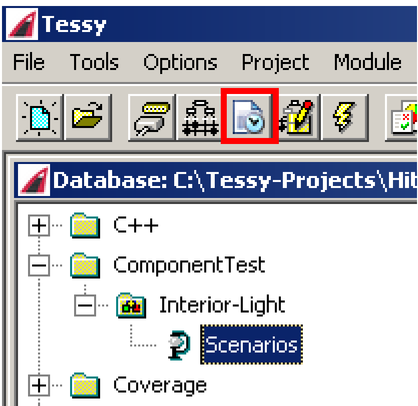


Fig. 29 Opening the Scenario Editor (SCE) by using the scenario editor's icon

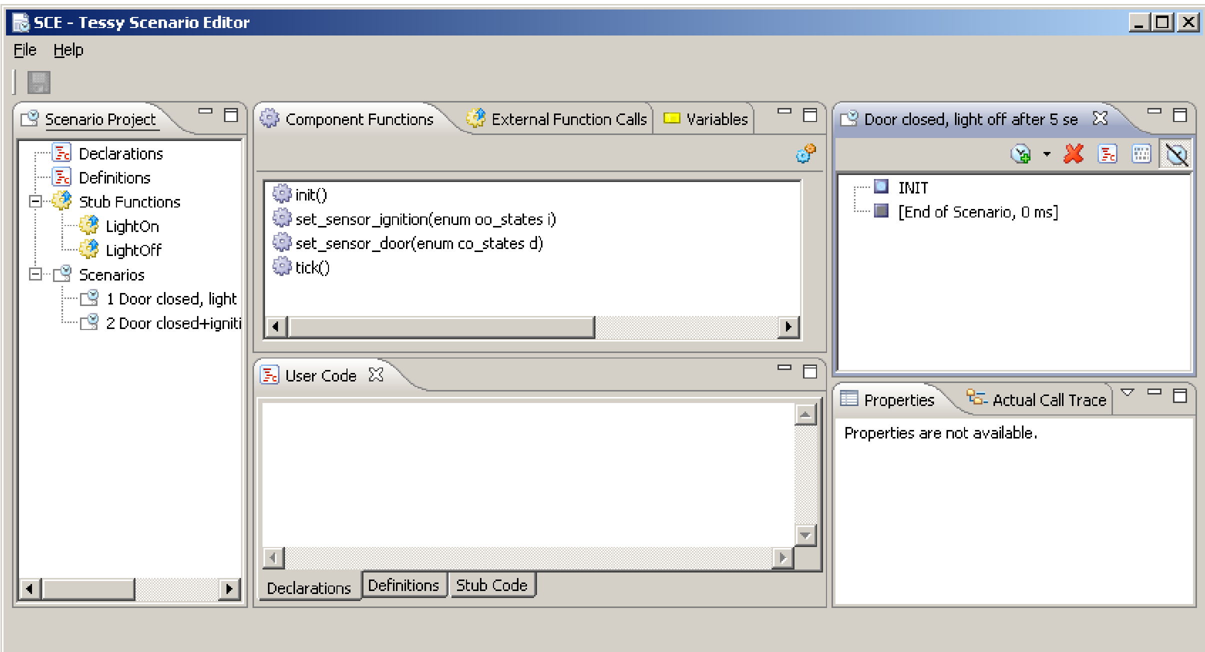


Fig. 30 Initial view of the Scenario Editor (SCE)

On the left hand side, in the Scenario Project, we see the two stub functions that we have specified in the TIE.

Below the stub functions, we see the two scenarios that we have defined in Tessy's component window.

In the middle, we see three tabs: Component Functions, External Function Calls, and Variables.

Component Functions

This tab lists the functions of the component that can be called from the outside of the component. The component functions stimulate the component. The component functions are used to design the scenario.

Please note that the static functions, e.g. iLightOff(), are not listed here. This is different to the section "Component Functions" in the TIE.

Setting the work task

As we know, the implementation of the component Interior-Light assumes that the function tick() is called every 10 ms. I.e. the function tick() is the work task or handler task or heartbeat of the component. To enable Tessy for temporal component testing, Tessy must know about this. I.e. we must specify tick() as work task for the component.

© Copyright 2009 Hitex Development Tools GmbH

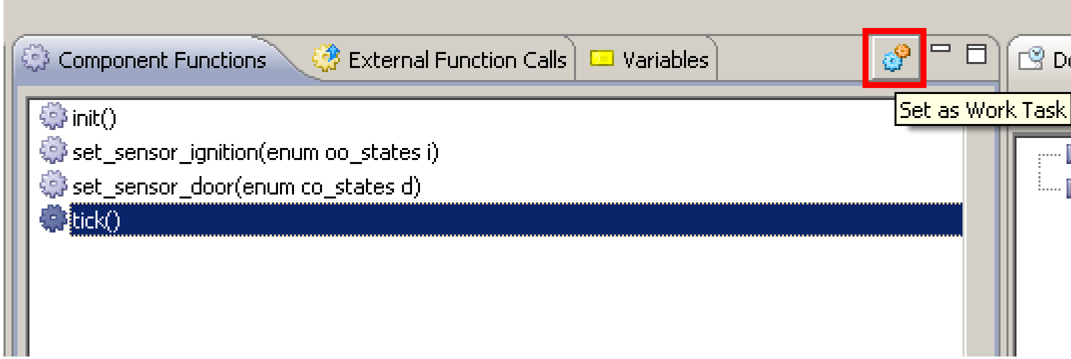


Fig. 31 Specifying tick() as work task for the component

To specify tick() as work task, select tick() and then click on the “work task” button, as in the figure above. As a result, the icon of tick() changes.

Please note:
Scenarios have to be present to be able to set the work task.

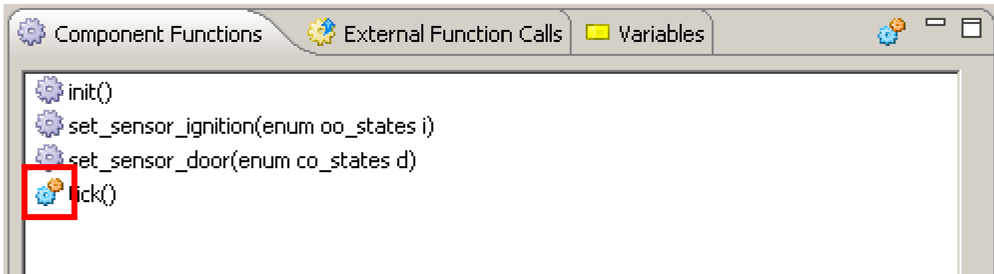


Fig. 32 tick() is set as work task for the component

External Function Calls

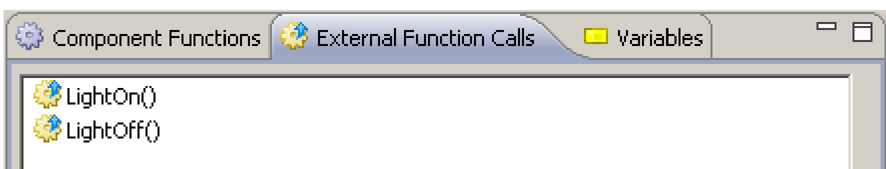


Fig. 33 The tab “External Function Calls”

The tab “External Function Calls” displays the two functions LightOn() and LightOff() that the component Interior-Light supposes in another component. Tessy provides stub functions for these two functions for component testing.

© Copyright 2009 Hitex Development Tools GmbH

Variables

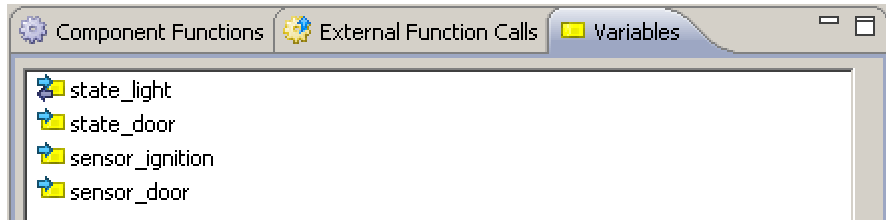


Fig. 34 The tab "Variables"

The tab "Variables" displays the variables. These variables were made visible in the TIE. The small arrows indicate the passing direction of the variables. The passing direction is "InOut" for state_light and "In" for the other variables. The passing directions were specified in the TIE.

Properties

To see the properties of the scenarios, select "Scenarios" in the "Scenario Project" tab on the left hand side of the SCE window.

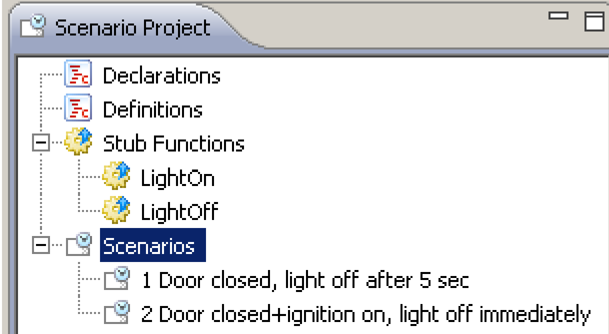


Fig. 35 Select "Scenarios" in the "Scenario Project" tab

Then the properties are displayed.

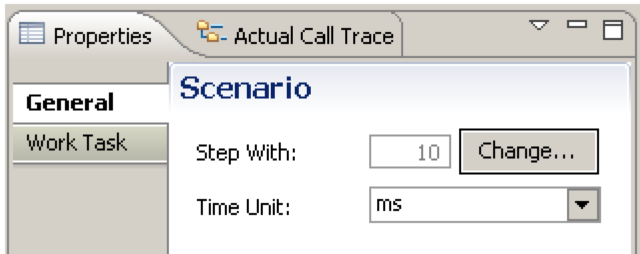


Fig. 36 The General Properties display the distance between calls to the work task

The Tessy default distance is 10 ms. This fits to the assumptions of the component Interior-Light. Therefore, it does not need to be changed.

© Copyright 2009 Hitex Development Tools GmbH

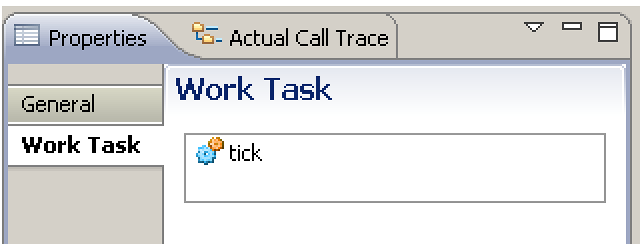


Fig. 37 The General Properties display the name of the work task

Select a certain scenario in the “Scenario Project” tab on the left hand side of the SCE window.

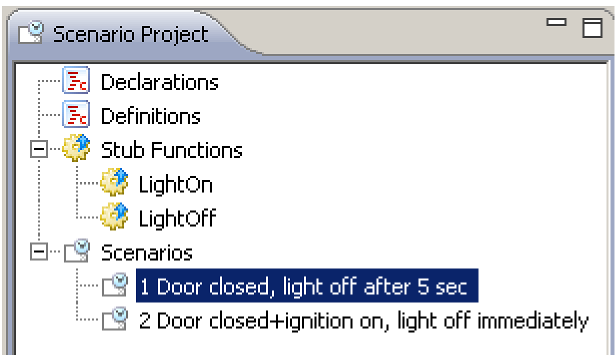


Fig. 38 Select a certain scenario in the “Scenario Project” tab

Then the properties of this scenario are displayed.

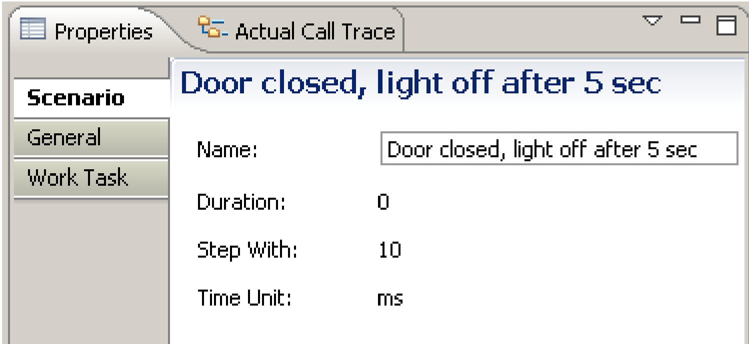


Fig. 39 Properties of a certain scenario

The duration of the scenario will be calculated automatically as soon as you build the scenario. It depends on the content of your scenario.

© Copyright 2009 Hitex Development Tools GmbH

2.4.6.4 Design Scenarios Using the Scenario Editor (SCE)

Scenarios (i.e. component test cases) are designed in the window in the upper right corner of the SCE.

First Scenario

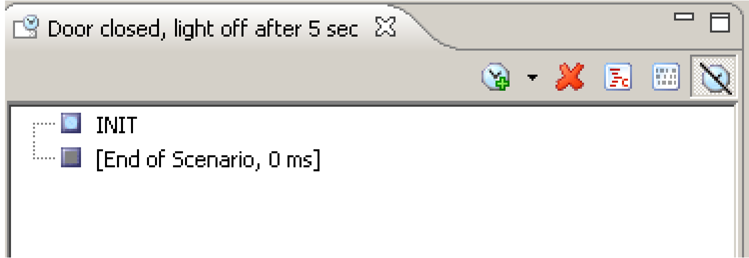


Fig. 40 Initial view of the first scenario

Because we want to test the temporal behavior of the component, we need to establish a time base. We do this be clicking several times on the button or by using the pull-down menu.

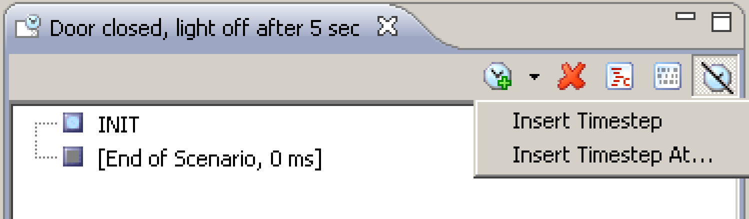


Fig. 41 How to insert time steps in a scenario

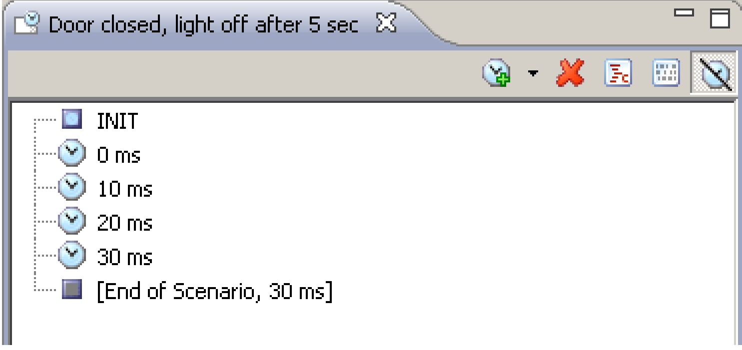


Fig. 42 The first scenario with a few time steps

The scenario above consists of 4 calls to the work task, i.e. tick() in our case. The calls occur at 0 ms, 10 ms, 20 ms, 30 ms simulated time. Nothing happens otherwise.

Obviously, it is possible to do something prior to the first call to tick() at 0 ms simulated time, i.e. between INIT and 0 ms. The idea is to do the initialization of the component at this time, if this should be required.

© Copyright 2009 Hitex Development Tools GmbH

Please keep in mind, that prior to INIT the variables are initialized with the input data for the scenario from the left hand side of the TDE.

To stimulate the component besides the calls to tick(), you can simply drag&drop a function from the Component Functions to the scenario.

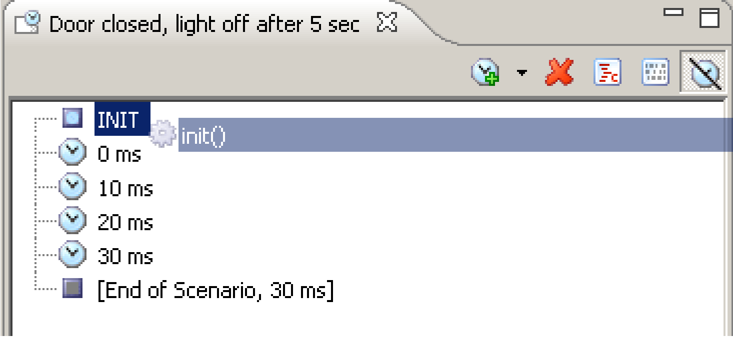


Fig. 43 The component function init() is dragged to the scenario

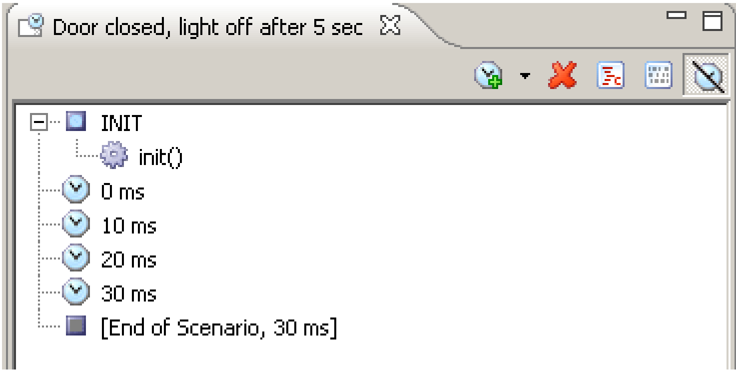


Fig. 44 Init() is now dropped to INIT

In the scenario above, the call to init() might be superfluous, because init() should set the variables to the same values as they will be set by Tessy due to the settings in the TDE. However, with different values in the TDE, it can be checked if the call to init() initializes the variables as expected.

Now we stimulate the component actually. We drag the component function set_sensor_door() to 30 ms simulated time.

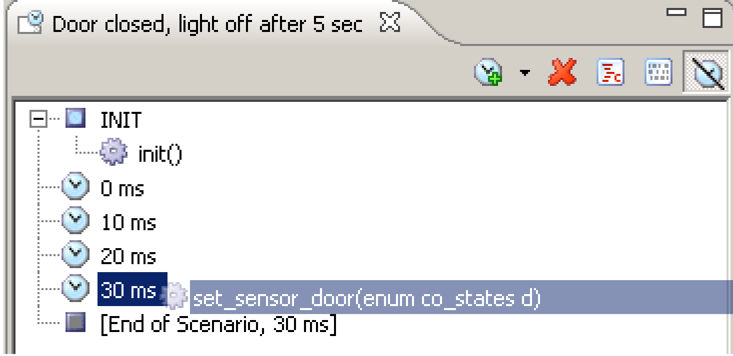


Fig. 45 set_sensor_door() is dragged to 30 ms simulated time

© Copyright 2009 Hitex Development Tools GmbH

With set_sensor_door() marked in the scenario, we can specify a value for the parameter of set_sensor_door() in the properties of set_sensor_door().

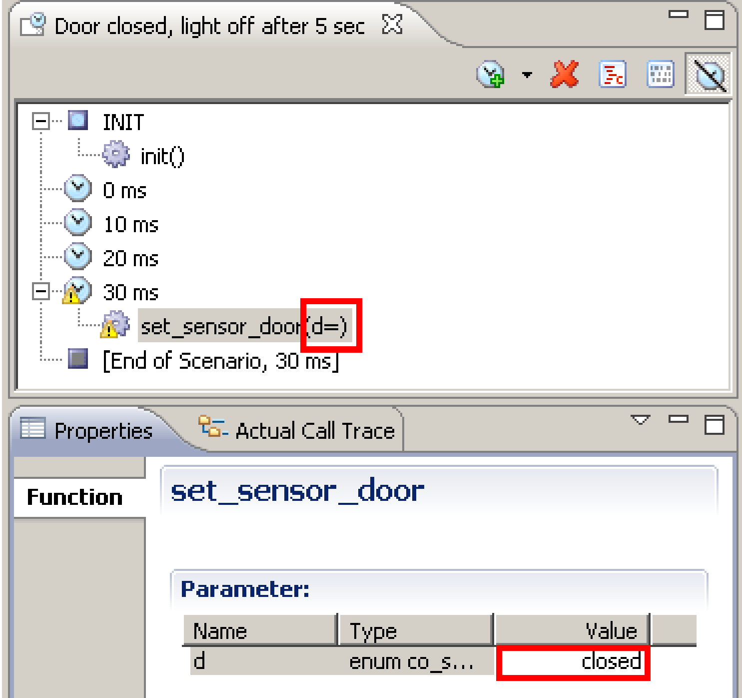


Fig. 46 The parameter of set_sensor_door() is set

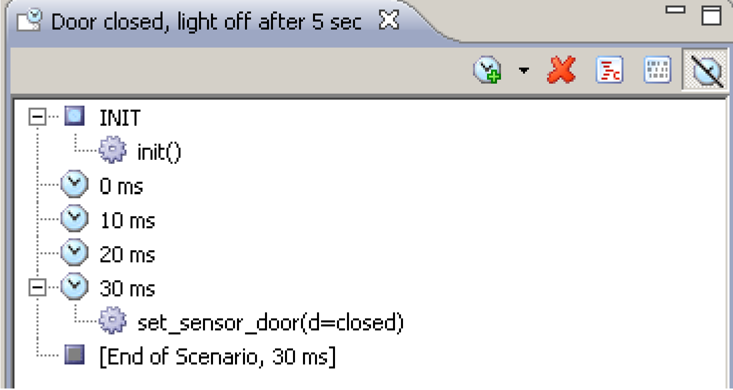


Fig. 47 The scenario with the stimulating call to set_sensor_door()

In the scenario above, after the fourth call to tick(), Tessy calls the component function set_sensor_door() with a parameter value of “closed”. This should cause the component to react by calling LightOn(). From the implementation of Interior-Light we know, that this call will happen one tick later, i.e. after Tessy has called tick() a fifth time, at 40 ms simulated time.

We can specify this expected result by dragging the function LightOn() from the “External Function Calls” tab to 30 ms simulated time.

© Copyright 2009 Hitex Development Tools GmbH

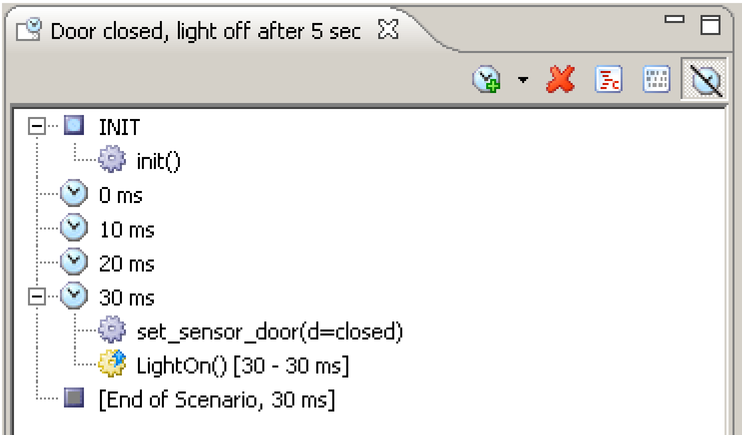


Fig. 48 The scenario with the call to LightOn() as expected result

The scenario above expects the call to LightOn() to happen in the time frame [30 – 30 ms], i.e. prior to the fifth call to tick(). However, we know that the call will occur one tick later, and we want to treat this behavior as correct.

With LightOn() marked in the scenario, we can extend the time frame.

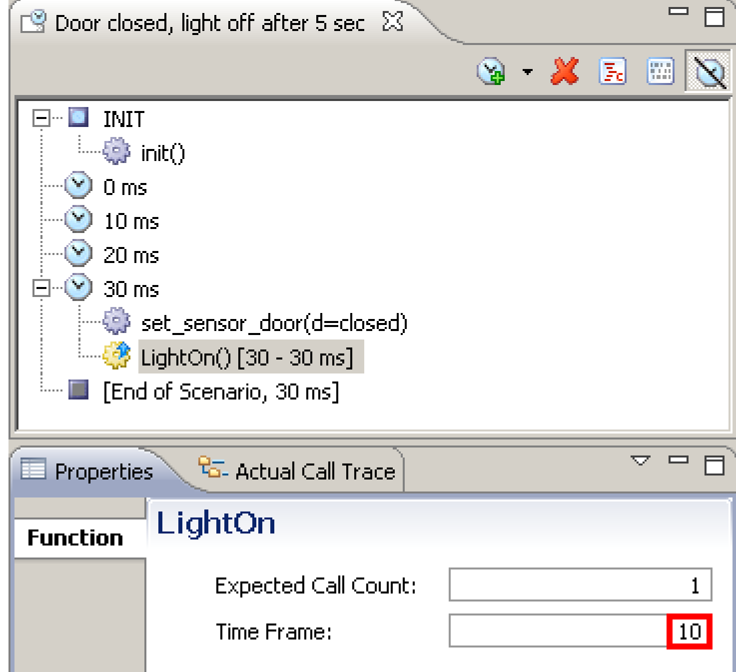


Fig. 49 The time frame for the call to LightOn() is extended by 10 ms

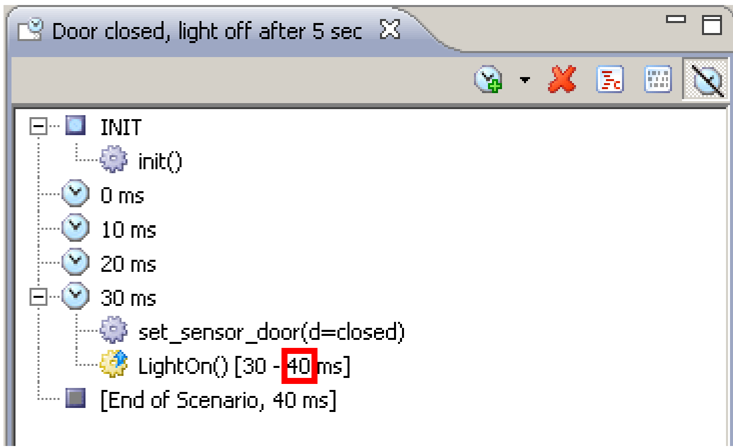


Fig. 50 The scenario with extended time frame for LightOn()

Besides the call to LightOn(), we also expect a call to LightOff(). Because in this scenario the ignition will not be operated, we expect the call to LightOff() to occur 5 seconds after the call to set_sensor_door() at the latest.

We drag LightOff() to 30ms and extent the time frame by 5000 ms.

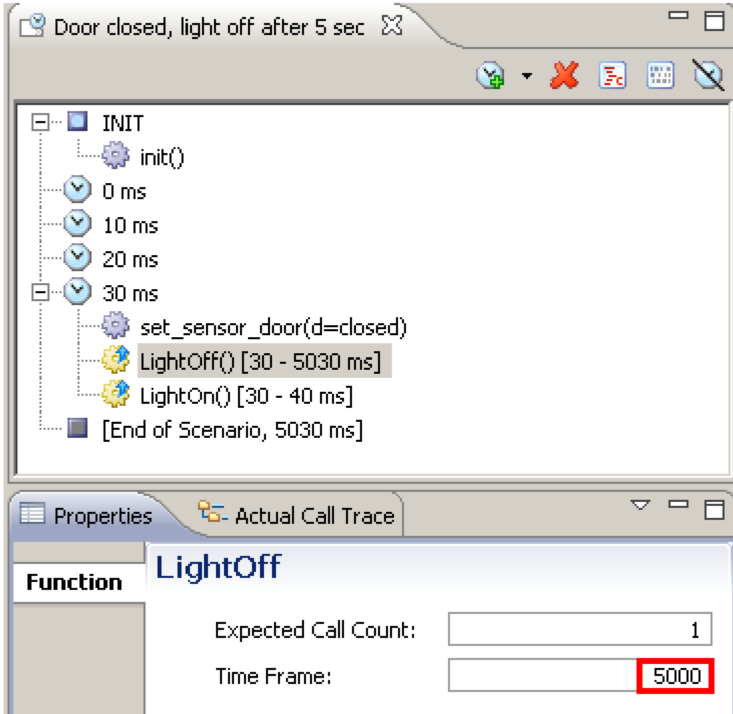


Fig. 51 The scenario with extended time frame for LightOff()

In the scenario above, we have specified that we expect the call to LightOff() after 5 seconds at the latest, i.e. we would take a call to LightOff() after, say, 4 seconds as a correct result.

Our scenario is now complete. We can save our work in the SCE (File > Save).

© Copyright 2009 Hitex Development Tools GmbH

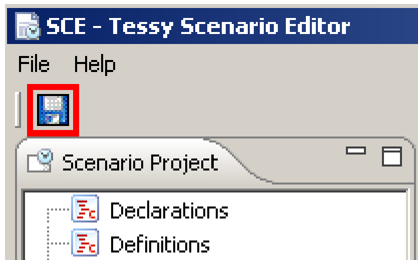


Fig. 52 Saving the work done in the SCE

Second Scenario

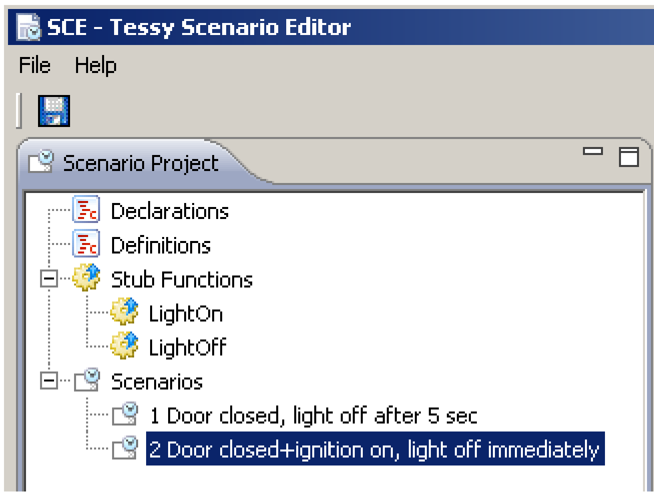


Fig. 53 We select the second scenario

As for the first scenario, we insert five time steps into the scenario. Then we drag&drop the component function set_sensor_door() to 10 ms simulated time. Then we set the value of the parameter to "closed". To specify the expected behavior, we drag&drop the function LightOn() from the "External Function Calls" tab to 10 ms simulated time. We extend the time frame for LightOn() by 10 ms. This is all done like in the first scenario.

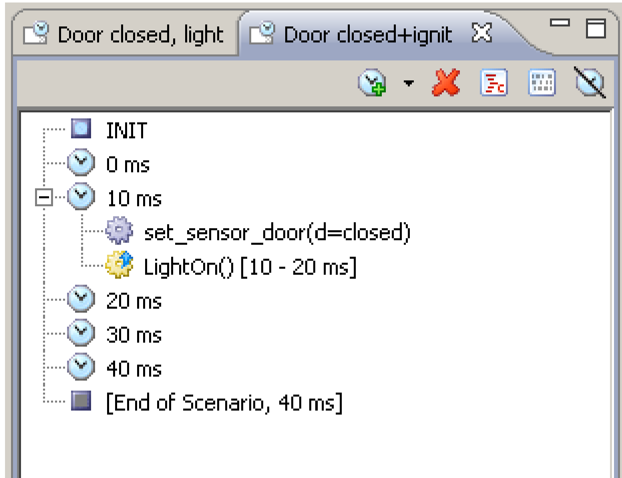


Fig. 54 The second scenario (intermediate state)

© Copyright 2009 Hitex Development Tools GmbH

In the second scenario, the ignition shall be operated. We could do this by using the function set_sensor_ignition() from the “Component Functions” tab. This would be very similar to using set_sensor_door(). We would set the parameter of set_sensor_ingnition() to “on” and not to “closed”, but that would be the only difference.

However, to demonstrate the use of variables in the scenario, we use the variable “sensor_ignition” for the purpose to switch the ignition on. We drag&drop the variable “sensor_ignition” from the “Variables” tab to 40 ms scenario time.

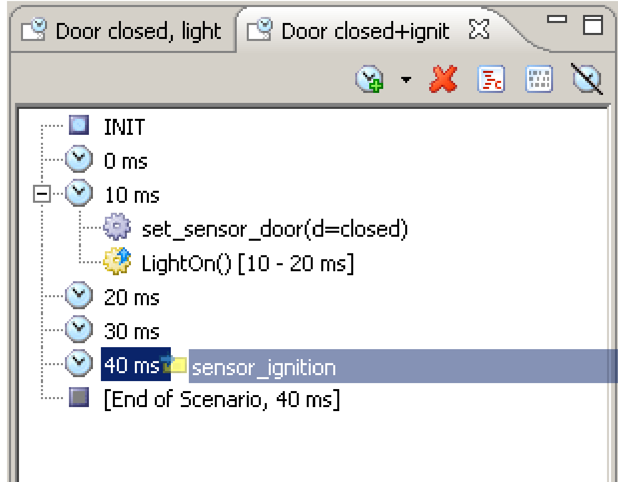


Fig. 55 The second scenario (intermediate state)

With sensor_ignition marked, we are able to set the input data for that variable to on.

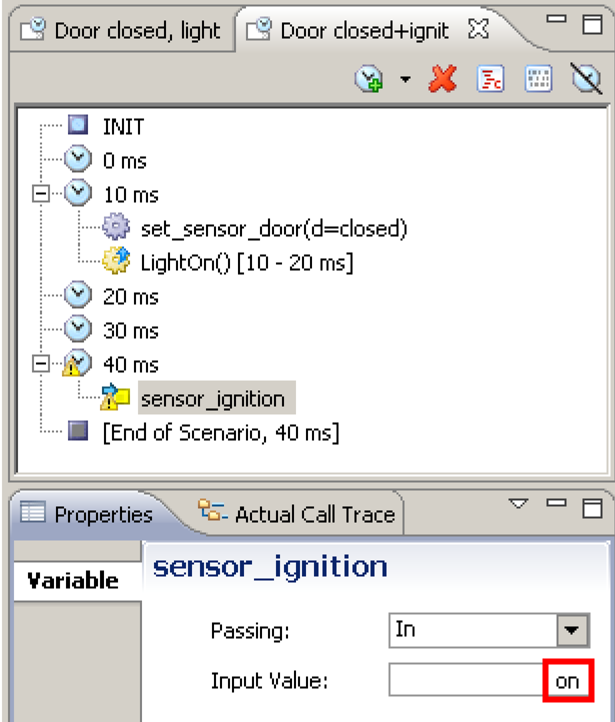


Fig. 56 The second scenario (intermediate state)

© Copyright 2009 Hitex Development Tools GmbH

For the component Interior-Light, setting of the variable sensor_ignition causes the same effect than calling the function set_sensor_ignition() with the parameter “on”. (Actually set_sensor_ignition() sets sensor_ignition.) The expected result of that action is the interior light going off immediately, i.e. after the next call to the work task tick(). This is specified in the scenario by drag&drop of the function LightOff() to 40 ms simulated time and by extending the time frame by 10 ms. This is done very similar as in the first scenario.

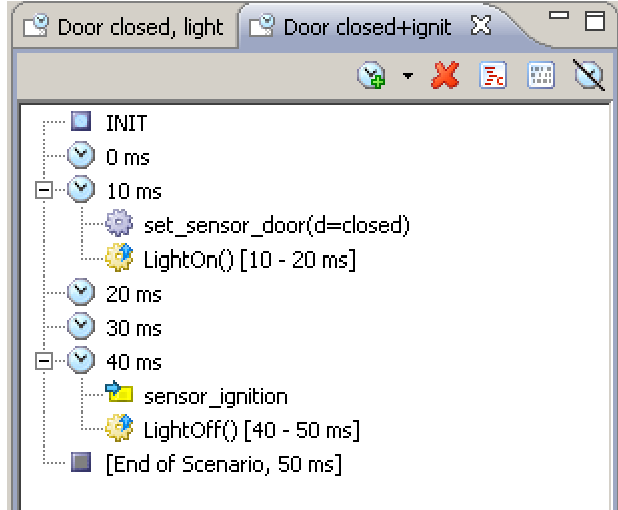


Fig. 57 The second scenario (final state)

Please note:
The second scenario does not comprise a call to init(). This does not cause the scenario execution to fail, because the initialization of the variables is done by Tessy, with the values that were specified in the TDE.

Save your work in the SCE.

2.4.7 Execute the Scenarios

After we have saved our work in the SCE, we select the scenarios in Tessy and then execute the tests (using the Tessy menu Testobject > Execute Test).

© Copyright 2009 Hitex Development Tools GmbH

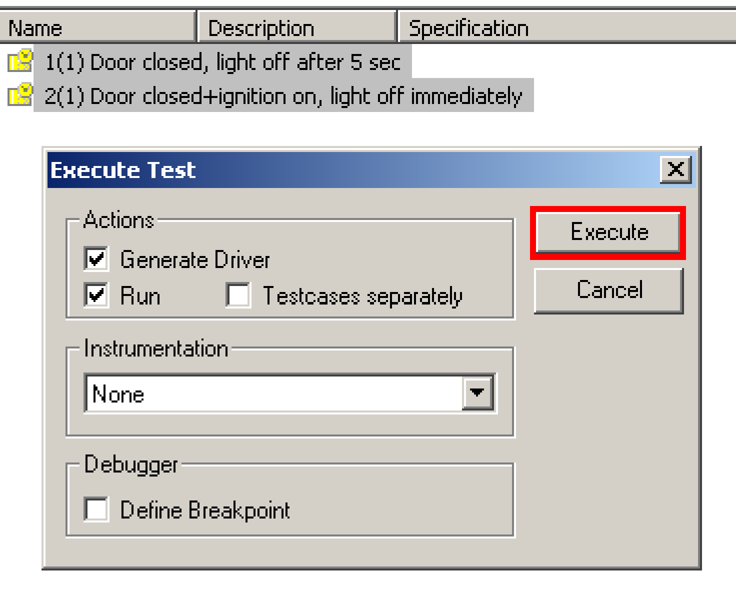


Fig. 58 Execute the test/scenario

2.4.8 View the Results

2.4.8.1 Overall Results

After the scenarios were executed, the color of the icons of the scenarios are changed to green. This indicates an overall “passed” verdict.

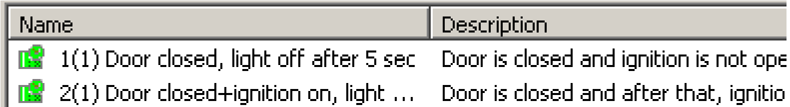


Fig. 59 Results in Tessy’s component window

2.4.8.2 Results in the TDE

Double-clicking on the icon of the first scenario opens the Test Data Editor (TDE) with the initial and final data for the first scenario.

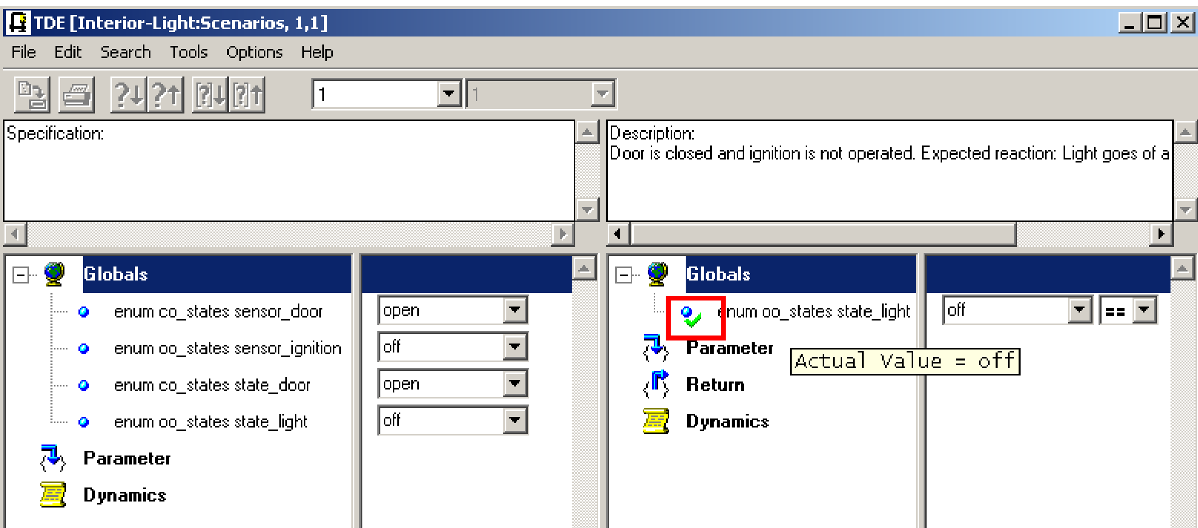


Fig. 60 Results in the TDE for the first scenario

Please note the green tick mark in the figure above.

The results in the TDE for the second scenario are the same as for the first scenario.

© Copyright 2009 Hitex Development Tools GmbH

2.4.8.3 Results in the TDE

The “Scenario Project” tab gives overall results for the two scenarios.

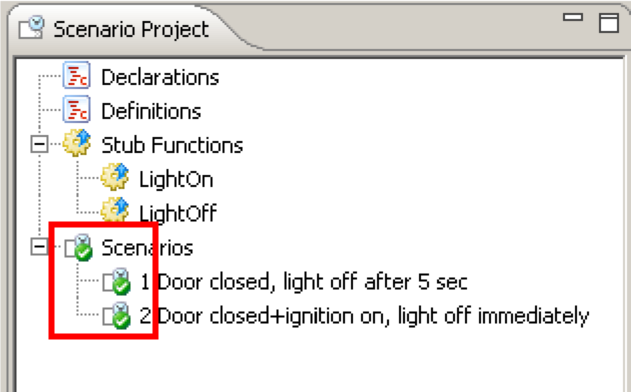


Fig. 61 Overall results in the SCE

Please note the green tick marks in the figure above.

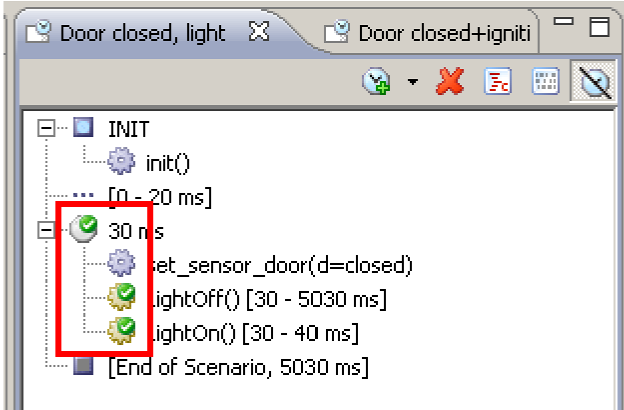


Fig. 62 Results of the first scenario in the scenario window of the SCE

Please note the green tick marks in the figure above.

For each scenario execution, Tessy records a call trace. To see the call trace belonging to a scenario, select the scenario in the “Scenario Projects” window.

© Copyright 2009 Hitex Development Tools GmbH

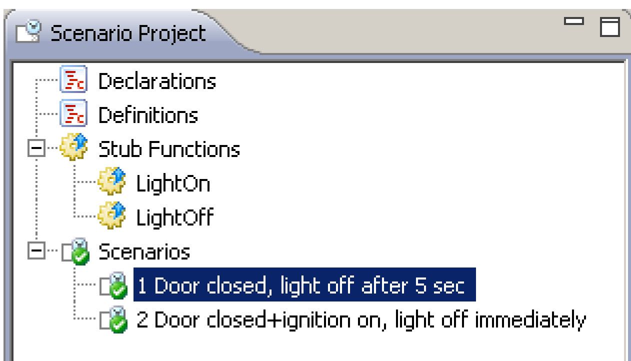


Fig. 63 Select scenario to see its call trace

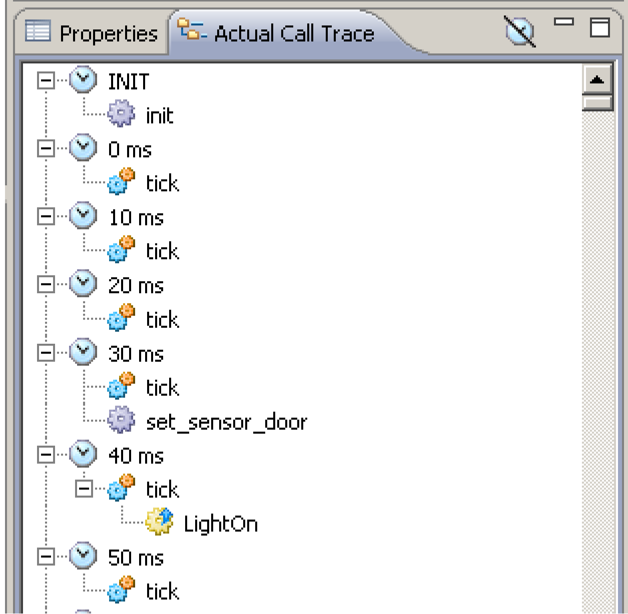


Fig. 64 Call trace of the first scenario of the SCE (initial view)

The call trace displays the stimulating calls and the calls to other components. It is possible to hide the time stamps containing only calls to tick() by using the button .

© Copyright 2009 Hitex Development Tools GmbH

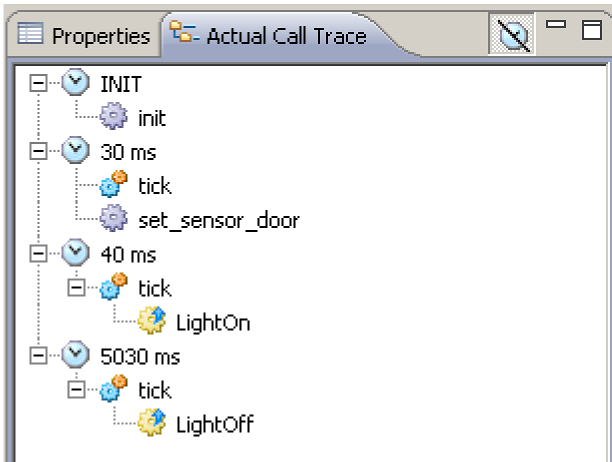


Fig. 65 Call trace of the first scenario of the SCE (collapsed view)

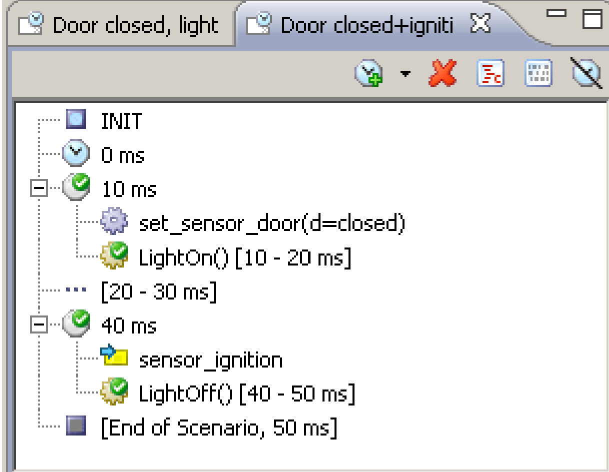


Fig. 66 Results of the second scenario in the scenario window of the SCE

© Copyright 2009 Hitex Development Tools GmbH

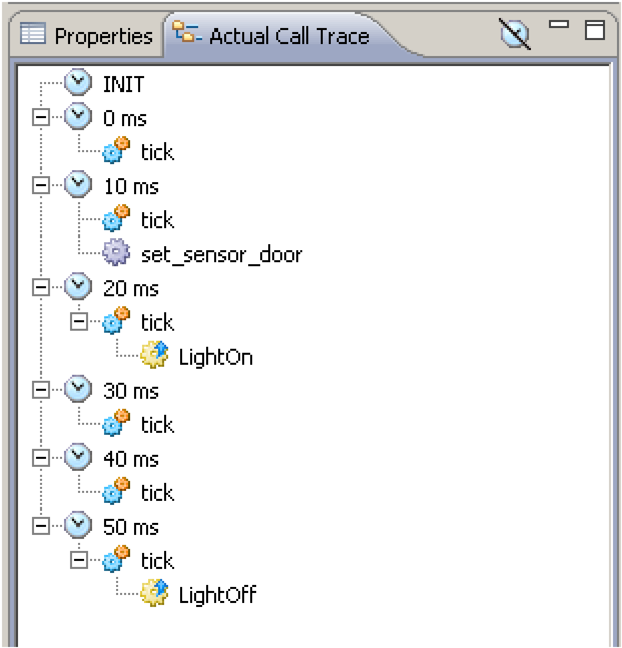


Fig. 67 Call trace of the second scenario of the SCE (initial view)

© Copyright 2009 Hitex Development Tools GmbH

3 Variations of the Example

This section shall give inspiration for own experiments (or for extensions of this tutorial in the future).

3.1 Check initialization

3.1.1 Data from the TDE

Change the initialization of the test input data of the scenarios in the TDE.

Scenario 1 still will pass, because init() provides the correct initialization in spite of “inappropriate” data in the TDE.

Scenario 2 probably will fail (depends on your new data in the TDE).

3.1.2 Data from init()

If init() initializes the variables to “inappropriate” values, the scenario will fail, regardless of the values in the TDE.

3.2 Check Final State

Change the expected result in the TDE for the variable state_light to “on”. This will cause the scenario to fail.

3.3 Check Intermediate State

Check the value of the variable state_light during a scenario execution. Verify that state_light is set to “on” at some point in time.

3.4 Check That Event Is Not Happening

Check that an event is never happening or not happening in a given time frame. Such an event could be a call to an error routine or the like. Or check that LightOff() is not called until five seconds are passed.

Hint: Use events with call count 0, e.g. LightOff() CallCount=**0** [30-5020 ms]. This will require LightOff() CallCount=1 [5020 – 5030 ms] to check that the light goes off as required.

3.5 Unite LightOn() and LightOff()

Use Light() instead of LightOn() and LightOff(). The parameter of Light() indicates if the light should be on or off.

```
void Light(int OnOff)
{
        Light = OnOff;
}
```

Fig. 68 Suggested implementation of Light()

Use Eval macros in the stub functions to find out if Light() was called with the correct value in its parameter.

© Copyright 2009 Hitex Development Tools GmbH

3.6 Use an Second Component

Provide light_control.c as implementation of the component Light-Control.

```
void LightOn(void)
{
        Light = 1;
}
void LightOff(void)
{
        Light = 0;
}
```

Fig. 69 Suggested contents of light_control.c

Now the “extended” component comprises two source files. How does the interface look like?

Now the value of the variable Light is the actual result of the scenario. How to check this?

3.7 Change the Time Base

Change the time base in the implementation of interior_light.c from 10 ms to 20 ms. This will cause the scenarios to fail.

Change the time base in the scenario also from 10 ms to 20 ms to remedy the problem.

Caution: This is not reversible!

Caution: Lower resolution causes inaccuracy.

3.8 Several Handler Functions

Is it possible to have several handler functions for a component? Can these different handler functions be called at different timely distances?

This would allow integrating a “producer” component and a “consumer” component to a single (bigger) component. Producer and consumer could run at a different pace → waits, overflows, etc.

3.9 Specify Scenarios in the CTE

Use the Classification Tree Editor CTE to specify scenarios. Import scenarios to Tessy.

© Copyright 2009 Hitex Development Tools GmbH

4 Versions Used

Work was started in May 2009, using a beta version of Tessy V2.9.

All examples were executed using the Gnu C compiler delivered with Tessy.

5 The Author

Frank Büchner studied Computer Science at the Technical University of Karlsruhe (TH). Since graduating, he has spent more than twenty years working in different positions in the field of embedded systems. Over the years, he specialised in the testing of embedded software and passes on his expertise regularly on congresses and seminars. He is currently with Hitex Development Tools GmbH, Karlsruhe.

Frank Büchner

Dipl.-Inform.

Hitex Development Tools GmbH

Greschbachstr. 12

76229 Karlsruhe

0721 / 9628 – 125

frank.buechner@hitex.de

Any comments to this paper are welcome!

■

© Copyright 2009 Hitex Development Tools GmbH