Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 1537

Model-in-the-Loop and Software-in-the-Loop Testing of Closed-Loop Automotive Software with Arttest

Norman Hansen[1], Norbert Wiechowski[1], Alexander Kugler[1], Stefan Kowalewski[1], Thomas Rambow[2] and Rainer Busch[2]

Abstract: In this paper, we present Arttest, a tool for functional testing of block diagrams developed with MATLAB/Simulink. We introduce testing concepts for closed-loop tests of automotive software on model and software level, the integration of the concepts into a signal specification language and correspondent tool support. Furthermore, we show the applicability of the concepts and the test execution automation based on an example for model-in-the-loop and software-in-the-loop tests.

Keywords: Software Verification, Software Testing, MIL, SIL

1 Introduction

In the automotive industry, major innovations are nowadays driven by software[Br07]. Over the past years, functionality realized by software grew from basic headlight control to advanced systems, such as Active Brake Assist (ABA) and Electronic Stability Control (ESC), interacting with multiple sensors, actuators and other systems. Since software may directly influence the driving behavior by controlling actuators, e.g., brakes, software failures may result in damage to passengers and to the environment. To increase the safety of software controlled systems in the automotive domain, the ISO 26262[IS11] recommends extensive testing for safety critical software. In particular, testing safety critical software using the final hardware is highly recommended by the ISO 26262. However, removing a software failure in late development phases implicates high costs[Jo12]. In order to detect and remove software faults as early as possible in the development process and thus decrease costs, software can be tested on model level[BK08, La04].

When performing tests, the model of the System Under Test (SUT) is often tested in combination with a plant model, describing the behavior of the environment of the SUT[Wi17, Na04]. By feeding the signals from the plant model to the SUT and the outputs of the SUT to the plant model, a closed-loop system is created. For instance, the plant model

[1] Lehrstuhl Informatik 11 - Embedded Software, Ahornstraße 55, 52074 Aachen
[hansen, wiechowski, akugler, kowalewski]@embedded.rwth-aachen.de
[2] Ford Research and Innovation Center Aachen, Süsterfeldstraße 200, 52072 Aachen
[trambow, rbusch1]@ford.com

doi:10.18420/in2017_154

of an ESC system could model the behavior of a car for a specific driving scenario as shown in Figure 1.

Depending on the SUT and its complexity, all input signals might be connected and fed back to a plant model as shown in Figure 1. Consequently, there are no input signals to the closed-loop system which could be stimulated by tests. When performing Hardware-in-the-Loop (HIL) tests, signals of closed-loop systems generated by a plant model can be overridden with signals specified by a test engineer enabling the stimulation of the controller model and fault injection tests[Na04]. To enable similar tests in early development stages, i.e., on Model-in-the-Loop (MIL) and Software-in-the-Loop (SIL) level to save costs by early fault removal, we target the replacement of plant model signals by specified signals.

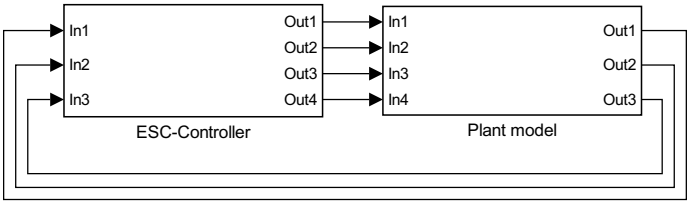

Fig. 1: Closed-loop system of an ESC controller with plant model

In Section 2, we present a signal specification language enabling closed-loop tests which may dynamically react to conditions being satisfied. In addition, the language enables arbitrary internal signals of the closed-loop model being tested to be overridden, e.g., for fault injection. The implementation of closed-loop tests using the presented specification language is supported by Arttest, a tool for the creation and automated execution of requirement based functional tests[Wi17]. In Section 3, we describe the test automation of MIL and SIL tests and present an application of the concepts using an example in Section 4. Related work is discussed in Section 5 while Section 6 concludes this work.

2 Signal Specification Language

To implement tests for closed-loop systems, a language is required which is able to express, that internal signals of a model are overridden with specified values for a given duration. Overriding internal signals may further be triggered by events, e.g., a condition containing signals of the model being satisfied. We designed a signal specification language with formal syntax and semantics that fulfills these requirements. The language is designed to resemble natural language so that tests can be easily understood, even without extensive prior knowledge of the language.

2.1 Syntax

A simplified excerpt of the language's grammar for a selection of basic language constructs used to describe signals is given in Table 1 using EBNF[Sc96] notation.

⟨*TC*⟩	=	⟨*actions*⟩
⟨*actions*⟩	=	**{** ⟨*action*⟩ **}**
⟨*action*⟩	=	⟨*wait*⟩ | ⟨*setto*⟩ | ⟨*simultaneously*⟩ | ⟨*rampto*⟩ | ⟨*when*⟩ | ⟨*override*⟩
⟨*wait*⟩	=	"wait for"{⟨*number*⟩} "seconds "
⟨*signal*⟩	=	"<"⟨*string*⟩ ">"
⟨*setto*⟩	=	⟨*signal*⟩ "set to ["⟨*number*⟩ "]"
⟨*rampto*⟩	=	⟨*signal*⟩ "ramps to ["⟨*number*⟩ "] within {"⟨*number*⟩ "} seconds"
⟨*when*⟩	=	"when ["⟨*condition*⟩ "] during {"⟨*number*⟩ "} seconds ("⟨*actions*⟩ ")"
⟨*override*⟩	=	"override"⟨*signal*⟩ ("on"| "off")
⟨*condition*⟩	=	⟨*string*⟩

Tab. 1: Excerpt of the signal specification language grammar in EBNF

The signal specification language is organized in actions which compose a test case (TC). Table 1 shows the syntax for *wait*, *setto*, *rampto*, *when* and *override* actions. The non-terminal symbol ⟨*number*⟩ defines arbitrary integer or decimal values and the non-terminal symbol ⟨*string*⟩ is restricted to valid MATLAB expressions. The ⟨*condition*⟩ symbol allows the specification of conditions in MATLAB syntax, i.e., expressions evaluating to $true$ or $false$ including signal names or function calls to MATLAB functions. Signal names are restricted to valid MATLAB variable names. Based on the syntax, we define the formal semantics of the signal specification language.

2.2 Semantics

A specification of signals *spec* is a n-tupel of actions $a_1, ..., a_n$ with $a_{i,i\in\{1...n\}} \in \mathbb{A}$, $\mathbb{A}$ being the set of all actions and $n \in \mathbb{N}_{\geq 0}$. Actions can be classified in three categories, actions influencing signal values directly, e.g., by changing signal values, actions activating or deactivating the overriding of internal signals and actions which are hierarchically composed of other actions. Actions modifying signal values are for instance *setto* and *rampto* actions. *Override* actions are of the second category and *when* actions are hierarchically composed of other actions. A *when* action a_{when} may be hierarchically composed such that $child(a_{when}) : \mathbb{A} \rightarrow \mathbb{A}^m, m \in \mathbb{N}_{\geq 0}$. For instance, $child(a_{when}) = ('wait\ for\ \{2\}\ seconds',$ $'< signal\_b >\ set\ to\ [4]')$ for *when* action a_{when} as given in Listing 1.

```
when [signal_a > 0] during {4} seconds then (
        wait for {2} seconds
        <signal_b> set to [4]
)
```

List. 1: Hierarchical composition of a *when* action

Consider τ to be the sampling time of the SUT and $\omega \in \mathbb{N}_{\geq 0}$ to be the number of time steps τ for the worst case duration of *spec*. Let $\mathbb{S}_{spec}$ be the set of signals $s : string \rightarrow \mathbb{R}^{\omega}$ defined by *spec* and $signal(a) : \mathbb{A} \rightarrow \mathbb{S}$ the function as given in Table 2, defining the signals whose values are influenced by action a.

a	$signal(a)$
$< s >$ set to $[y]$	s
wait for $\{x\}$ seconds	$\emptyset$
$< s >$ ramps to $[y]$ within $\{x\}$ seconds	s
when $[cnd]$ during $\{x\}$ seconds $(a_1,...,a_m)$	$(signal(a_1), ..., signal(a_m))$
override $< s >$ on	s
override $< s >$ off	s

Tab. 2: Definition of $signal(a)$ with $a, a_i \in \mathbb{A}$ and $s \in \mathbb{S}_{spec}$

The function $start(a) : \mathbb{A} \rightarrow \mathbb{N}_{\geq 0}$ assigns a starting time to all actions a with $start(a_1) := 0$, a_1 first action of the specification and $start(a_i) = start(a_{i-1}) + length(a)$ for all actions of the same hierarchy level. If there is no preceding action a_{i-1} on the same hierarchy level, then $start(a_i) = \kappa + 1$, κ being the first sampling time satisfying the condition of the *when* action containing a_i.

Let $length(a) : \mathbb{A} \rightarrow \mathbb{N}_{\geq 0}$ be the function defining the length, in number of equidistant sampling points, of each action according to Table 3.

a	$length(a)$
$< s >$ set to $[y]$	0
wait for $\{x\}$ seconds	$\lceil \frac{x}{\tau} \rceil$
$< s >$ ramps to $[y]$ within $\{x\}$ seconds	$\lceil \frac{x}{\tau} \rceil$
when $[cnd]$ during $\{x\}$ seconds $(a_{w_1},...,a_{w_m})$	$\lceil \frac{x}{\tau} \rceil$ if $cnd \mid\!=_{start(a),start(a)+\lceil \frac{x}{\tau} \rceil} false$, else $\kappa - start(a) + \Sigma_{i=1}^{m} length(a_i)$
override $< s >$ on	0
override $< s >$ off	0

Tab. 3: Definition of $length(a)$ with $a \in \mathbb{A}$, $x, y \in \mathbb{R}$, $\kappa \in \mathbb{N}_{\geq 0}$ and $s \in \mathbb{S}_{spec}$

$\lceil value \rceil$ denotes the rounding of $value$ to the next value in $\mathbb{N}$ in direction of $+\infty$ if $value \notin \mathbb{N}$. $cnd \mid\!=_{b,e}$ with $b, e \in \mathbb{N}_{\geq 0}$ denotes the evaluation of cnd to either $false$ or $true$ within the time interval starting at sampling point b and ending at sampling point e such that $cnd \mid\!=_{b,e} true$ if $\exists k \in [b, e]\ cnd(k) \mid= true$ with $\kappa = inf(\{k \in [b, e] | cnd(k) \mid= true\})$. For instance, assume the condition $cnd := signal\_a + signal\_b > 1$ with signal_a and signal_b sampled at rate 0.01 as shown in Figure 2. $a + b > 1 \mid\!=_{0,400} false$ since the sum of signal_a and signal_b is not greater than one for the first 400 sampling times. However, $a + b > 1 \mid\!=_{300,700} true$, since the sum of both signals is greater than one for at least one sampling point within the interval [300; 700], with sampling point $\kappa = 600$ being the first sampling point causing condition cnd to be true.

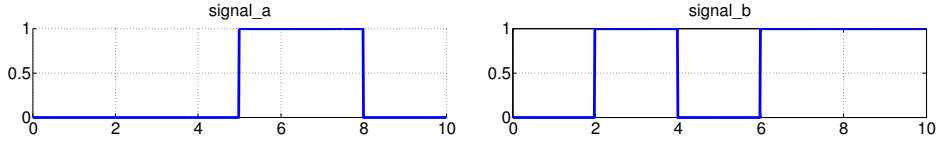

Fig. 2: Example for two signals *signal_a* and *signal_b* being used within a condition

a	$val(a)$
$< s >$ set to $[y]$	$s(start(a)) = y$
wait for $\{x\}$ seconds	$\forall s \in \mathbb{S}_{spec}, \forall i \in \{1, 2, ..., length(a)\}$, $s(start(a) + i) = s(start(a))$
$< s >$ ramps to $[y]$ within $\{x\}$ seconds	$\forall i \in \{1, 2, ...\lceil \frac{x}{\tau} \rceil\}$, $s(start(a) + i) = s(start(a)) + i * (\frac{y - s(start(a))}{length(a)})$ $\forall s' \in \mathbb{S}_{spec} \backslash \{s\}, \forall i \in \{1, 2, ..., length(a)\}$, $s'(start(a) + i) = s(start(a))$
when [cnd] during $\{x\}$ seconds $(a_1,...,a_m)$	if $cnd \mid =_{start(a), \lceil \frac{x}{\tau} \rceil}$ $true$ then $val('waitfor\{' \tau * \kappa '\}') \circ val(child(a))$ κ being the sampling time satisfying cnd else $val('waitfor\{'x'\}seconds')$

Tab. 4: Definition of $val(a)$ with $x, y \in \mathbb{R}$ and $s \in \mathbb{S}_{spec}$

As shown in Table 3, the length of *when* actions depends on whether and at which point in time the condition *cnd* is satisfied during the test execution. If the condition of a_{when} is never satisfied within the time interval $[start(a), start(a) + \lceil \frac{x}{\tau} \rceil]$, the child-actions $child(a_{when}) = (a_{w_1}, ..., a_{w_m})$ of a_{when} are not considered. Otherwise, the actions $(a_{w_1}, ..., a_{w_m})$ are interpreted sequentially with $start(a_{w_1})$ being the simulation time, denoted by $\kappa \in [start(a), start(a) + \lceil \frac{x}{\tau} \rceil]$ where *cnd* evaluates to *true* for the first time. Consequently, the length of a test may vary depending on conditions being satisfied or not during test execution.

The semantics of signal specification $spec = (a_1, ..., a_n)$ is given by sequential application, denoted $\circ$, of valuation function $val(a)$ to the actions of *spec*, i.e., $val(spec) = val(a_1) \circ ... \circ val(a_n)$. Valuation function $val(a)$ is described in Table 4 for actions a having an influence on the specified signals of the test and $s(0) = 0, \forall s \in \mathbb{S}$, zero being the initial value for every signal s.

As Table 4 shows, there is no valuation influencing signal values for *override* actions. *Override* actions determine the time and duration when internal signals are to be overridden with the values specified by actions which influence the signal to be overridden and follow the action enabling the override. Switching from the simulated to the specified values for a signal s is triggered by the action $override < s > on$. When interpreting $override < s > off$, the test execution switches back to the simulated values of signal s.

We distinguish between two different signal categories, reference signals and stimulus signals. Reference signals are compared with simulated signals and based on the comparison results, a test either passes or fails. Thus, reference signals do not influence the simulation. In contrast, stimulus signals are fed into the model, either as direct inputs to a model or to override internal signals.The language enforces a strict differentiation between both categories and correspondent signal specifications by dividing the test into two sections as shown in Listing 2.

```
Test:
Step 1:
Step 2:

Acceptance:
Criterion 1:
Criterion 2:
```

List. 2: Sections for stimulus and reference signal specification

The first section is the *Test* section, structured into sequentially executed *Steps* which use the introduced language for stimulus signal specification. Note, that the *Step* keyword followed by an identifier is only a structural element and has no impact on the semantics of the presented signal specification.

The second section is the *Acceptance* section, structured into *Criterions* using the introduced language for reference signal specification. A *Criterion* matches exactly one *Step* with $start(a_{c1}) = start(a_{s1})$, a_{c1} being the first action of the *Criterion* and a_{s1} the first action of the matching *Step*. If a *Criterion* has a longer duration than the matched *Step*, the specified reference signals are cut to the length of the step. Using *when* actions, reference signals may be specified depending on occuring events. However, since reference signals influence only the acceptance of a test and not the test execution, *override* actions may not be used within *Criterions*.

3 Closed-Loop Test Automation with Arttest

Arttest is a stand-alone application created with the intent to ease and automate test related activities such as test specification, test execution and generation of a test report. Arttest tests are based on the language presented in Section 2 with Arttest providing features such as content completion, syntax highlighting and visual previews of the specified signals. Since Arttest focuses on MIL and SIL tests in the automotive domain where Simulink is widely used for model-based software development, MATLAB/Simulink is the test execution platform supported by Arttest. Figure 3 depicts the test process in Arttest. After project creation and setup, a test harness is generated automatically based on the chosen model to test, before tests are specified in Arttest by a tester. Subsequently, when test execution is triggered, tests are executed, evaluated and test reports are generated fully automatically.

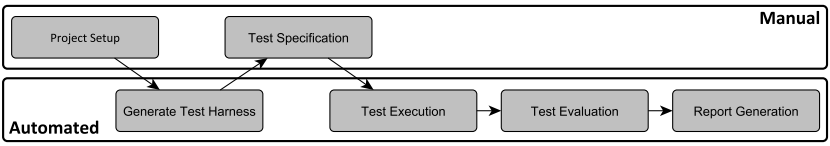

Fig. 3: Arttest Test Process

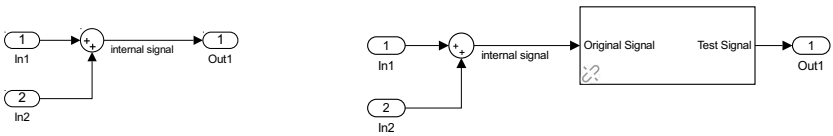

Fig. 4: Internal signal of the harness (left) being enriched by an Arttest block (right) to switch between simulated and specified signals during test execution

The automated harness generation extends the model or subsystem to be tested with Simulink blocks required by Arttest to feed specified signals to the model. When considering open-loop tests which do not react to events by adapting stimulus signals dynamically, it is sufficient to feed signals to the inputs of the SUT using *From Workspace* blocks and to log the output signals created by the simulation of the SUT. However, when executing closed-loop tests, further changes to the generated harness are required to enable the evaluation and overriding of arbitrary internal signals. Thus, when starting the test execution, the generated harness is enriched, as described below, depending on the executed tests and correspondent use of *override* actions, internal signals to be evaluated and *when* actions.

Overriding internal signals is realized by adding an *override* block from the Arttest block library to the harness for every signal which might be overridden according to the specification. Figure 4 shows an internal signal of a model on the left side, which is assumed to be overridden by a test. On the right side of the figure, the result of the automated enrichment of the harness, performed before test execution, is shown. The enriched *override* block allows Arttest to switch between the simulated signal (original signal) and a specified signal according to the executed test.

Evaluating internal signals requires a similar harness enrichment as for overriding signals. Internal signals which are only evaluated and thus not modified need to be recorded, such that Arttest may retrieve the signal data after simulation and perform an evaluation to decide whether the test passed or failed. To record the signal, a block from the Arttest library is added with the signal to be recorded being the input to the added block.

***When* actions** require harness enrichments to access signals used in conditions. The enrichment is identical to the enrichment for the evaluation of internal signals. Furthermore, the use of *when* actions within a test may change the length of the test, whether signals are overridden or not and which values are used to override signals, depending on the condition evaluating to $true$ or $false$ and the time when the condition evaluates to $true$. Thus, an Arttest block is added to the harness, which evaluates the conditions of *when* actions during test execution and decides whether the conditions evaluate to $true$ within the specified time. In case the condition evaluates to $true$ within the specified time frame, the simulation is paused, the state of the model is communicated to Arttest and new signal values based on the retrieved information are calculated and fed to the harness. Finally, before continuing the simulation, the updated values for the internal signals to be overridden, if there are any, are updated within the Arttest blocks which were added during harness enrichment. Figure 5 illustrates this process.

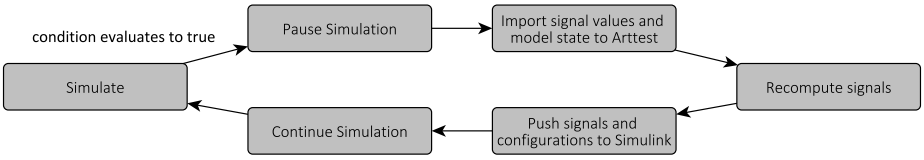

Fig. 5: Arttest process to execute tests containing *when* actions

When enriching the harness model with Arttest library blocks, properties of the model and signals may not be changed. For instance, changing sampling times of blocks and data types of signals would influence the model's semantics in unintended ways. The Arttest library blocks are configured such that the model and signal properties remain unchanged. Furthermore, the Arttest library is designed to be compatible with the Simulink *ert* system target file for code generation. Consequently, code can be generated and compiled based on the enriched harness to perform closed-loop SIL tests. The generation and compilation of code for the enriched SUT is automated by Arttest, enabling the reuse of implemented tests on software level without the need to adapt existing MIL tests manually.

4 Example

To show how the presented concepts for closed-loop MIL and SIL tests work, we test a window control system with a plant model as shown in Figure 6. In order to keep the example as simple as possible, we will assume that a certain driver behavior is hard-coded into the environment model.

The window control model has six input signals being described in Table 5 with driver commands having priority over passenger commands and up commands overriding down commands.

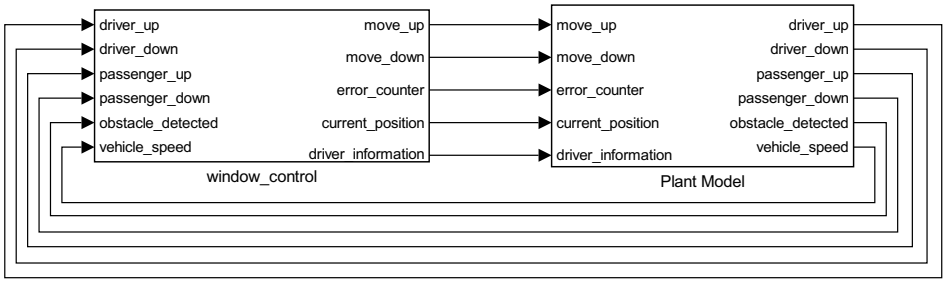

Fig. 6: Window control and plant model

Input Signal	Description
driver_up	*true* if the driver gives the command to close the window
driver_down	*true* if the driver gives the command to open the window
passenger_up	similar to driver_up with the passenger giving the command
passenger_down	similar to driver_down with the passenger giving the command
obstacle_detected	*true* if an obstacle is detected preventing the window from being closed
vehicle_speed	speed of the vehicle in km/h

Tab. 5: Description of the input signals to the window controller

The output signal move_up is $true$ in case the window is in the process of being closed. The signal move_down indicates similarly that the window is being opened. In case an obstacle is detected and commands indicate that the window shall close, the window is not lifted any further and the error counter increases. To save energy while driving fast, an information is shown, indicated by boolean signal driver_information, to the driver in case the vehicle speed exceeds 35 km/h and the window is open. The decision to save energy by closing the window is left to the driver.

The plant model describes the scenario that the driver enters a vehicle, the window being open. The driver needs 1.5 seconds to start the engine and subsequently accelerate the vehicle to 50 km/h. When the driver information is shown, it takes the driver one second to notice the information and press the correspondent button until the window is closed.

When simulating the closed-loop system shown in Figure 6 and monitoring the input and output signals of the window_control subsystem, the described behavior as shown in Figure 7 can be observed. Comparing the signals generated by the closed-loop model, we notice that vehicle_speed reaches the threshold of 35km/h after 19 seconds and driver_information switches from zero ($false$) to one ($true$). At second 20, the driver_up signal switches from zero to one and consequently, the window starts to close, which is indicated by the linear rise of signal current_position and move_up being $true$.

Using Arttest and the presented signal specification language, we intend to test if the window_control model closes the window as soon as the driver gives the command.

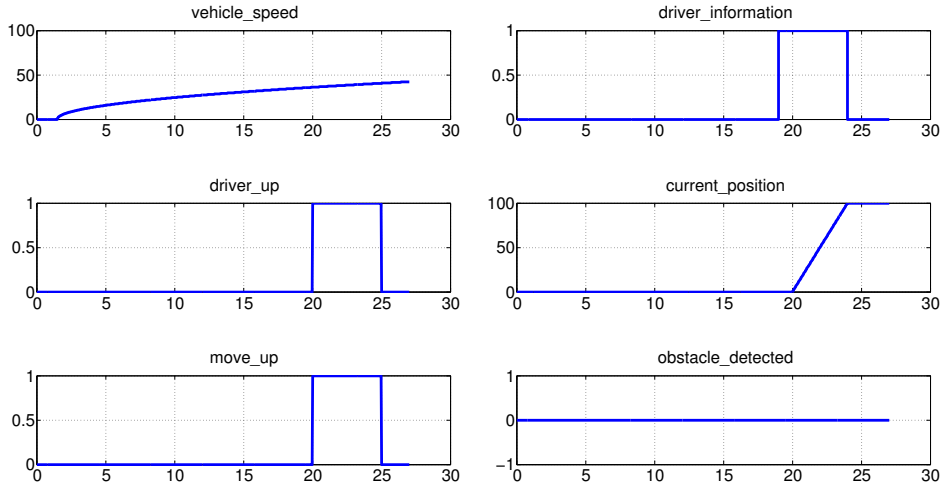

Fig. 7: Signals generated during execution of the model in Figure 6

Moreover, the test shall override the obstacle_detected signal of the plant model such that an obstacle is detected while the window is being closed and ensure that the window is not further lifted. Listing 3 shows the correspondent test based on the signal specification language presented in Section 2.

```
Test:
Step 1:
when [current_position >= 50] during {40} seconds then (
        override <obstacle_detected> on
        <obstacle_detected> set to [TRUE]
)
wait for {5} seconds

Acceptance:
Criterion 1:
when [driver_up == TRUE] during {40} seconds then   (
        <current_position> ramps to [50] within {2} seconds
)
```

List. 3: Test implementation with overridden internal signal and dynamically adapted reference signal current_position

We execute the test from Listing 3 using Arttest and MATLAB R2014a on model and software level. The signals logged during execution of the MIL and SIL test are identical and shown in Figure 8. Since the test does not influence the vehicle speed, the signal vehicle_speed remains unchanged compared to the signal from Figure 7. This holds for the other signals until second 22, too.

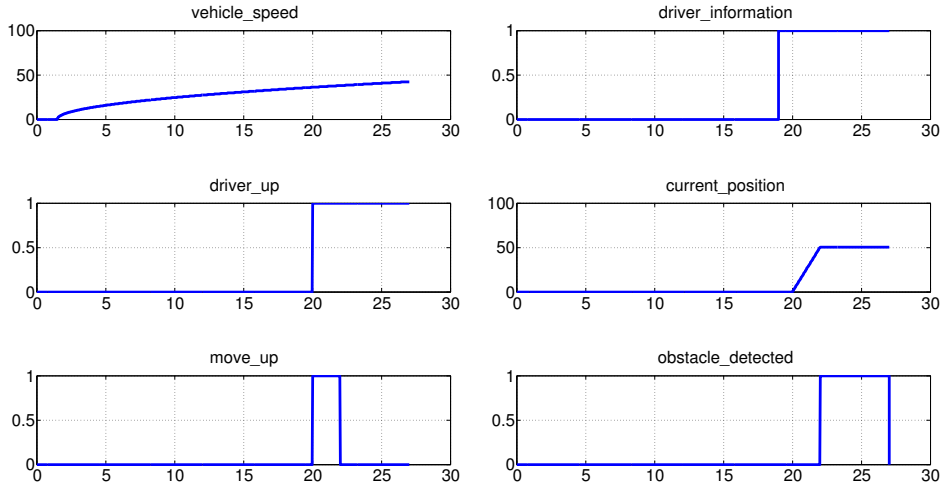

Fig. 8: Signals generated during execution of the test from Listing 3 for the model in Figure 6

According to the executed test case, when current_position reaches 50%, the signal obstacle_detected of the plant model is overridden and its value switches from $false$ to $true$. The window_control model reacts to the detected obstacle causing the window to be halted and thus move_up to be $false$ and current_position to hold the current value as long as the obstacle remains detected.

For evaluation purpose, the logged values for current_position shown in Figure 8 are compared to the specified reference signal. Since the condition $driver\_up == TRUE$ was satisfied during test execution at second 20, the specified reference signal is zero until second 20 and then ramps to the value 50 within two seconds. Since the reference signal matches the simulated signal, the test passes.

5 Related Work

There are many approaches from industry and academia available addressing functional test specification and automation on different test levels[La04]. For instance, MTest[3], developed by Model Engineering Solutions, supports the specification of functional tests using a domain specific language. MTest is integrated into MATLAB and supports MIL, SIL and Processor-in-the-Loop (PIL) test executions such as regression/back-to-back tests and coverage analyses.

Simulink Test[4], developed by The Mathworks, is natively integrated into Simulink providing

[3] http://www.model-engineers.com/de/mtest.html
[4] https://www.mathworks.com/products/simulink-test.html

support for MIL, SIL, PIL, HIL and back-to-back tests. Tests are created using specific Simulink blocks provided by Simulink Test, which additionally automates tasks such as test harness generation.

Another commercial solution for functional testing of model-based developed systems is TPT[5], a tool developed by Piketec[BK08]. TPT supports the test execution of MIL, SIL, PIL and HIL tests. Tests can be either specified graphically using automatons, by importing measured data or by test generation techniques.

ECU-Test[6] is a software developed by tracetronic to create and execute tests by interaction with various software and hardware test environments from different vendors[Re03]. ECU-Test features MIL and SIL tests besides HIL test execution.

With focus on HIL tests, dSpace offers solutions such as Automation Desk[7], a software solution for test implementation and automation[Na04]. Tests and evaluation criteria are specified using a graphical specification.

Another approach, targeting the automation of tests for model-based systems is SIMOTEST, developed at Fraunhofer-IESE[BE11]. SIMOTEST supports the IEEE 1641[IE10] standard for signal and test specification enabling signals to be described as a composition of frequently used signal patterns. Similar to MTest, SIMOTEST is integrated into MATLAB and automates test executions and evaluations.

Besides the IEEE 1641 standard, there are other approaches to describe signals, such as the Testing and Test Control Notation (TTCN-3), a well-defined formal language developed to describe tests[Gr03]. Apart from a textual and a table based description, the third version of TTCN introduces a graphical format (GFT) to describe test sequences using a similar notation to UML sequence charts. With Continuous TTCN3[SBG06], extensions to TTCN3 are presented allowing test specifications for reactive embedded systems.

6 Conclusion

This paper presents a test specification language for closed-loop tests, able to express dynamic reactions to events occurring during test execution and influence simulations by overriding signals, e.g., for fault injection. After introducing formal syntax and semantics of the language, Arttest, a tool supporting the automated execution of the specified tests with MATLAB/Simulink, is presented. Using a closed-loop example, we show how the language and the concepts can be used to create a test implementation. The example comprises the overriding of an internal signal and is executed as a MIL and a SIL test with Arttest.

[5] http://www.piketec.com/en/2/tpt.html
[6] https://www.tracetronic.com/products/ecu-test
[7] https://www.dspace.com/en/inc/home/products/sw/test_automation_software/automdesk.cfm

References

[BE11] Böhr, Frank; Eschbach, Robert: SIMOTEST: A tool for automated testing of hybrid real-time Simulink models. In: Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on. IEEE, pp. 1–4, 2011.

[BK08] Bringmann, Eckard; Krämer, Andreas: Model-based testing of automotive systems. In: Software Testing, Verification, and Validation, 2008 1st International Conference on. IEEE, pp. 485–493, 2008.

[Br07] Broy, Manfred; Kruger, Ingolf H; Pretschner, Alexander; Salzmann, Christian: Engineering automotive software. Proceedings of the IEEE, 95(2):356–373, 2007.

[Gr03] Grabowski, Jens; Hogrefe, Dieter; Réthy, György; Schieferdecker, Ina; Wiles, Anthony; Willcock, Colin: An introduction to the testing and test control notation (TTCN-3). Computer Networks, 42(3):375–403, 2003.

[IE10] IEEE 1641: Signal and Test Definition. 2010.

[IS11] ISO 26262: Road vehicles - Functional safety - Part 6: Product development at the software level. 2011.

[Jo12] Jones, Capers: Software defect origins and removal methods. Namcook Analytics, Tech. Rep., 2012.

[La04] Lamberg, Klaus; Beine, Michael; Eschmann, Mario; Otterbach, Rainer; Conrad, Mirko; Fey, Ines: Model-based Testing of Embedded Automotive Software Using Mtest. In: SAE Technical Paper. SAE International, 2004.

[Na04] Nabi, Syed; Balike, Mahesh; Allen, Jace; Rzemien, Kevin: An overview of hardware-in-the-loop testing systems at Visteon. Technical report, SAE Technical Paper, 2004.

[Re03] Reuss et al., BMW: Automatisierter Motorsteuergerätetest mit Hardware-in-the-Loop Prüfständen. 2003.

[SBG06] Schieferdecker, Ina; Bringmann, Eckard; Grossmann, Jürgen: Continuous TTCN-3: Testing of Embedded Control Systems. In: Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems. SEAS, ACM, NY, USA, pp. 29–36, 2006.

[Sc96] Scowen, Roger: International standard(ISO 14977) Extended BNF. 1996.

[Wi17] Wiechowski, Norbert; Rambow, Thomas; Busch, Rainer; Kugler, Alexander; Hansen, Norman; Kowalewski, Stefan: Arttest - a New Test Environment for Model-Based Software Development. In: SAE Technical Paper. SAE International, 2017.