Test-Driven Development of a PID Controller

Thomas Dohmke and Henrik Gollee, *University of Glasgow*

Because the slUnit framework lets users apply TDD to dynamic systems, it's particularly appropriate for developing controllers.

In 1922, Nicolas Minorsky introduced the proportional-integral-derivative controller,[1] which is commonly used for control systems. PID controller tuning is often governed by empirical techniques such as the formulas of John Ziegler and Nathaniel Nichols,[2] by solving an optimization problem,[3] or by simple trial-and-error with specific input stimuli. In addition, graphical programming languages such as Simulink (www.mathworks.com/products/simulink) are becoming increasingly important for control-system design, mainly in the automotive industry, the aerospace industry and other industries such as process control. They allow model-based development,[4] in which the model is the central artifact supporting an iterative development style and supplementing the traditionally mathematical approaches to control-system design.

This iterative development style and the tuning techniques are comparable to test-driven development because the user first defines a criterion—for example, the expected reaction time to an input stimulus—and then develops the controller. So, it's possible to adopt TDD, which to date has been applied mainly to development of traditional software systems, to control-system design. Here we describe our slUnit model-based testing framework (http://slunit.dohmke.de) and demonstrate how to use it to design a controller for a simplified vehicle system. Because slUnit lets us apply TDD to dynamic systems, it's particularly suitable for control-system design.

A model-based testing framework

One main aspect of TDD is the definition and (automated) execution of tests. To aid this process, researchers developed the xUnit family of testing frameworks for most programming languages.[5,6] These frameworks employ an object-oriented architecture with a set of key classes that lets developers easily create, execute, and manage hierarchies of tests and prevent duplication.[7]

When we designed slUnit, the main issue was how to translate xUnit's class-based hierarchy into Simulink's structure. In Simulink, the *block* element implements the basic functionality such as operators. The *subsystem* element can aggregate blocks as well as other subsystems. The *model* element is equivalent to a single file and represents the top-level subsystem.

Figure 1 shows two example Simulink systems. The xUnit architecture isn't one-to-one portable to these elements. So, we use four basic testing patterns[5] to describe slUnit: *assertions*, *test methods*, *all tests*, and *fixtures*. Our basic testing terminology is in accordance with the *IEEE Standard Computer Dictionary*.[8]

Another fundamental difference between xUnit and Simulink is the latter's time-dependent behavior. With xUnit, we can test a function call's return values with static arguments, while a Simulink system's signals and states always depend on the point of time at which the simulation step is calculated. The slUnit framework must take this into account.

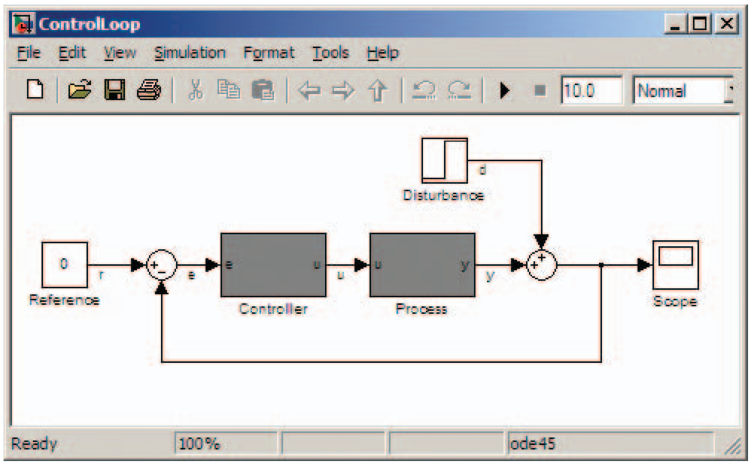
Assertions

The assertion is the basic element for automatic testing. It evaluates whether a result or output value equals the expected value, thereby checking whether a test is passed or failed. In xUnit, you can express a simple assertion as

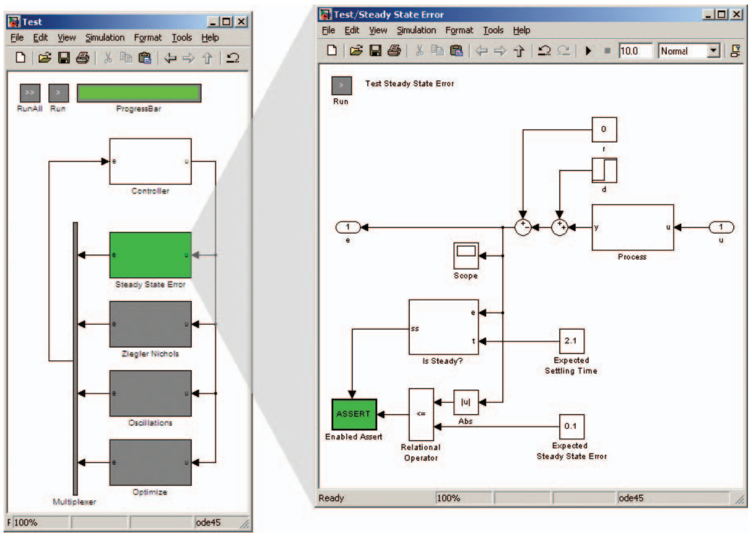
```
assert(y == foo(x));
```

where `y` is the expected value and `foo(x)` is a function that's the test objective. The assertion fails if the Boolean expression is false, and the test stops at the assertion's line. However, this doesn't stop the next test's execution because tests are independent of each other in xUnit. If all assertions pass, the test passes, which is typically indicated by a green progress bar in the framework's GUI. If one or more fail, the test fails, which is usually indicated by a red bar. With the red bar, a list of failures appears, including each failure's expected result, its line, or a user-defined message, so that the developer can find the problem's position and cause.

slUnit realizes this concept with the assert block, which has one Boolean input similar to the `assert` method. The block's background color indicates the assertion's state—green for success, red for failure. This lets users evaluate a test without needing an additional GUI or a command line output, and makes locating the failed assertion easier. Unlike the xUnit assertion, the assert block must consider its input signal's time-dependent behavior. The assert block fails if the input signal becomes false and won't pass for the rest of the time, but this won't stop the test. So, assertions are independent from each other. This is typical for Simulink-based tests, which mostly analyze the time-dependent behavior of different signals. If consecutively executed assertions are required, users can build them through additional Simulink components such as enabled subsystems or state machines.



(a)



(b)

Figure 1. Two example Simulink systems: (a) a closed-loop control system and (b) a test for steady-state error.

Test methods

In xUnit, tests are usually organized as methods and classes, because most xUnit frameworks are implemented in an object-oriented programming language. A single test case is represented as a test method, which is part of a class inherited from the basic class `test class`. Typically, test methods that test the same test objective are methods of the same

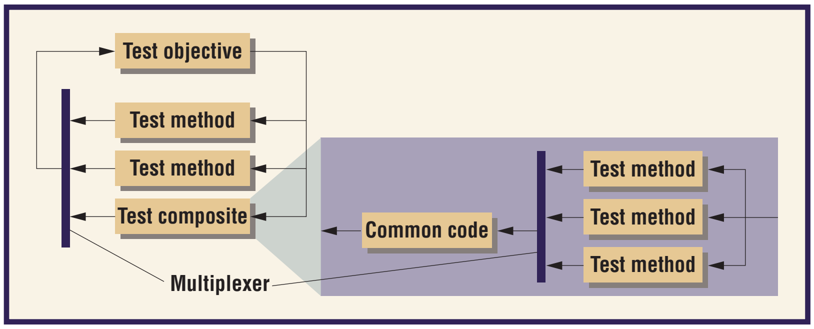


Figure 2. An slUnit model's architecture.

class. Test classes are aggregated in a test suite, which is treated itself as a test class. This means that it can also be part of another test suite.

slUnit uses subsystems to map this class-based hierarchy to Simulink's structure. We propose that the test objective is part of the model in the form of at least one subsystem. So, we must mark the tests with a certain mask type (an attribute to classify Simulink subsystems) to distinguish them from other subsystems used either as structural components or as the test objective. Furthermore, a subsystem can also be a test class with nested subsystems that are the test methods, or it can be a test suite consisting of nested subsystems that are test classes, and so on. Because there's no difference between the subsystem for a test class or a test suite, we call these subsystems *test composites*. The model itself also represents a test composite; the only difference is that it also contains the test objective.

Figure 2 shows such an slUnit model's architecture. The major element is the multiplexer block, which switches the active test method's outputs to the test objective's inputs (because the test objective can be stimulated by only one test method at a time). Each test composite includes one multiplexer block, which handles the switching of its test methods and further test composites, allowing a nested architecture of tests.

All tests

In xUnit, GUIs let users directly execute test methods, test classes, and test suites without creating extra code. So, it's important that users can run a single test method, a single test class with one or more test methods, a single test suite with one or more test classes, or a number of test suites.

With slUnit, the tests are executed as the model is simulated, which means that the model's signals and states are calculated for a number of simulation steps. Each simulation should execute only one of the defined test methods. Therefore, the assert blocks must be activated and deactivated automatically at runtime, so that only those assert blocks are enabled whose test method is executed. The results are displayed as the subsystem's background color and are inherited from all assertions, which means that the subsystem block is red if at least one assert block failed. With an additional simulation command, users can execute all tests of a test composite. The results are displayed as the subsystem's background color and inherited from the results of the included test methods or test composites.

Fixtures

Fixtures share common code between test methods, usually in combination with the test class aggregating all these methods. The xUnit family provides the methods `setUp()` and `tearDown()`, which are called before and after each test method to manage the fixture. They prevent duplication and ensure that each test is independent even when it uses external resources. This is because `tearDown()` is called always after each test method executes, independently of whether that method was successful.

In slUnit, fixtures are realized by aggregating the test methods in a test composite, so we can share common code by extracting it to the same level as the test methods. The test methods' outputs are then the inputs of not only the test objective but also the common code, whose outputs themselves become inputs of the test objective and/or the test methods. The test composite in figure 2 uses this concept. For this model-based common code, there's no need to release the fixture except for interaction with the Matlab workspace, which we can do with a callback function.

Test-driven control-system design

Consider the closed-loop control system in figure 1a. Here, the system's output, y, is fed back and compared to a reference value, r, to obtain an error signal, e. A controller then uses the error signal to obtain a suitable control signal, u, which is the input to the process.

Closed-loop control can implicitly compensate for system disturbances. For example, in a vehicle model, drag or the road's slope can cause external disturbances. In this case, controller design aims to make the vehicle undergo a desired acceleration while compensating for the external disturbances.

Development involves three main steps. First, we analyze the system by defining one or more tests to describe its performance or, in general, its requirements. Second, we design the controller so that the system passes these tests. To achieve this, we specify another test to find the parameters systematically. Finally, we remove duplication and evaluate the system.

Step 1: Analysis

In classical approaches to control-system design, the real system is usually described by a mathematical model expressing the relationship between the input variable $u(t)$ and the output variable $y(t)$, where t is the time variable. We can represent a vehicle's acceleration characteristics in a simplified form by this second-order differential equation:

$$\frac{d^2y(t)}{dt^2} + 28\frac{dy(t)}{dt} + 280y(t) = 280u(t - T) \quad (1)$$

where $T = 0.1$ seconds is the system's time delay.

To specify the desired system performance, we evaluate the output resulting from specific input stimuli. Such a specific input can be the unit step input, described through the mathematical function

$$f(t,t_s) = \begin{cases} 0 \text{ for } t < t_s \\ 1 \text{ for } t \geq t_s \end{cases} \quad (2)$$

where t_s is the time at which the value of f changes from 0 to 1. Applying such a unit step input as the output disturbance of the closed-loop system whose reference value is set to 0 would result in some output response. We can express this response's characteristics by performance criteria such as the *steady-state error*, which is the difference between the reference value $r(t)$ and the final output value y_{ss}.[9]

We can now specify a first test that expresses the closed-loop system's desired behavior (see figure 3). With this *steady-state error test*, we can start the TDD cycle. We write the test—or more specifically, model it—as a Simulink subsystem (see figure 1b). The "make it compile" procedure is done with the "update diagram" function, which checks for correct data types, sample times, and other modeling errors.

The update process fails because we haven't yet implemented a controller. In fact, the build process usually fails in xUnit if we define a test for a function whose prototype doesn't exist. So, creation of the test objective is part of the TDD cycle. We create a simple controller that sets the value of $u(t)$ to the value of $e(t)$:

$$u(t) = e(t) \quad (3)$$

We then run the test by simulating the model. Figure 4a shows the response. The test fails because the output response's final value is 0.5. To make it pass, we must extend the controller, which happens in the next step.

Description:	**We define the test through the closed-loop system, whose disturbance value is stimulated by a unit step at time $t_s = 1$ s and whose error value is checked for the steady-state error.**
Output:	**The difference $e(t)$ between the reference value $r = 0$ and the output $y(t)$ of the process model in equation 1.**
Input:	**The control variable $u(t)$, which is connected to the process model's input.**
Assertions:	**We determine the steady-state value y_{ss} by observing the changes of $y(t)$ for three consecutive simulation steps.** **Changes in $y(t)$ must be lower than a threshold $dy_{exp} = 0.1$, and $e(t)$ must be less than the expected steady-state error $e_{exp} = 0.1$.**

Figure 3. The steady-state error test.

Step 2: Design

To fulfill the previous test's requirements, we implemented a PI controller[1]—that is, a PID controller without the derivative part:

$$u(t) = Pe(t) + I\int_0^t e(\tau)d\tau \quad (4)$$

where P is the proportional gain and I is the integral gain. An increase of P results in a faster response, while I eliminates the steady-state error.

To specify the gains, we could use empirical design methods, but we want to present a more systematic, test-driven way. So, we remove the feedback from the system and use a step function to stimulate the process itself to determine the maximum slope R and the lag time L from the step response (see figure 5). We can estimate the slope using a derivative and a maximum block, while simultaneously calculating the lag time from the values of t, y, and R:

$$R^n = \max\left(R^{n-1}, \frac{y^n - y^{n-1}}{t^n - t^{n-1}}\right) \quad (5)$$

$$L^n = \begin{cases} t^n - \dfrac{y}{R^n} & \text{for } R^n > R^{n-1} \\ L^{n-1} & \text{otherwise} \end{cases} \quad (6)$$

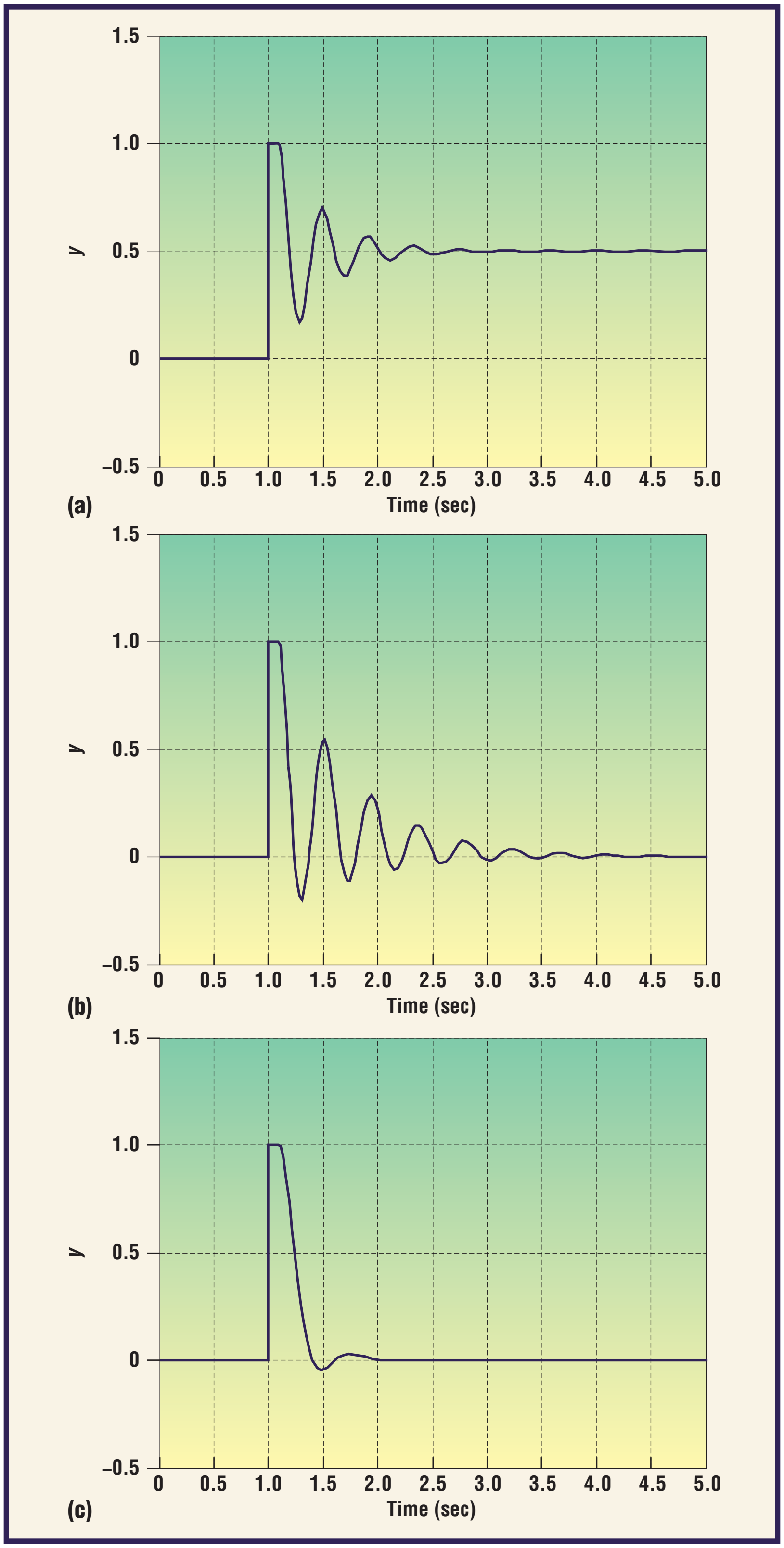


Figure 4. The closed-loop system's step responses after the (a) steady-state error test (P = 1, I = 0), (b) Ziegler-Nichols test (P = 1.1, I = 2.9), and (c) optimize test (P = 0.5, I = 4.1). P stands for proportional gain; I stands for integral gain.

The superscripts n and $n - 1$ denote the values of the current and the previous simulation step.

For the gain of a PI controller, Ziegler and Nichols proposed two formulas based on R and L:[2]

$$P = 0.9\frac{1}{RL} \tag{7}$$

$$I = 0.3\frac{P}{L} \tag{8}$$

So, the next test (see figure 6) asserts that neither R nor L is zero and calculates the value of P and I. This *Ziegler-Nichols test* has no inputs and outputs in the context of the controller, because the test objective is the process model. For two different test objectives, we would usually create two testbeds—that is, two Simulink systems. However, in this case we choose to work with a single testbed, because testing the process model is part of the controller design and therefore documented in the testbed as a part of the development process.

For our example, the Ziegler-Nichols test results in the parameters P = 1.1 and I = 2.9. We can now implement the controller using these parameters; consequently, both the steady-state and Ziegler-Nichols tests pass (see figure 4b). So, the major difference between the two kinds of tests is that the first specifies our system's behavior ("what it should do") and the second helps us implement the controller in a systematic, traceable way ("how it is done").

Step 3: Evaluation

As we mentioned before, the third step has two parts. First, we remove duplication, which is the last step of the TDD cycle. In our example, duplication occurs in the use of the process model, which is implemented for both tests. So, we can separate the process model as a separate subsystem and use it in a fixture.

Second, we evaluate the developed system for software aspects—for example, using black-box testing to determine whether all the embedded system's requirements are fulfilled or white-box testing to analyze branch coverage. Furthermore, we analyze control aspects such as stability—for example, with Bode diagrams or Nyquist charts.

At this stage, one iteration of the development process has completed. Depending on the results, we can start the cycle again with step 1—for example, to enhance the controller for different reference values, disturbance variables, or process models, especially if the process is nonlinear.

Further design iterations

Our closed-loop control system's response (see figure 4b) isn't completely satisfying. A further design iteration could therefore aim to reduce the number of oscillations, defined through the *number-of-oscillations test* (see figure 7). This test specifies another behavior of our system for the development cycle's analysis step. It fails because the current system has four oscillations (which is too many for such a system).

We therefore proceed with the design step to modify the controller. We create the *optimize test* (see figure 8) to minimize this cost function:

$$J = \int_{t_0}^{t_1} \left| e(t) \right| dt \qquad (9)$$

J is the *performance index* and expresses the control error over the defined time interval. We expect that a reduction of *J* also leads to fewer oscillations. We omit details of the optimization, which modifies *P* and *I* recursively, because they aren't this article's focus.

Although the optimize test reruns itself a number of times, it's independent from other tests, expects no special execution order of tests, and considers the fixture's integrity. So, we can handle it like a common xUnit test method.

This optimization leads to a controller with $P = 0.5$ and $I = 4.1$, which passes all tests. Figure 4c shows the corresponding closed-loop step response.

The development of embedded control systems in Simulink usually continues with automatic code generation, the build process, and several tests: software-in-the-loop (SiL), hardware-in-the-loop (HiL), and system tests in the real environment of the controller—here, the vehicle. Our test-driven control-system design cycle integrates into this process because it doesn't interrupt the system's model-based architecture. For example, the automatic code generation can be done directly from the testbed, either for the test objective or for the test cases to reuse them for HiL tests.

Furthermore, the next development cycle usually considers the results of SiL, HiL, or the system tests. With TDD, such changes cause one or more new tests, which describe the modified requirements either with a modified or different process model or with a stimulus from measurements. Then, we can enhance the controller—for example, with a separate set of parameters or an additional filter. At the same time, all previously defined tests ensure the controller's basic behavior, leading to higher quality for the complete system.

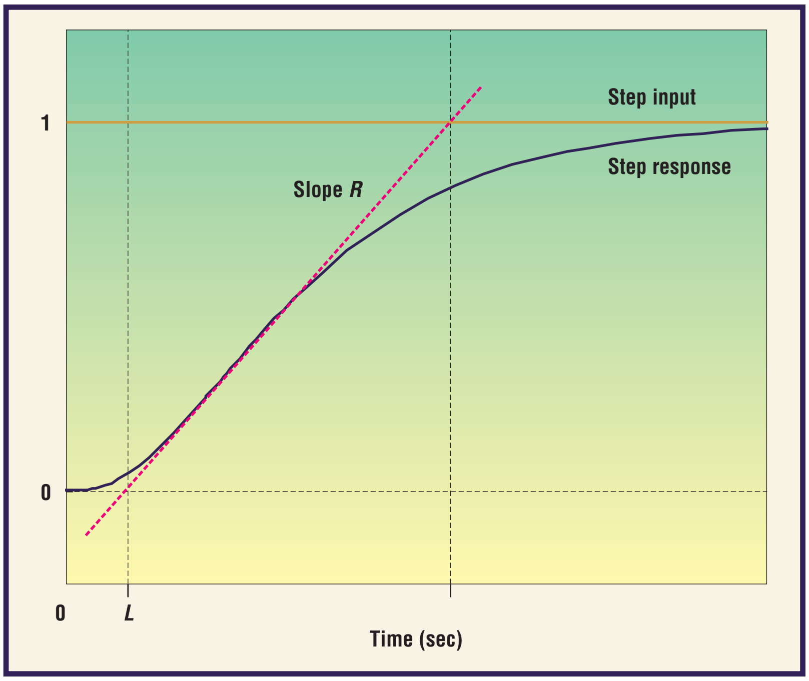


Figure 5. Determining the Ziegler-Nichols parameters from the process's unit step response. *R* is the maximum slope; *L* is the lag time.

Description:	We define the test through the process model, whose input is stimulated by a unit step at the time $t_s = 0$ s. We use the given formulas (equations 7 and 8 in the main text) to estimate the maximum derivative and the lag time. We evaluate the result through two assertions that also enable the calculation of *P* and *I*.
Output/Input:	None.
Assertions:	*R* must be greater than zero. *L* must be greater than zero.

Figure 6. The Ziegler-Nichols test.

Description:	We define the test through the closed-loop system, whose disturbance value is stimulated by a unit step at the time $t = 1$ s and whose error value is evaluated for the number of oscillations.
Output/input:	See the steady-state error test (figure 3).
Assertions:	We count the number of oscillations of $e(t)$ by evaluating the amplitude and the sign of the derivative of $e(t)$. This number must be less than or equal to the expected value $n_{exp} = 2$.

Figure 7. The number-of-oscillations test.

Description: We define the test through the closed-loop system, whose disturbance value is stimulated by a unit step at the time $t = 1$ s and whose error value is integrated according to equation 9 in the main text. The test aims to minimize the value of J between $t_0 = 1$ s and $t_1 = 30$ s by modifying the values of P and I and rerunning itself until J has reached a local minimum.
Output/Input: See the steady-state error test (figure 3).
Assertions: P must be greater than zero. I must be greater than zero.

Figure 8. The optimize test.

Further research will compare this approach with other design methods. In addition, we believe we can extend our approach to different types of real-time systems and evaluate how to use TDD for their development.

About the Authors

Thomas Dohmke is a postgraduate student at the Department of Mechanical Engineering and a member of the Centre for Systems and Control at Glasgow University. His research interests include software development processes and control-system design, focusing on the development of automotive driver assistance and safety systems. He works for Robert Bosch GmbH. He received his graduate-engineer degree in computer engineering from the Technical University Berlin. He's a student member of the IEEE. Contact him at the Centre for Systems and Control, Univ. of Glasgow, Glasgow, G12 8QQ, Scotland, UK; thomas@dohmke.de.

Henrik Gollee is a lecturer at the Department of Mechanical Engineering and a member of the Centre for Systems and Control at Glasgow University. His research interests include the application of advanced control methods in assistive technology and human-machine interfaces, focusing on applications in spinal cord injury rehabilitation. He received his PhD in systems and control from the University of Glasgow. He's a member of the IEEE. Contact him at the Centre for Systems and Control, Univ. of Glasgow, Glasgow, G12 8QQ, Scotland, UK; h.gollee@mech.gla.ac.uk.

References

1. N. Minorsky, "Directional Stability and Automatically Steered Bodies," *J. Am. Soc. of Naval Engineers*, vol. 34, no. 2, 1922, pp. 280–309.
2. J. Ziegler and N. Nichols, "Optimum Settings for Automatic Controllers," *Trans. ASME*, vol. 64, no. 11, pp. 759–768, 1942.
3. F. Lewis, *Applied Optimal Control and Estimation*, Prentice-Hall, 1992.
4. A. Rau, "Model-Based Development of Embedded Automotive Control Systems," PhD dissertation, Dept. of Computer Science, Univ. of Tübingen, 2002.
5. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
6. G. Meszaros, *XUnit Test Patterns*, Addison-Wesley, 2007.
7. P. Hamill, *Unit Test Frameworks*, O'Reilly, 2004.
8. *IEEE Std. 610*, *IEEE Standard Computer Dictionary*, IEEE, 1990.
9. K. Dutton, S. Thompson, and B. Barraclough, *The Art of Control Engineering*, Addison-Wesley, 1997.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.