Model based testing and Hardware-in-the-Loop simulation of embedded CANopen control devices

Mirko Tischer; Dietmar Widmann, Vector Informatik GmbH

CANopen is mainly used in connecting devices in embedded networks. During the development process of the ECU (electronic control unit), the maturity of the embedded system increases with testing in early stages. A good approach is to test before the first prototype is available. Step by step, the test environment is expanded and is able to simulate CANopen bus communication behavior. Application behavior is also added to the test environment. This paper shows the methods available at different development stages and their application. Starting with a simple network simulation that just integrates EDS files, a test environment grows as the development advances. Application behavior is added by hand-written C modules, libraries or even MATLAB/Simulink models. Finally, the simulated ECU and finished ECU use the same application code. To obtain a complete Hardware-in-the-Loop simulator, additional hardware is required to simulate sensors and actors. Testing is possible in parallel to every development step. Test procedures may range from simple interactive elements to a fully automated test system.

Introduction

Today, ECUs exhibit a high software content and place stringent requirements on the tests to be performed (large share of software functionality contrasts with stable complexity in the HW environment, which results in more testing effort in functional tests). Increasingly shorter development cycles practically force product manufacturers to reduce testing effort to a reasonable level, but without unacceptable level of risk to test quality. Verification and validation are playing an increasingly larger role in the overall development process.

Given this situation, it appears necessary not only to proceed more efficiently in these areas, but to significantly shift testing forward in time in development phases. The advantages are obvious: deviations in the functionality are detected early on and can therefore also be corrected earlier. With consistent implementation, this leads to more stable ECU implementations and consequently also to a higher maturity level of the end product.

How does model-based testing contribute to better product quality?

Model-based testing is understood as a technology in which testcases can be derived from a model. After their execution, these testcases can also be automatically evaluated and documented. In all cases, the user must work with a model and follow a systematic and explicit working method in testing.

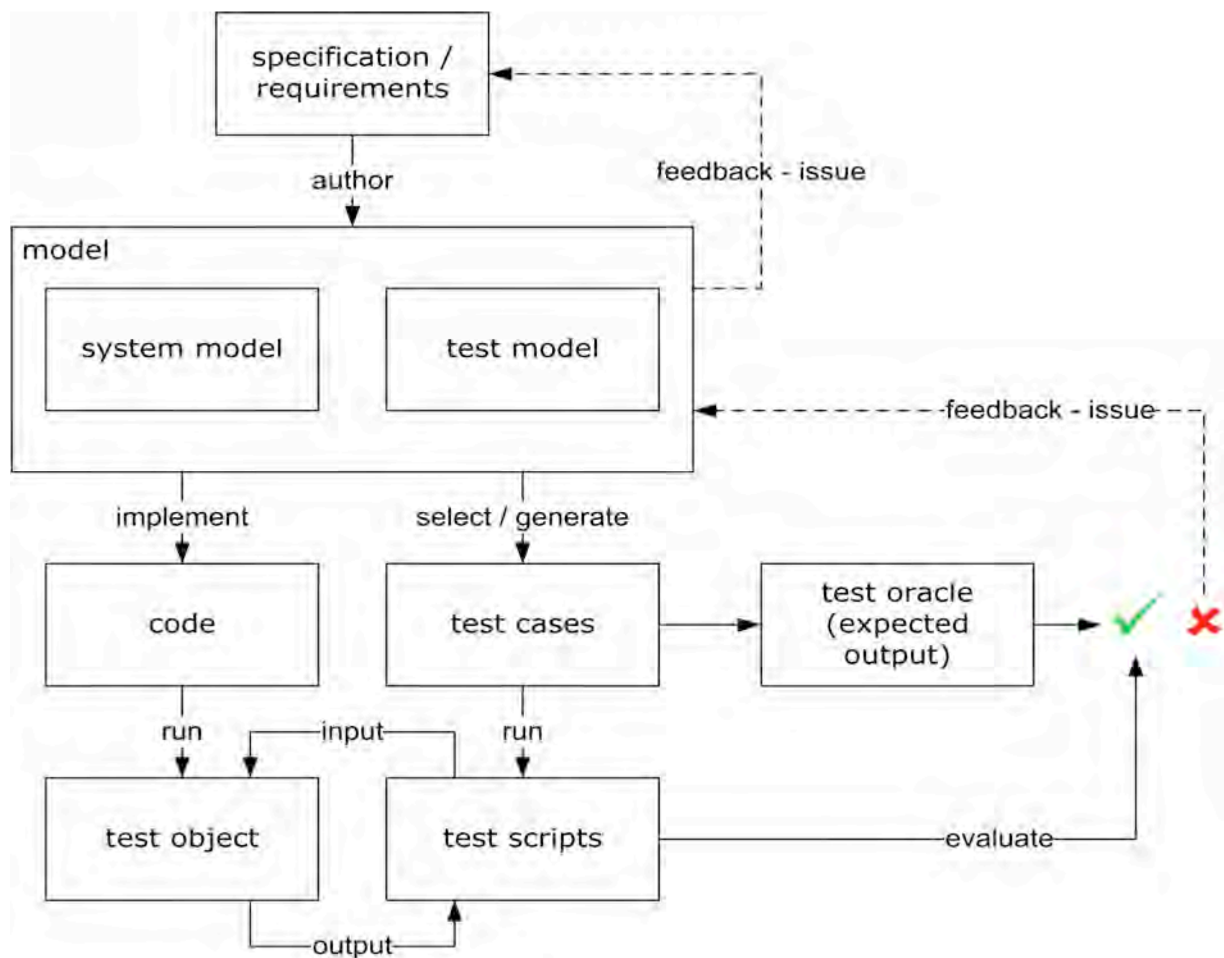


Figure 1: Overview – model-based testing

For model-based testing (MBT), we need at least the following:

- Requirements / specification – Requirements describe the system and are absolutely necessary to construct a model.
- A model based on these requirements – The actual creative step in this process is implementation of requirements in a model, which requires enormous effort. In the further course of development, the actual software for the ECU is derived from a system model (generated or – in part – manually implemented). Actually, the necessary test sequences can be derived from the system model. However, to permit testing of the system model as well, a separate test model can also be developed from which test sequences are then derived.
- Testcases and test oracle – The testcases and the related tests scripts can then be generated, for the most part, from the model. In this process, the test oracle establishes the expected test result.
- Test execution environment – To enable execution of the tests scripts, a runtime environment must be provided. Naturally, this also includes a simulation of the CANopen ECU's environment.
- This simulation may sometimes assume great complexity in simulating the relevant hardware (sensors and actuators), physical environment conditions and possibly even communication with other ECUs.

Just providing these components requires a substantial amount of effort. For these efforts, the user is rewarded with shorter test execution times (due to automated execution), an environment model (as part of the test execution environment) and the ability to flexibly react to different testing focal points.

Where does the model for a CANopen ECU come from?

It is easy to represent a CANopen ECU in a model using the components shown in Figure 2 <CANopen device model>. Unfortunately, the meaningfulness of this model is still very low, because it lacks the details needed to describe at least minimal application functionality. A model has the task of describing the behavior of the modeled system and is always kept simpler than the modeled system – but not too simple.

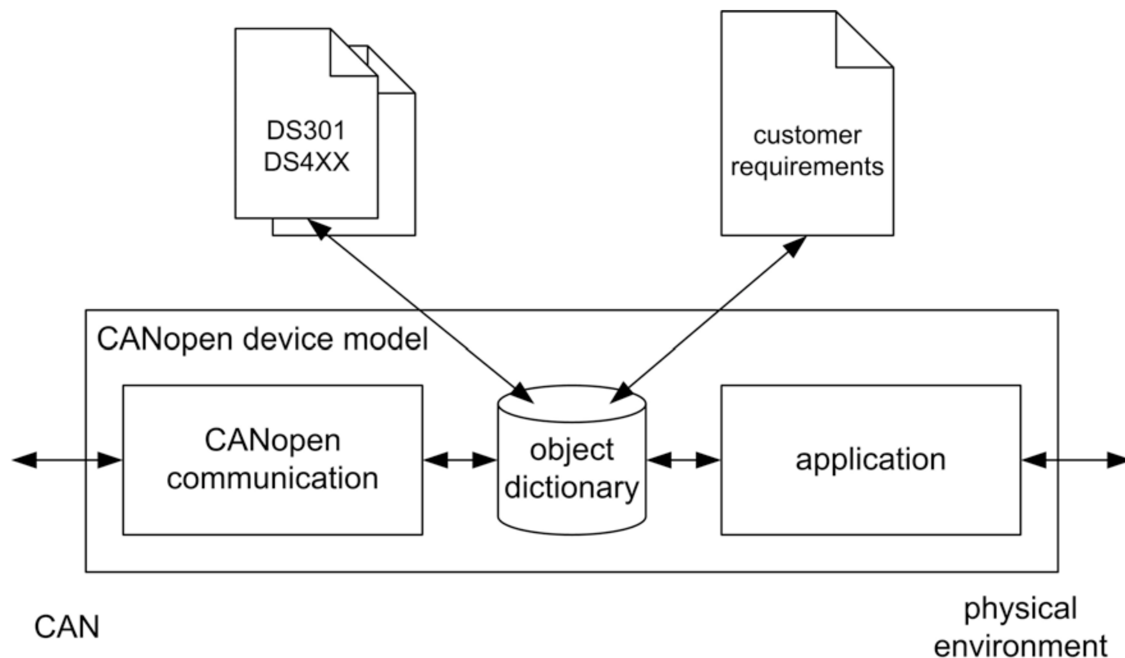


Figure 2: Device model – CANopen

The left part of the figure (communication over CANopen and the object dictionary) is already modeled to a great extent by the underlying specifications (requirements specification and model are simultaneously covered by the DS301 or DS4xx specification here). Unfortunately, this modeling is insufficient to execute a complete generation of a CANopen layer. Formal descriptions can certainly also be found in specifications by CiA, e.g. via finite state machines (FSM) and message sequence charts. However, they do not describe the comprehensive interrelationships between all individual aspects, and so these interrelationships are also lacking in a generation. It is especially difficult to form the model for the communication portion using the many optional components of CANopen. It is often unclear when an optional property should or should not be used.

The question is whether it is worthwhile to create a detailed model of the communication portion of a CANopen ECU that enables full generation of ECU code. This is doubtful, because the specification, as noted, permits too many formal gaps. A code generation from the model (here: system model) would therefore be associated with much effort, and in most cases it is not economically justifiable. It would be different if the requirements for functional safety of the CANopen ECU would require

the use of formal methods. In general, however, CANopen communication is handled by commercially available software components, which users link to their applications as a library. Then the model (here: test model) for the communication portion of a CANopen ECU only needs to be detailed enough to permit generation of the necessary testcases.

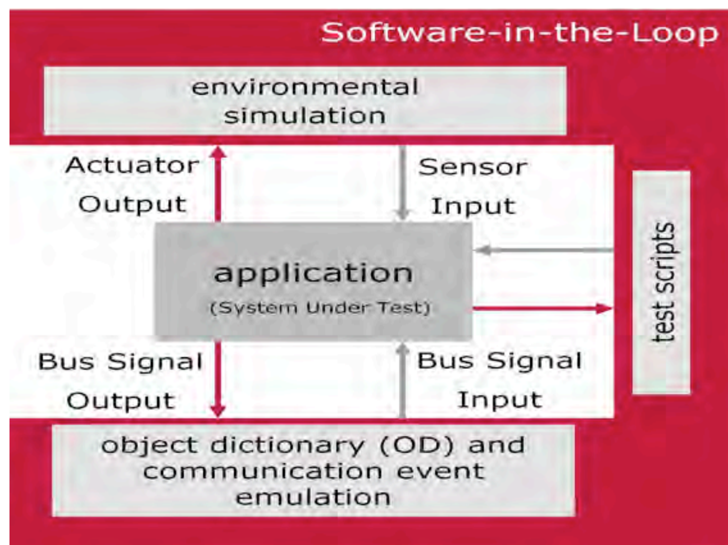
In practice, it has been demonstrated that a statically preconfigured model is entirely sufficient to generate the related tests and simulation. The user only needs to input parameters in the model (number of PDOs and layout of the object dictionary, etc.).

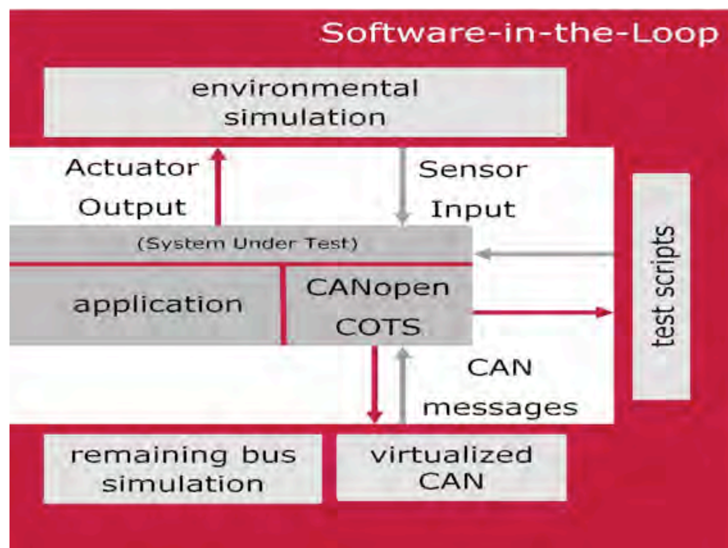
For the application portion of the CANopen ECU, the work process also begins with requirements in a specification, and the relevant models are constructed from this information. Actually, this is ALWAYS a manual process that requires tremendous know-how. At any rate, each individual requirement must be evaluated and find a place in the model. Often, this activity is already part of the software development process for the relevant ECU, and the resulting models can also be used to create the test sequences.

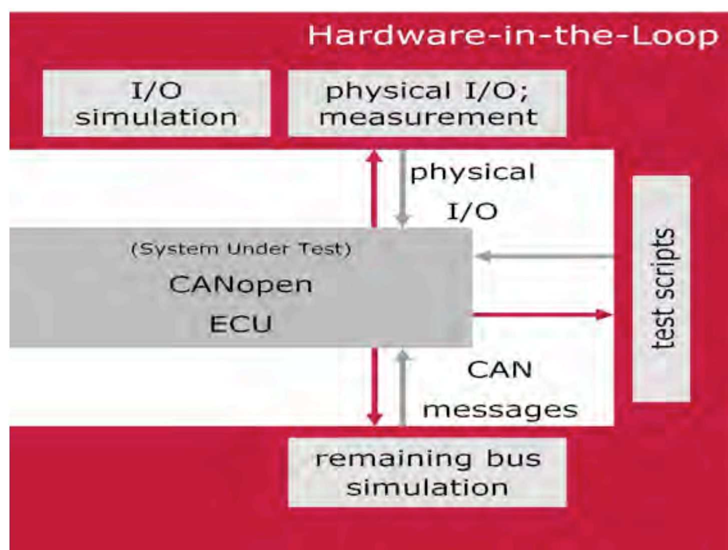
Integration levels and execution environment

As the integration level (model, component, software, ECU) changes during development, the requirements for the individual test execution environment also change. The following integration levels are distinguished here:

- Model-in-the-Loop (MiL): The model (system model, executable specification) is subjected to a test. That is, input vectors are automatically changed, and observations are made of whether the output vectors remain within an expected range. Many modeling environments today already offer the capability of executing tests on this level.
- Software-in-the-Loop (SiL): On this integration level, the implemented algorithms can be tested on the code level. The code of the individual software components is executed on the simulation platform. Often, it is a mixture of generated code, hand-written code and driver or function libraries.
- Processor-in-the-Loop (PiL): The code is now executed on the target processor or on an 'instruction set simulator'. The system environment is often still simulated on this integration level as well.
- Hardware-in-the-Loop (HiL): The transition from Processor-in-the-Loop Integration is fluid. Now the physical environment is also represented via real digital I/O or PWM (pulse width modulation) signals.


Software-in-the-Loop
environmental simulation
Actuator Output
Sensor Input
application
(System Under Test)
test scripts
Bus Signal Output
Bus Signal Input
object dictionary (OD) and communication event emulation


Software-in-the-Loop
environmental simulation
Actuator Output
Sensor Input
(System Under Test)
application
CANopen COTS
test scripts
CAN messages
remaining bus simulation
virtualized CAN


Hardware-in-the-Loop
I/O simulation
physical I/O; measurement
physical I/O
(System Under Test)
CANopen ECU
test scripts
CAN messages
remaining bus simulation

Figures 3 through 5 – Integration levels (indicates requirements for the test adapter)

To efficiently conduct tests on the given level, a very flexible test execution environment must be provided. The following aspects present a special challenge:

- The test sequences used on the specific integration levels should be reusable. Therefore, the test sequences must be convertible, or suitable interfaces must be available for bypassing the different levels.
- It must be possible to adapt test sequences to a wide variety of test adapters. Here, it makes sense to support multiple programming languages, so that adjustments can be made flexibly and cost-effectively by means of interface changes.
- To attain reproducible results, it must be possible to automate all test sequences.
- The execution environment must lend itself to a high level of configurability by parameters. The ability to generate the entire environment or parts of the environment is a good idea.

Architecture of the test execution environment for CANopen development

As can already be seen from the device model (Figure 2), in a CANopen ECU the application and communication portion (with process data objects (PDO) and service data objects (SDO)) are loosely interrelated via the object dictionary. Therefore, it is possible for the developer to begin with the design and implementation of the application portion without having to deal with CANopen in detail. However, the following points must be considered so that the execution environment can be used relatively simply on the SiL and HiL integration levels:

- The NMT state information of the communication layer should be evaluated. Even if this aspect does not entirely or clearly originate from CANopen specifications, it is still necessary to have an application react appropriately to changes in the communication state. This makes it possible to couple the finite state machines (FSM) to one another.
- Access to all application parameters that are located in the object dictionary should be encapsulated so that these parameters can be easily simulated.
- All accesses to hardware should be simulated via a hardware abstraction layer (HAL).

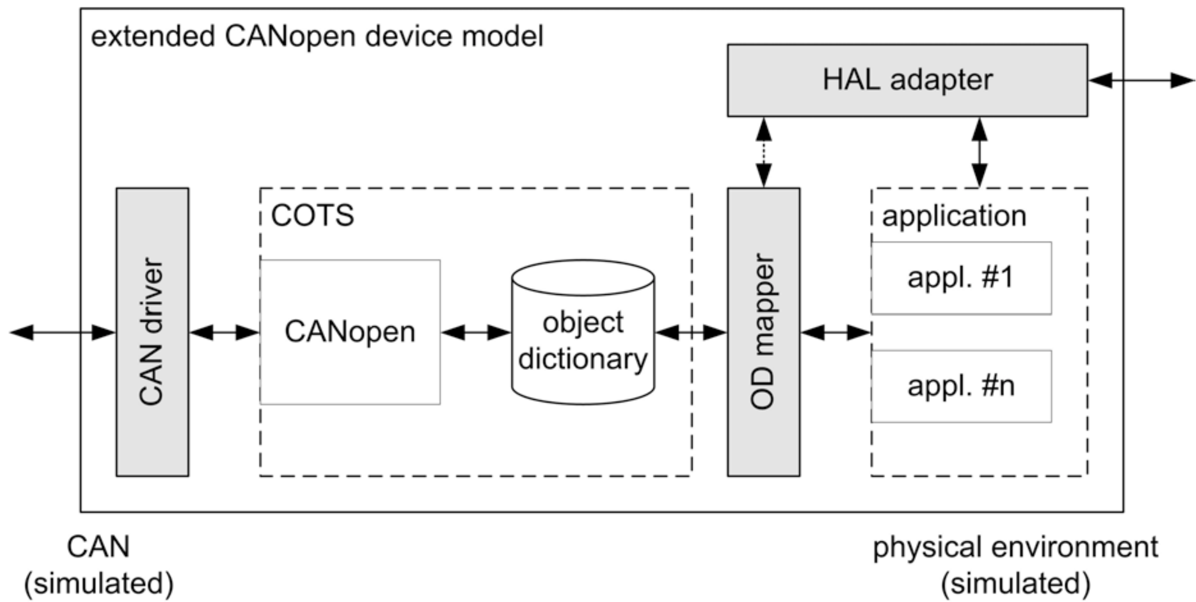


Figure 6: Extended CANopen device model

This implicitly extends the device model by adding the interfaces shown in Figure 6 (CAN driver; CANopen COTS component; HAL adapter; object dictionary (OD) mapper; application). As a result, the application can now be executed as a SiL on a simulation platform (under Microsoft Windows on a PC), and functional tests can be executed. The test execution environment even accepts models created with MATLAB/Simulink. Often, however, the application software is created as a library (dynamic link library; DLL) and is added to the test execution environment via the provided interfaces (application programming interface for the C language; C-API). Then the simulation environment includes full CANopen functionality, which can also cover a rest-of-bus simulation (network of multiple ECUs) and an abstraction of the physical environment. The adaptation layers (adapters) indicate changes to the relevant parameters, and the application uses the related values.

communication stub in CAPL:

```
on message pressure1_msg  ()
{
  if (NMTstate == OPERATIONAL)
  {
    putValue(pressure1, this.val);
    // get pressure from message
  }
}
```

application stub in CAPL:

```
on envVar pressure1  ()
{
  // environment has been changed
  if (getValue (this) > 1000)
  {
    // handle this
  }
}
```

Figure 7 (Sample code): Event function for pressure change and polling of an environment variable.

The application can now be completed, step by step, by adding other components that contain additional algorithms and control sequences, and a simple integration in the test execution environment is possible at each step.

The CANopen functionality of the simulation environment must be adapted to the specific requirements of the ECU under development. For this purpose, a configuration tool is used that can also process formats defined for CANopen. This always begins with an electronic description of the CANopen ECU by an electronic data sheet (EDS). This yields a configuration of CANopen communication in device configuration format (DCF). The layout of the object dictionary and configuration of the PDOs can be seen in this configuration. This information is used to generate the interfaces needed to interface to the application components under development.

Once the application has been prepared to this stage, the next step is to link the CANopen protocol stack. This is commercial-off-the-shelf (COTS) software, which already implements a whole series of CANopen services. To also integrate these components into a SiL configuration within the test execution environment, it is of course important to encapsulate all accesses to the object dictionary, hardware accesses and access to CAN messages via interfaces (refer to Figures 3 - 5; CAN driver; HAL adapter; object dictionary (OD) mapper). The application and CANopen components are combined into a library and are inserted in the test execution environment via the provided interfaces. The CANopen simulation must now be adapted accordingly, and it now only contains any rest-of-bus simulation. This adaptation is made with tool support, and the tool regenerates relevant interfaces.

Once the hardware for the CANopen ECU is available, the software is integrated – which consists of the application and CANopen services. Naturally, the test execution environment must also be adapted to the required degree. The previously simulated physical environment is now made available via real hardware.

The tests to be executed now run in a HiL configuration and also make it possible to simulate aspects with real-time relevance.

What else does the test execution environment give us? In principle, all paths for ECU testing merge together here. In addition to the environment simulation, other aspects of primary significance are test execution control and the representation of test results. While the tester can activate and start the available test sequences in test flow control, results representation ensures that test results (output as XML and HTML) are shown in a well organized and easy to understand way. In addition, all details of a test run are recorded in log files – including the bus communication.

How are the tests specified?

Along with semi-automated sequences (where interaction with the user is necessary during the test run, e.g. to input values), which are supported by special configurable dialogs, another capability of primary interest here is the specification of fully automated test sequences. To offer strong support for reusability, test sequences are organized into test modules. These test modules may exist in different formats, where these formats only differ in their syntax. The structure of the test modules is always uniform. The test execution environment supports efficient creation of tests by a whole series of Help functions (test service library), which permit generation of signal waveforms

(via stimuli functions) or simple access to CAN messages.

For maximum flexibility in creating test sequences, C# (.NET) is directly supported in addition to a manufacturer-specific C-like language. There is also the option of specifying very simple test primitives via XML files. Here, the XML file represents a pattern that is provided with test data and can be executed directly within a

test module. The advantage of this approach is obvious, but it is also possible to create the test specification directly in a human-readable format by means of a suitable XSLT transformation.

Figure 8 (sample code) CAPL vs. XML

The test modules themselves can be written by hand or be generated with suitable generators. Authoring tools are also available, which support creation of the modules (filling out of XML patterns; generation of C# modules from graphic representations).

Summary

This paper shows the different aspects of model-based testing based on the example of a CANopen ECU. The primary focus here is on creating the simulation environment, because this is the area where enormous efforts must always be anticipated. It was shown that a suitable architecture of the test execution environment enables reuse of test sequences and components of the simulation for different integration levels.

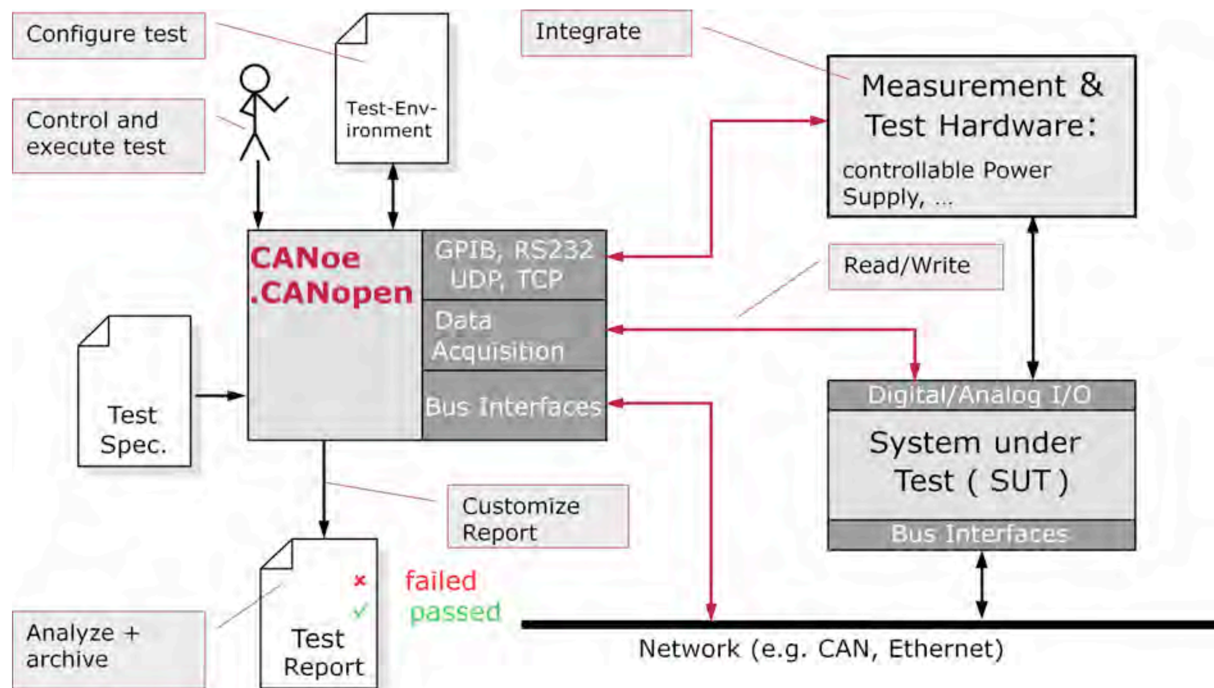


Figure 9: Test Execution Environment CANoe

Sources:

Peter Liggesmeyer: Software-Qualität - Testen, Analysieren und Verifizieren von Software (2. Aufl.). Spektrum Akademischer Verlag 2009

Kai Schmidt, Vector Informatik GmbH: CANopen tests – automatically generated,Proceedings ICC2008

Mirko Tischer
Vector Informatik GmbH
Prototyping and testing CANopen systems
Ingersheimer Str. 24
DE – 70499 Stuttgart
Tel: +49 – 711 – 80670-2152
fax: +49 – 711 – 80670 – 249
mirko.tischer@vector.com
www.vector.com