Abstract—Software integration tests for embedded systems must cater for the physical process with which the systems interact and can include user input. This can make testing very time-consuming because test engineers often manually execute test specifications with many thousand lines of instructions. Furthermore, such manual tests are often imprecise because human operators cannot execute interactions at a granularity of a few milliseconds.

This article presents the CAST (Computer-Aided Specification and Testing) approach to automating the testing of embedded systems, which consists of three parts: a domain-specific language, which allows test engineers to specify test cases formally with a familiar syntax; an execution engine, which allows them to run tests either automatically or interactively; and an interface, which connects the execution engine to the embedded system. We validate the proposed approach by deploying it to a product testing environment and show that our solution provides several advantages such as significantly reduced testing times and more concise test specifications.

Keywords—test automation; embedded system; domain-specific testing; embedded software testing

I. INTRODUCTION

Both research and practical experience in the field of software engineering have yielded sophisticated methods, processes, and tools for software testing. Thorough testing of today’s software employs unit tests, integration tests, model-based testing, test coverage analysis, and many other well-proven techniques.

Some of these techniques are however difficult to apply to embedded systems because of the special nature of these systems. Whereas testing desktop or server software allows test engineers to execute the tests in the same ecosystem in which the software under test is executed, the test environment and the software tested reside in different ecosystems in the embedded world [1]. Embedded systems usually provide very small supplies of memory and CPU power. Furthermore, they provide interfaces to the physical world, such as digital or analog I/O, whereas non-embedded software often relies on API only. In addition, many functions of embedded systems are not tested by software engineers, but by experts in the respective domain of the system, e.g., plant automation.

Integration testing with the actual hardware in the loop still involves numerous manual tasks such as setting input values, waiting for timeouts, reading results from the display, and pressing buttons. In fact, only a small part of such tests are automated in practice [2]. As a result, testing embedded systems is often time-consuming, which increases cost and time-to-market while it lacks the time precision that an automated test can offer.

A. Motivation

Our research work was triggered by the needs of one of the ABB development teams, which builds and maintains a software library for an embedded real-time controller used in automating processes, e.g., in production plants. The software library comes with an extensive test case specification, which contains step-by-step instructions for test engineers to reproduce both desired and undesired scenarios. This specification consists of around 30 documents written in English with each document comprising up to 300 pages.

These drawbacks constituted a strong motivation to automate the testing process. We identified two major challenges though. First, the test case specification is written in English and maintained by domain experts. In an automated test environment, domain experts without programming experience should still be able to maintain the test specification. Second, physical interaction with the embedded system is required during testing. Therefore, we need to identify a
way to bridge the gap between the system on which tests are executed and the embedded system.

B. Outline of the Solution

In order to overcome the drawbacks shown in the previous section we have developed a solution called CAST (Computer-Aided Specification and Testing), which provides a framework for the automatic execution of tests for embedded real-time systems. This framework is illustrated in Figure 1 and covers many tasks in test automation that were identified in [2, pages 210+]: for specification and test management, our framework provides the domain-specific language (DSL) TESLA, which will be introduced in Section II. Using TESLA, test cases can be precisely described. Furthermore, large test case specifications can be organized in several test suites, which can be run whenever necessary, as many times as needed, all at once, or independently. This shifts the focus of the test engineer away from a tedious repetition of steps and points her to the critical tasks for which human intervention is necessary. TESLA further supports test data generation with built-in iterators and random data generators.

The Test Execution Engine, which is covered in Section III, is a machine interpreting the tests and acts as record and playback tool. This engine is connected to the embedded system under test via a Device Interface, which gives access to all the variables and lets the system interact with the device (start, stop, code download, etc.). This allows developers to adapt our testing framework to different embedded systems. The device interface is introduced in Section IV and supports defect management: if the tests are all executed correctly, the test engineer is notified at the end of the execution, while in case of failures the engineer can be notified during the execution (for critical failures) or at the end with a detailed report.

We have achieved significant improvements in terms of time required for test execution as well as timing precision of tests for the given real-time controller that motivated our work. We present these empirical results in Section V. In Section VI, we investigate related work and compare our approach with it. We draw conclusions in Section VII and discuss a generalization of our approach to arbitrary embedded real-time systems.

II. TEST SPECIFICATION AND MANAGEMENT

Test case specifications are often written in natural language and captured in a human-readable way, e.g., in Word files (describing test steps) or Excel sheets (describing the overall settings and parameters). Since natural language is ambiguous and typically lacks structure, such representation is not suited for automated execution. Furthermore, the lack of precise semantics may lead to misunderstandings between domain experts themselves. As a result, there is a strong need for formal specification of tests to disambiguate their meaning, shorten the test specification, and enable automatic execution.

Our goal was to address these issues, while creating a test representation that maps as closely as possible to natural language test specifications. At the same time, acceptance by test engineers was also required. Since domain-specific languages [3] address all these requirements, we decided to use this abstraction for specifying the test cases. We created the domain specific language TESLA (Test Specification Language) for formalizing test specifications.

A. An Overview of TESLA

TESLA is a language that enables specifying test cases for testing of embedded systems. Once the test specification is written in TESLA, it can be executed as many times as needed by the test execution engine. The tests specified in TESLA are run automatically to their completion and usually no supervision is needed. However, TESLA supports also supervised test execution, as tests can contain constructs that involve tester interaction. In particular, TESLA has the possibility to include interactive prompts and manual checks that require tester attention into automated tests.

TESLA is a domain specific language that uses concepts from general programming languages (e.g., namespaces and imports) and adapts them to the domain of embedded systems. Additionally, TESLA introduces test-specific constructs (e.g., checks of device variables) and device-specific constructs to interact with the embedded system (e.g., download a library). First, we will discuss the language’s primitive blocks (actions and checks) from which test cases are constructed, and then the overall test package structure.

B. Primitive Test Blocks

Test cases in TESLA are constructed from two basic blocks: Actions and Checks. Action blocks contain statements that are used to set the embedded system into some desired state. These are then typically followed by check blocks, which verify that the transition to the state and the resulting state were correct. The statements in action
blocks are executed synchronously, one by one, and in the order specified. Action blocks can additionally contain a description, which will become part of the test report. The action statements supported by TESLA can be seen in Table I.

After the execution of an action the tester will typically check its effect on the state of the embedded system. TESLA offers check blocks for this purpose. A check block is a set of expressions that should hold after an action was performed. A check expression will typically contain some device variables and will check for their values or quality, or will contain events and alarms and will test for their state and occurrence. Check blocks are executed asynchronously, with respect to the last action, and their order does not matter. The execution of checks results in failed or successful report entries, which are also highlighted in the TESLA editor for convenience.

Usually a test case runs until its completion and all action and check blocks are executed. This behavior can be altered through by declaring check blocks or tests as critical. In this case, failed checks will result in the whole test being immediately aborted.

Check blocks can have different timing requirements, as seen in Table II. To this end, every check specifies an interval $[t_{\text{start}}, t_{\text{end}}]$ during which a given predicate has to hold. In TESLA, these intervals are specified relative to the action time $t_{\text{action}}$, but in the following we assume that intervals are absolute. Also, some interval tolerance can be specified through the TESLA editor (e.g., some events that arrive one cycle later can be still considered correct). Time is discrete; one time step corresponds to one control cycle of the embedded system.

In the following, the details of the individual timing requirements are explained.

**In-interval check:** This check asserts whether a predicate $P$ holds at least once during an interval, i.e.,

$$\exists t \in [t_{\text{start}}, t_{\text{end}}] \ P(t)$$ (1)

If the predicate $P$ combines several variables, they all have to be "read" at the same time $t$. This check is quite weak since $P$ could have been true all the time or become true multiple times. The smallest time $t$ for which (1) holds is called witness of success and is recorded with the result of the check.

**Never-in-interval check:** This check asserts whether a predicate $P$ never holds during an interval, i.e.,

$$\forall t \in [t_{\text{start}}, t_{\text{end}}] \ \neg P(t)$$ (2)

This type of check is much stronger than the previous one. Still, it does not say anything about what should happen outside the checked interval. However, multiple checks can be defined for the same action for that purpose. The smallest time $t$ for which (2) holds is called witness of failure and is recorded with the result of the check.

**Change-to-true-in-interval check:** This check asserts whether a predicate $P$ becomes true during an interval, i.e.,

$$\exists t \in [t_{\text{start}}, t_{\text{end}}] (\forall t' \in [t_{\text{action}}, t - 1] \ \neg P(t')) \land P(t)$$

This check is a combination of the two previous checks. It asserts that the predicate changes its state from false to true during the interval. The smallest time $t$ for which (3) holds is called witness of success and is recorded with the result of the check.

**Duration check:** This check asserts whether a predicate $P$ holds for some duration $d$. The duration must start during the checked interval, i.e.,

$$\exists t \in [t_{\text{start}}, t_{\text{end}}] \ (\forall t' \in [t_{\text{action}}, t - 1] \ \neg P(t')) \land (\forall t'' \in [t, t + d - 1] \ P(t'')) \land \neg P(t + d)$$ (3)

The duration is only measured the first time $P$ becomes true. Also, the predicate must become false within one cycle after the duration. Again, the time $t$ for which (3) holds is recorded as the witness of success.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous wait</td>
<td>Synchronous wait for some duration.</td>
</tr>
<tr>
<td>Device variable</td>
<td>Synchronous write to a variable in a module on</td>
</tr>
<tr>
<td>write</td>
<td>the device.</td>
</tr>
<tr>
<td>Alarm acknowledgment</td>
<td>Acknowledge all alarms and events on the device.</td>
</tr>
<tr>
<td>Change device state</td>
<td>Set embedded system to online/offline state.</td>
</tr>
<tr>
<td>Download</td>
<td>Compile tested library, download it, and start</td>
</tr>
<tr>
<td></td>
<td>it on the embedded system.</td>
</tr>
<tr>
<td>Interactive prompt</td>
<td>Interactively prompt for supervised execution.</td>
</tr>
<tr>
<td></td>
<td>Requires some manual tester action.</td>
</tr>
</tbody>
</table>

Table I

**ACTION BLOCK COMMANDS IN TESLA**

<table>
<thead>
<tr>
<th>Check type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-interval check</td>
<td>Verify whether a given predicate is true at least once during some interval. As an example, this can be used to check whether an alarm is active.</td>
</tr>
<tr>
<td>Never-in-interval check</td>
<td>Verify whether a given predicate remains false during some interval. As an example, this can be used to check that no alarm was raised.</td>
</tr>
<tr>
<td>Change-to-true-in-interval check</td>
<td>Verify that a given predicate is initially false, but becomes true during the specified interval. Can be used to test state transition of the device, e.g., that an alarm occurred in the given interval, and not elsewhere.</td>
</tr>
<tr>
<td>Duration check</td>
<td>Verify that a given predicate becomes true in some interval and lasts for some duration before it returns back to false. Can be used to test time-based state transitions.</td>
</tr>
</tbody>
</table>

Table II

**POSSIBLE TIMING REQUIREMENTS OF CHECK BLOCKS IN TESLA**
C. Test Case Management

Thorough test case management is important to avoid redundant or even inconsistent tests and reduce the time needed for maintaining the test specification. TESLA offers a structured approach to test case management, which is illustrated in Figure 2. Each source file in the TESLA language is associated to a package and can contain test suites, test cases, modules, or test setups.

![Figure 2. The high-level structure of test specifications written in TESLA.](image)

**Test cases:** Action blocks and check blocks form test cases, which represent one single test scenario. To combat repetition of similar tests, test cases can be parametrized with structures, variables or events. Tests are independent units and are parametrized and run from a test suite.

When a test is run, important errors (such as failed checks) are directly highlighted in the TESLA editor and a test report in different formats is generated. Running a test also creates a detailed log of events and a trace of all variable value changes on the device, which can assist in debugging the device.

**Modules:** Software running on embedded systems typically consists of multiple modules - basic blocks specifying some device behavior. Each module has some input and output variables and can raise alarms and events. TESLA supports the concept of modules as a way of declaring variables, structures, alarms, and events on the device. Each module in TESLA has an associated path, which will be used to locate the module on the embedded system (e.g., through OPC [4]). By combining the module path with the name of a variable, alarm, or event, TESLA can locate and access these on the embedded system.

To declare variables in the module, one has to provide their names and types. TESLA uses type checking to exclude runtime problems caused by mismatching types, e.g., when assigning an integer value to a boolean variable. Alarms and events are declared with their source objects and conditions or messages, respectively. Additionally, to mirror the structure of variables on the embedded system, TESLA introduces a concept of structures, i.e., groupings of variables that correspond to types in the library tested.

Modules can be automatically generated from the embedded system (e.g., by browsing the OPC variable tree) and stored as a reusable library for the tester. This library can then be imported in different packages and used for auto-completion of variable, alarm or event names in the TESLA editor. This way, also misspelling of variables is detected because only artifacts declared in modules can be used in the test cases.

**Setup:** Apart from test cases and modules, TESLA introduces setups. The role of setups is to bring the system to a well-defined initial state before it is tested. Setups can therefore contain only action blocks, which specify the sequence of steps to initialize the device (e.g., go offline, set some offline variables, download, set some online variables, or acknowledge alarms). Setups can extend each other, in which case the parent setup will be run before the child. The connection between a setup and a corresponding test case is specified when running a test in a test suite.

During manual testing, setups were created using settings specified in Excel sheets and by using an application that read and applied them. TESLA editor integrates a tool that can convert these Excel sheets to TESLA setup blocks using minimal user effort, resulting in a setup library that can be reused for different test cases.

**Test suite:** Test suites are entry points for test case execution. In each test suite it is specified which test cases have to be run together with the corresponding setups. Since each test performs its own hierarchy of setups, tests are independent from each other. The inputs to the parametrized test cases are also instantiated in the test suite. This allows test engineers to execute the same test multiple times, but with different values for some fields. This feature combined with loops and random number generation makes it very easy to reuse tests, execute a relevant subset of all test cases, or to specify a desired execution order.

D. Other TESLA Concepts

Besides the previously introduced features, TESLA offers a few more benefits that make life of an embedded system tester easier.

Some tests defy automation because they involve physical manipulation of the embedded system such as disconnecting and reconnecting the power supply. For such test cases, TESLA supports supervised execution of tests, which is made available through prompts and guided checks. These are interactive dialog boxes in the GUI that require a given activity from the tester. A prompt is used to specify an action that has to be performed to set the device into some state (e.g., restart a controller manually). Guided checks are interactive GUI elements that require some feedback from the tester, usually expressing a need to verify a state of the device that cannot be determined through some API. Using TESLA, engineers can arbitrarily combine supervised and fully automatic tests.

TESLA provides expressions similar to general programming languages. However, it adds several new features.
TESLA supports random variables, which have different values each time they are used. Range types, list types, and structures are also supported in expressions with corresponding operators (e.g., “in” for lists). TESLA expressions also contain alarm and event specific operators to test for their state, occurrence, and attributes (e.g., severity).

Parametrized test cases in TESLA are also noteworthy because parameters can be structures, device variables, or primitive values. TESLA automatically decides whether to pass parameter by value or by reference such that the passed expressions or variables are read or written as late as possible in the place where they are used.

TESLA also provides for loops. Apart from simple for loops (iterating over single list or range) as found in most programming languages, TESLA also has the capability to iterate over many lists or ranges at once, synchronously. As a consequence, iterating over two lists is the same as iterating over a list containing pairs of values.

The TESLA editor performs various static checks and also a number of dynamic checks, which ensure that the test cases behave correctly. Among all static checks, TESLA checks for the existence of used variables on the embedded system, for infinite recursions, loops or type hierarchies, or for correctness of write actions to device variables. The goal of all these checks is to minimize mistakes when specifying test cases.

E. Example TESLA Code

We demonstrate an example of TESLA code on a simple room conditioning system. Assume a room controller with a temperature sensor and two connected ACs. We would like to check that the system behaves well on a temperature above 25 degrees with one AC turned on, i.e., we want to test that no power overload happens on the AC.

Each file in TESLA has to define to which package it belongs. In the Definitions package we define two modules: the module Types contains the structure ACStatus, a representation of an AC. The module Room declares instances of ACStatus types, as well as a variable representing temperature sensor measurements and an alarm that should occur on power overload.

In the second file we import the Definitions package such that we can reference it when needed. The setup specifies the initial system state before each test - the system is running on the embedded system and both ACs are turned off. Next we define a test case, which turns on an AC, sets the room temperature and checks if no power overload happens on the AC. The check here is defined as critical, meaning that if it fails, the test case will finish abnormally.

F. Implementation Details

We used the Eclipse IDE and the Xtext\(^1\) plug-in to create an efficient and appealing development and execution engine for TESLA. This environment comprises several tools: an editor, which comes with user-friendly features such as syntax highlighting, outline view, content proposals, and validation; a tool to browse variables on the OPC server; and a test execution engine with an interpreter and a debugger, which will be introduced in the subsequent section.

\(^1\)www.xtext.org
III. TEST EXECUTION ENGINE

In order to execute tests written in TESLA, either an interpreter or a code generator is required. Xtext includes the template language Xtend2 and is therefore slightly biased towards the code generation approach. However, since Xtend2 had just been released, its editor and compiler had several bugs, which made it difficult for us to use it beyond smaller examples. Furthermore, although Xtend2 seems a promising approach, we suspect that we would have to fall back to Java for the more complex parts of the code generation.

Also, code generation adds an additional indirection because the generated intermediate code needs to be compiled first. This problem can be overcome by writing an Eclipse extension that compiles the DSL automatically when the file is saved.

The biggest difference between interpretation and compilation, however, is that an interpreter can run in the same Java virtual machine as the IDE while the generated code will always run in a new process. This means that it is much simpler for an interpreter to interact with the UI of the IDE. As an example, we can access the user’s plugin settings, highlight parts of the DSL, or provide stepped execution of test cases.

Since a tight UI integration was very important to our users, we have chosen to write an interpreter for TESLA. As a compromise between interpretation and compilation we extracted all the UI-independent parts of the interpreter into the Java Testing Library (cf. Figure 3). The idea is that this library could be reused for code generation or even for an embedded DSL using Java or Scala.

The interpreter is given an abstract syntax tree (AST) that can be processed using the API of the Eclipse Modeling Framework (EMF). Xtext takes care of parsing, syntax validation, and reference resolution based on our rules for TESLA. The task of interpretation then consists of giving a meaning to each element in the AST.

Our interpreter is written in Scala to leverage the power of pattern matching and functional programming. The heart of the interpreter is a recursive function and one big match expression over all AST nodes. For each interesting node we define its behavior using one case statement. A node either computes a value, performs a remote action, or calls the interpretation function recursively on its child nodes.

Let us now consider how TESLA programs are executed. The entry point of a TESLA program is a test suite. The tests in a test suite are executed sequentially. The execution of a test consists of two phases: create OPC subscriptions and perform actions & checks.

A. Phase 1: Create OPC Subscriptions

A key idea in our execution engine is that all read operations are asynchronous. This allows us to read all the variables of interest after every cycle and check for arbitrary patterns of variable changes. Synchronous communication is impracticable in this case and would quickly exhaust the resources of the OPC server. Asynchronous communication is realized using the OPC Advise feature. In this case, the
client first subscribes to a set of variables. Then, the server
will continuously inform the client about every variable
change, event, or alarm to which the client is subscribed.

The goal of the first phase of the execution is to create
the list of subscriptions. This requires a mini-interpreter that
goes through the test and collects all the OPC variables,
alarms, and events appearing in read operations. All other
commands are ignored in this phase.

Once the OPC subscriptions have been made, reading is
completely passive. The test execution engine has a back-
ground thread that listens for server updates and maintains an
event log. The event log contains all the OPC item updates,
OPC events and OPC alarm state changes received so far.
It can be used to obtain the value of any variable, or the
state of any alarm, for any point in time during the test.
This is an important feature of the execution engine and
is used to support check expressions over time intervals
(cf. Subsection II-B). We have also used the event log to
draw timing diagrams, which support the test engineer in
debugging.

It is important to note that every server update has a time
stamp. This allows us to perform accurate testing regardless
of network latency, notification delays or, to some extent,
CPU load. Since time stamps are not easily readable for the
human tester we came up with a simple trick to associate
the hardware cycle number to every update: Using the
configuration software of the embedded system we created
a cycle counter variable and simple program that increments
the counter every cycle. The item updates for the cycle
counter variable are received at regular intervals (e.g., every
250 ms) and can be used to get the current server time.

B. Phase 2: Perform Actions and Checks

A test consists of an alternating sequence of actions and
checks. Actions are synchronous and have a direct effect
on the hardware. After an action is complete we record the
server time in \( t_{action} \) and evaluate the checks for that action.

Checks can only perform reads and do not interact with
the hardware. Thus, checks are evaluated against the history
of the hardware device using the event log described above.

C. Execution Engine Architecture

The detailed architecture of our solution can be seen in
Figure 3. A typical setup consists of an embedded system
under test connected to a machine with an OPC Server and a
Hardware Builder running. The OPC Server is used to mon-
itor and change online variables on the controller, whereas
Hardware Builder will be used to download, compile, and
change offline parameters of a library. To access these tools
CAST uses a Device Interface layer which acts as an OPC
client and provides methods to access the Hardware Builder
conveniently.

Since our preferred OPC toolkit and the Hardware Builder
both provide a C# interface, the Device Interface needed to
be written in C#. On the other side we have the TESLA
runtime which is built on top of Xtext and Eclipse, both of
which are Java based. In addition, the TESLA runtime will
typically not run on the same machine as the C# Device
Interface. To fill the gap between different languages and
machines, the Java Testing Library layer was added. This
layer provides remote access to the C# Device Interface
and consists of a server program on the machine control-
ing the device as well as a client API on the machine
executing TESLA code. The TESLA runtime then uses this
Java Device Interface during interpretation to execute DSL
commands.

IV. DEVICE INTERFACE

The current manual solution for testing software employs
a human operator who is responsible for initializing the
hardware (offline), assigning values to variables (offline
or online), monitoring the changes in these values, and
comparing them to the expected values specified in the
test documentation. CAST replaces this human interface
with a machine interface by using the Hardware Builder
for initializing the hardware and OPC for changing and
monitoring variables when the system is online.

A. OPC

OPC [4] is a series of standards specifying the communi-
cation of real-time plant data between control devices. The
project uses the OPC Data Access (OPC DA) specification
for changing and monitoring variables on a device and
the OPC Alarms and Events specification (OPC AE) for
monitoring and notifying about alarms, operator actions, in-
f ormational messages, and tracking/auditing messages. The
project utilizes a C# library for developing an OPC client
that supports both the required OPC specifications.

Each OPC server is identified by a unique 128-bit string.
This unique string is passed as a parameter to the OPC client,
which then establishes a connection to the OPC server. The
OPC client provides us with the options to initialize OPC DA
and OPC AE objects; read, write and subscribe to a group
of OPC DA variables; get the current state of an OPC AE
object; and acknowledge all the alarms in the system. It is
possible to subscribe to any number of items in a single
subscription. A change in the value of any of these items
triggers a callback and the tester is notified of the changes
that have occurred. These changes are also logged by the
test execution engine. Similarly, it is also possible to read
and write any number of OPC variables using functions
provided by the client. While reads are asynchronous, writes
are synchronous. For this reason, the OPC client maintains
two separate connections to the OPC server - synchronous
and asynchronous. The client also provides callbacks when
the state of an OPC AE object has changed. Again, all
functions related to OPC AE are asynchronous in nature.
The OPC client facilitates the task of the tester. Previously, mundane tasks of the tester were not only repetitive, but also prone to human error. Such tasks include setting the device’s variables, waiting for feedback and then verifying it, reading the value of the variables on completion of the test, checking for alarms, or verifying that the alarm was raised and then clearing it. The client provides a way to automate these tasks. Numerous error checks within the client ensure that the tester is notified of the reason the test failed—whether it was a bad OPC variable name or whether the connection to the OPC server was lost. The client therefore provides us with a robust interface to automate tests.

B. Hardware Builder

The specific embedded system whose testing motivated the work presented in this article comes with a software called Hardware Builder. This software can be utilized to initialize the controller, download a library to the controller, and to take the controller online or offline. The Hardware Builder is also capable of changing the offline parameters of the downloaded library to ensure that the library is set up according to the test specification. The Hardware Builder also provides us with methods to check the values of internal variables in the library (those not accessible via OPC) when the controller is online.

The Hardware Builder therefore complements OPC by filling in the gaps in our device interface and allows for the provision of a complete solution. During manual testing, tasks such as downloading the library had to be done by hand. The tester had to find the directory holding the required library and then load it to the controller. This is now accomplished automatically. Similarly, to initialize the downloaded library, the tester had to open an Excel sheet containing the initial settings and then initialize the library using an Excel plugin. This task is now again automated, thanks to the API of the Hardware Builder. After each test run, the controller had to be taken offline, the library reinitialized, and then the controller had to be taken online again. This cycle was tedious and repetitive and prone to human errors. Using the Hardware Builder, these tasks can now be accomplished automatically and correctly. Furthermore, the Hardware Builder is also written in C# and provides us with COM classes that integrate well with our OPC client, thus providing for a coherent and integrated device interface.

The Hardware Builder, however, is not essential for the proper functioning of our solution. Our solution can be used with any controller that provides an OPC interface. The Hardware Builder is also not tied in to our solution. It can be easily replaced with a similar service that may be provided by another controller. Our solution can thus automate tests for any controller that supports OPC.

V. Validation of the CAST Approach

Since our proposed approach replaces manual testing with automatic testing we validated CAST through a deployment of the system in a production facility. After setting up the Eclipse-based CAST environment, the test engineers specified a representative subset of test cases in the TESLA language and successfully executed them. The remainder of this section presents a summary of the test engineers’ feedback and explains how CAST addresses the drawbacks listed in the introduction of this article.

D 1 - Shorter execution time. With the manual approach, a complete manual test run used to take from 3 to 5 days, depending on the object under test and the test type. Based on the representative subset of test cases that they formalized and executed, our test engineers estimate that a complete test run using our CAST approach takes from 15 to 30 minutes if applied to the same objects and test types. A visual representation of this can be seen in Figure 4.

The CAST execution time does not take into account the manual effort required to translate the specifications from natural language to TESLA. This is because the effort was deemed comparable to the initial effort of writing the specifications in the first place, and the overhead is therefore common to both approaches. Moreover, many original specifications are very repetitive, with only some parameters changing from one test to the next, and can be dealt with much faster by using a for loop in TESLA.

D 2 - Conciseness of test specification. The manual approach used different representations using tables, icons, abbreviations, and natural language. Even for experienced testers some ambiguities were hard to resolve due to omitted prefixes or misspelled names. CAST provides a uniform and well-defined representation to store test specifications. The TESLA editor performs type-checking, detects undeclared variables, and proposes auto-completions. A new TESLA library with all definitions can be regenerated with a few clicks whenever the hardware library changes.
D 3 - Perfect repeatability. Even with the most thorough tester, each manual test run would often yield slightly different results or would be executed slightly differently with respect to the others. An automated CAST test, on the contrary, has always a 100% perfect repeatability in both the structure of the execution and the yielded results, when testing the same system under the same preconditions.

D 4 - Better time accuracy. With a manual approach, time accuracy was low and depended mostly on the reflexes of the engineer doing executing the test, which varies between different testers and between different test runs for the same tester. In practice, test engineers can time actions within intervals between 0.5 s and 1 s. In contrast, CAST achieves a precision of a few milliseconds. Furthermore, the most severe drawback of the manual approach is that engineers cannot track events down to a specific control cycle, which often leaves them clueless about when a specific event logically occurred with respect to other events. This has been solved with CAST, which offers cycle-level accuracy in order to track down specific timing problems.

D 5 - No repetitive work. Interaction of humans with machines should be made more pleasant for the former. This is especially true when considering tedious and repetitive testing procedures. All of the testing engineers involved found the proposed CAST approach and TESLA DSL both intuitive, effective, and efficient. In particular, we are proud of one comment from one of our test engineers: “Creating my first test case was really straightforward and was even some kind of fun... :-) I really wish we had had this tool before we started all our testing activities and we could have written all our test specs in this environment.”

In addition, CAST provides some improvements that were initially not identified as drawbacks of the manual testing approach. Among these improvements we would like to highlight increased test coverage. Whereas manual test specifications ask the test engineer to only check the most important variables after each action is executed (i.e., the ones assumed to change) in order to save time and to avoid overwhelming her with too many data, an automated CAST test can be configured to check much more (or even all) the variables after each step without any negative influence on the testing (machines not being easily bored or losing focus). This enables testing of variables on an input and output range that is larger by at least one order of magnitude and checking the code for robustness in different usage situations, going beyond current requirements and even anticipating future ones.

VI. Related Work

Structured integration testing of embedded systems with the actual hardware in the loop has gained momentum in the past years. Schlager [5] introduces an approach to hardware-in-the-loop (HiL) simulation in which the environment of the embedded system is simulated by providing stimulus for the system’s input and consuming the system’s output. Whereas the integration of the embedded system is covered comprehensively, test case management is not. In contrast, our CAST approach supports test case management with parametrized test cases, which are organized in test suites and test packages.

Several publications discuss testing software in the automotive area. Lamberg et al. [6] introduce a model-based testing approach in which state machines are defined graphically to specify the desired system behavior. The interface to the hardware and how the tests are executed are not covered in this paper. Our experience shows that the intricacies of test execution are among the most difficult tasks of setting up a HiL testing environment. Another model-based testing approach is presented by Bringmann and Krämer [7]. In their approach, the test cases are also specified graphically. They are then compiled and executed on a virtual machine in real-time. A test report is created after the complete test has finished. Unlike in CAST, interactive testing and partial execution of tests are not possible, which is required by some of our test cases.

The language TTCN-3 [8] provides a structured approach to test specification and implementation. Its main limitation is the lack of support for fine-granular time intervals, which is required for testing real-time systems. An extension to TTCN-3 was proposed by Grossmann et al. [9], which adds time intervals to TTCN-3. A testing environment for the real-time extensions of the language is presented by Serbanescu and Schieferdecker [10]. The CAST approach supports different kinds of fine-granular interval checks. Our test specification language TESLA is tightly integrated with the test execution engine.

Lu et al. [11] focus on the real-time aspects of testing embedded systems, in particular, power electronics. In their approach, they use off-the-shelf components and open-source software. A drawback of their approach is that they introduce a tight coupling between the testing environment and the system under test. In contrast, CAST offers an explicit device interface, which has two advantages. First, our test execution engine can work in asynchronous mode whereas the time-critical tasks are done by the OPC server. Second, the device interface can be easily exchanged, which makes CAST more adaptable to other embedded systems.

Hierons et al. [12] present a survey of different formal methods and testing approaches. They suggest a closer integration of the two domains in order to achieve a more rigorous testing procedure. The amount of surveyed work is extensive, and the approach envisioned by the authors would surely bring benefits to the field. However, we know from experience that test engineers are unlikely to adopt formal methods because they require an orthogonal set of skills. Therefore, CAST uses the domain-specific language TESLA, which was accepted by our test engineers because of its user-friendly syntax. Nevertheless, TESLA has formal
syntax and semantics, which makes it a precise means for specifying test cases.

VII. Conclusions

We have presented the CAST approach, which combines a domain-specific language for test case specification and automatic test execution with an embedded device in the loop. This approach allows test engineers to automate a large amount of tests, reducing testing time significantly from several weeks to some hours. This is a big step forward from the state of the art where testing knows little automation [2]. Besides much shorter testing times, the CAST approach also provides more concise test specifications and more accurate test execution.

The approach presented can be easily adapted and is thus applicable to a wide range of testing scenarios. TESLA is an extensible language, which can be adapted to specific test scenarios. In particular, shorthands for frequently occurring action sequences may be added to the language. The OPC interface is an industry standard and supported by numerous embedded systems. Since the individual components of CAST are clearly separated in our architecture (cf. Figure 3), the OPC interface as well as the Hardware Builder interface can be replaced by arbitrary other interfaces to fit specific devices under test.

Note that some test cases defy automation, e.g., tests that require physical interaction with the device such as removing or inserting I/O modules. A few of those test cases could be nevertheless automated (e.g., with the help of external actuators), but they rarely have to be executed and are so specific that extending the TESLA language and execution engine would require major engineering effort. To cover these, we currently support interactive test execution with the presence of a tester.

We have identified several directions for future work. Whereas CAST supports testing with random values, more structured approaches could be integrated such as model-based testing (e.g., [13]) or sophisticated random testing (e.g., [14]). Furthermore, CAST could be supplemented by formal methods [12], e.g., with UPPAAL [15]. Having a formal test specification language would also allow us to combine our testing framework with a specification framework because of the important relation between requirements and testing. In such an approach, requirements could be specified in a formal way and the relations between a set of requirements and their corresponding tests could be captured in a precise and automatically checkable way. Finally, we consider the analysis of test coverage orthogonal to our approach. However, a section about coverage could be added to the test report generated by CAST in the future. In that case, a test coverage tool would need to be integrated in the test execution engine.

References