Einsatz von Debuggern im Hardware-in-the-Loop-Test
Test vom Modultest bis zum Systemtest

Kristian Trenkel, iSyst Intelligente Systeme GmbH

Dieser Beitrag erläutert den Einsatz von Debuggern für den Hardware-in-the-Loop (HIL)-Test dar. Es werden dabei zwei Einsatzbereiche vorgestellt: Zum einen der Zugriff auf die Variablen der Software des zu testenden Steuergerätes während der Laufzeit. Damit ist die Realisierung von White Box- und Grey Box-Tests im HIL-Test möglich. Und zum anderen die Messung der Codecoverage der Testfälle, ohne Eingriff in die Software des Steuergerätes. Durch die realisierte, abstrakte Schnittstelle ist der Einsatz verschiedener Debugger ohne Änderung an den Testfällen realisierbar. Es steht dabei einerseits eine generische Schnittstelle für den Zugriff auf die Variablen der Software zur Verfügung. Auf der anderen Seite ist auch die Ausführung der Codecoverage von den eigentlichen Testfällen getrennt.

Die Entwicklung eingebetteter Systeme, wie sie beispielsweise im Automobilbereich eingesetzt werden, erfordert, wegen deren ständig steigender Komplexität, einen entsprechenden Einsatz von fortschrittlichen Entwicklungsmethoden und –verfahren. Mit der Komplexität steigt im gleichen Maße der Aufwand für den Test. Die nötige Testtiefe lässt sich nur durch eine höchstmögliche Automatisierung zeitlich und wirtschaftlich realisieren.

Die Testautomatisierung, wie sie zum Beispiel beim Hardware-in-the-Loop (HIL)-Test zum Einsatz kommt, ist vor allem im Automotiv-Umfeld verbreitet. Zur Erreichung der nötigen Testtiefe, vor allem im Hinblick auf sicherheitskritische Funktionen wie Fahrerassistenzsysteme, sind reine Black Box-Tests nicht ausreichend. Es besteht die Notwendigkeit auch White Box- und Grey Box-Tests mit der realen Hardware und Software in Echtzeit auszuführen. Für den Zugriff auf die Variablen des zu testenden Steuergerätes kommt häufig das Protokoll XCP [1] zum Einsatz. Die Verwendung des Protokolls beeinflusst jedoch die Laufzeit der Software im Steuergerät.

Durch Verwendung von Debuggern in Verbindung mit den Funktionen moderner Mikrokontroller ist es dagegen möglich in Echtzeit auf Steuergeräte-Variablen zuzugreifen und diese damit im Test zu lesen und zu schreiben. Darüber hinaus ist die Ausführung von Codecoverage-Messungen ohne Instrumentierung des Codes möglich. In der Praxis stellt sich damit die Herausforderung der Integration von Debuggern in die Testautomatisierungsumgebung. Im Folgenden ist die Integration und Anwendung der Debugger BlueBox mit der Software winIDEA der Firma iSYSTEM [2] sowie der Debugger PowerDebug mit der Software Trace32 der Firma Lauterbach [3] in die Testautomatisierung iTestStudio dargestellt.

Das Tool iTestStudio

Das Testautomatisierungswerkzeug iTestStudio der iSyst Intelligente Systeme GmbH wird zur Erstellung und automatisierten Ausführung von Tests unter Verwendung der Testsysteme, z. B. HIL-Testsysteme, eingesetzt. Dabei wird auf die verschiedenen Test-Tools, wie zum Beispiel CANape, CANalyzer und ControlDesk, über Automatisierungsschnittstellen (z. B. COM) zugegriffen.

Die Software iTestStudio besteht aus verschiedenen Komponenten, welche in Abbildung 1 dargestellt sind.

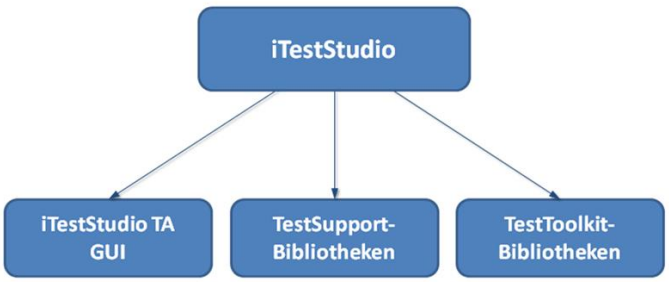

Abbildung 1: Komponentenstruktur des iTestStudio – Übersicht

Das Zusammenspiel der einzelnen Komponenten für die Testausführung ist in Abbildung 2 zu sehen.

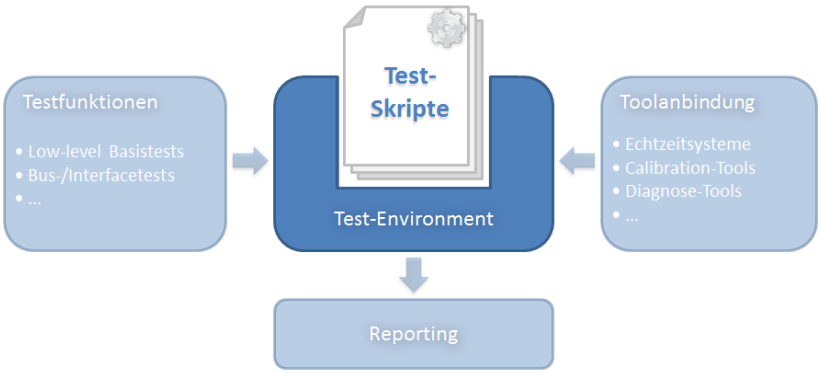

Abbildung 2: Komponenten für Testimplementierung und Testausführung

iTestStudio TA

Die Komponente iTestStudio TA (siehe Abbildung 3) stellt die eigentliche Benutzeroberfläche dar. Diese umfasst die Organisation der Testfälle mit der Zusammenstellung von Testserien und Testsequenzen (siehe Abbildung 4), das Ausführen/Starten der Testskripte und das Erstellen der Testreports.

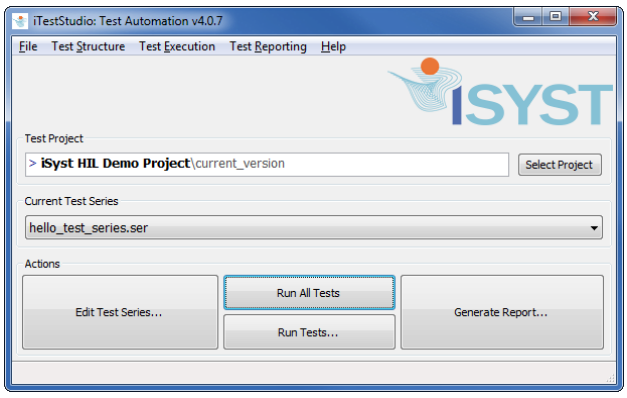

Abbildung 3: iTestStudio TA GUI

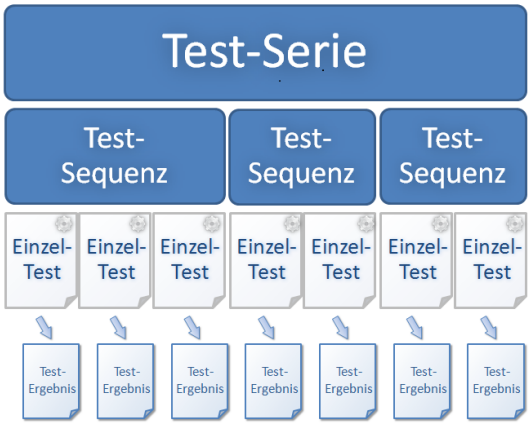

Abbildung 4: Testzusammenstellung und Generierung der Testergebnisse

TestSupport-Bibliotheken
Der zweite Teil beinhaltet die TestSupport-Bibliotheken. Diese Bibliotheken kümmern sich um die Ausführung der Tests mit der zugehörigen Python-Version, welche nicht zwingend die Python-Version der iTestStudio TA GUI sein muss. Darüber hinaus sind Funktionen zum Speichern der Ergebnisse enthalten.

TestToolkit
Die dritte Komponente ist das TestToolkit. Dieses enthält spezielle Testfunktionen, Schnittstellen zu Tools (wie Vector CANape oder dSPACE ControlDesk) sowie Container für Cal- und HIL-Variablen. Die Cal-Variablen repräsentieren die Variablen im Steuergerät, auf welche, zum Beispiel über XCP mittels des Tools CANape, zugegriffen werden kann. Für die Anbindung der Debugger wurden entsprechende Adapter zwischen den Automatisierungsschnittstellen der Debugger und den Cal-Variablen geschaffen.

Einbindung ins HIL-Testsystem
In Abbildung 5 ist der schematische Aufbau eines HIL-Testsystems zu sehen, wie er für die hier beschriebenen Tests eingesetzt wird. Eine ausführliche Beschreibung des Aufbaus ist in [4] zu finden.

Die zentrale Komponente des HIL-Testsystems ist der Echtzeitrechner, welcher die Umgebungssimulation für das Steuergerät ausführt. Darüber hinaus ist der Steuer-PC von zentraler Bedeutung. Dieser dient der Ausführung der Testfälle/Testskripte. Des Weiteren bindet er auf der einen Seite den Echtzeitrechner an. Auf der anderen Seite ist hier auch der Debugger angeschlossen.

Variablenzugriff
Für die Realisierung von White Box- und Grey Box-Tests im Rahmen des HIL-Tests ist der Zugriff auf Variablen im Steuergerät notwendig. Das Debug-Interface (z. B. BDM oder Nexus) moderner Mikrokontroller erlaubt den Zugriff auf den Speicher des Mikrokontrollers und damit auf Variablen der Software während der Laufzeit, ohne dabei die Ausführungszeit der Software zu beeinflussen. Zu beachten ist, dass nur der Zugriff auf globale Variablen möglich ist. Der Zugriff auf lokale Variablen ist nur innerhalb des jeweiligen Gültigkeitsbereiches der Variablen durchführbar, was ein Anhalten der Software an der entsprechenden Stelle notwendig machen würde.

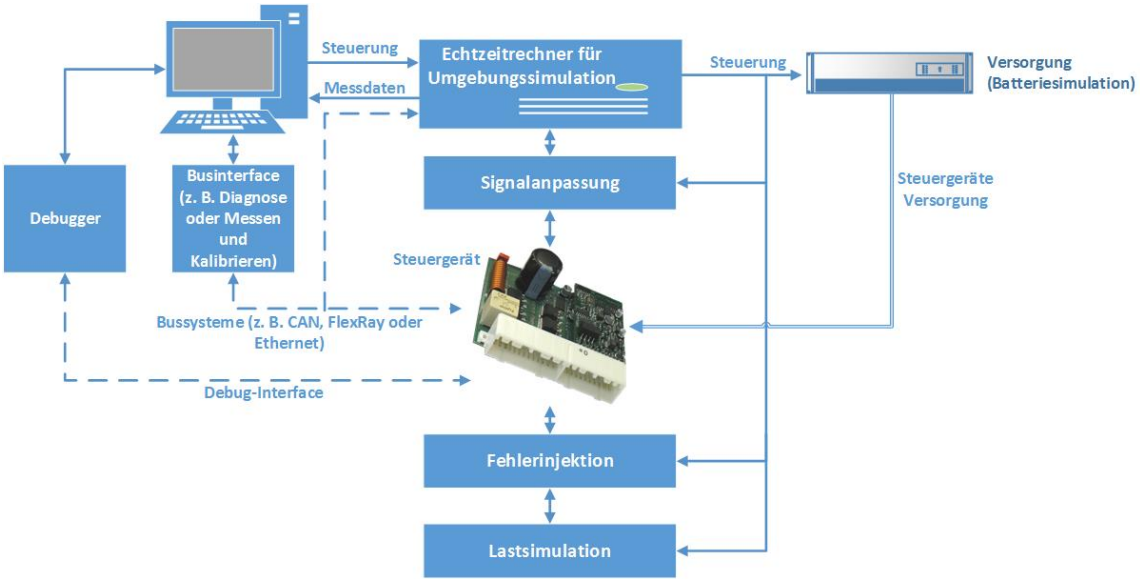

Abbildung 5: Schematischer Aufbau HIL-Testsystem

Aus Sicht des implementierten Testfalles sollte der Variablenzugriff abstrahiert sein. Innerhalb des Testfalles sollen keine Abhängigkeiten vom eingesetzten Tool bzw. der eingesetzten Schnittstelle vorhanden sein. Diese Anforderungen wurden durch ein zweistufiges Abstraktionskonzept realisiert.

Die erste Stufe bildet eine Klasse, welche eine einzelne Variable repräsentiert. Es existiert dabei eine allgemeine Basisklasse (*VarBase*), die das Interface definiert, und jeweils eine Implementierung für die verschiedenen Quellen von Variablen (z. B. CANape, winIDEA oder Trace32). In

Abbildung 6 ist ein Ausschnitt des Codes für die Klasse *EcuVar* zusehen, welche die Anbindung zu winIDEA realisiert. Hierbei ist nur der Parameter *context* der Methode *__init__* toolspezifisch. Dieser enthält die Referenz auf die Schnittstelle von winIDEA. Der Parameter *identifier* enthält den Namen der Variablen. Die übrigen Parameter erlauben es der Variablen weitere Informationen, wie zum Beispiel eine Maßeinheit, mitzugeben, um diese beispielsweise für die Ausgabe in den Testreport zu verwenden.

Als Schnittstelle für den Test stehen die Methoden *set* und *get* zur Verfügung, welche das Schreiben und das Lesen der jeweiligen Variablen, unabhängig vom eingesetzten Tool, ermöglichen.

```
class EcuVar(VarBase):
    """ Simple wrapper for debugger variables """

    # ##########################################################################
    def __init__(self, context, identifier,
                 unit="", alias="", descr="",
                 lookup=None, default=None, resettable=True,
                 cpl_id=None):
```

Abbildung 6: Variablenabstraktion – EcuVar

ie zweite Stufe der Abstraktion bildet eine Container-Klasse, welche alle Variablen für ein Testprojekt bzw. ein Steuergerät sammelt und das Handling toolspezifischer Eigenschaften (z.B. Starten des Tools) realisiert. In

Abbildung 7 ist die Definition des Variablen-Containers zu sehen.

```
class EcuVarContainer(VarContainerBase):
    """ Base class for ECU variable containers """
    def __init__(self, debugger_api):
        print "loading BlueBox variables:"
```

Abbildung 7: Variablen-Container – EcuVarContainer

Innerhalb des Variablen-Containers wird dann für jede Variable, welche im Test verwendet werden soll, eine Instanz der Variablen-Klasse (z. B. *EcuVar*) angelegt. In

Abbildung 8 ist die Anwendung der beiden Klassen zu sehen. Die Klasse *EcuVars* ist die projektspezifische Umsetzung des Variablen-Containers und enthält verschiedene Variablen. Im Beispiel in

Abbildung 8 ist dies für eine Variable zu sehen. Im Testfall ist die Variable *"(((TDFM).dummy_module).dummy_structure).State_en"* des Steuergerätes als *state_en* zugreifbar.

```
class EcuVars(EcuVarContainer):
    def __init__(self, debugger_api):

        EcuVarContainer.__init__(self, debugger_api)

        self.state_en   = EcuVar(debugger_api, "(((TDFM).dummy_module).dummy_structure).State_en",
                                 lookup={1: "UNDEFINED", 2: "INACTIVE", 3: "ACTIVE"}, resettable=False)
```

Abbildung 8: Anwendungsbeispiel des Variablen-Containers

Im eigentlichen Testfall kann nun auch auf die Variable mit *set* bzw. *get* zugegriffen werden, ohne dass bekannt sein muss, welches Tool verwendet wird. In Abbildung 9 ist beispielhaft die Anwendung in einem Testfall zu sehen. Als Erstes wird eine Referenz auf den Variablen-Container erzeugt und in der Variablen *ecu* gespeichert. Mittels *ecu.state_en* kann dann auch auf die Steuergeräte-Variable zugegriffen werden.

```
# get debugger variables
ecu = testenv.getDebug()

# set state_en to 1
ecu.state_en.set(1)

# get state_en value
state_en_value = ecu.state_en.get()
```

Abbildung 9: Anwendung im Testfall

Codecoverage

Für die Bestimmung der Testabdeckung ist die Messung des durchlaufenen Codes auf dem Steuergerät wichtig. Vor allem im Kontext sicherheitskritischer Systeme (z. B. ISO 26262) ist die Bestimmung der Codecoverage essenziell. Für diese wird meist eine Instrumentierung der Software des Steuergerätes verwendet. Dies hat aber Einfluss auf Laufzeit und Speicherverbrauch und somit unter Umständen auch auf die Testergebnisse.

Moderne Mikrokontroller erlauben es die durchlaufenen Codezeilen (durchlaufener Binärcode) mittels des Debug-Interfaces zur Laufzeit der Software in Echtzeit zu bestimmen. Diese Informationen können während der Testausführung aufgezeichnet und nach Abschluss aller Tests auf die Codecoverage des C-Codes des Steuergerätes gemapped werden.

Bei der Umsetzung der Codecoverage-Messung für das iTestStudio hat sich gezeigt, dass die Integration ohne Änderungen an den Testfällen möglich ist. Das Test-Environment, wie es in Abbildung 2 zu sehen ist, stellt einheitliche Funktionen zum Einschalten und Ausschalten des Steuergerätes zur Verfügung. Diese werden durch die Implementierung der Testfälle nur aufgerufen, während die Anbindung für die Codecoverage-Messung im projektspezifischen Teil der Funktionen erfolgen kann. Beim Einschalten des Steuergerätes wird die Messung gestartet und beim Ausschalten des Steuergerätes die Codecoverage-Information gespeichert.

Die eigentliche Auswertung der Codecoverage-Information erfolgt nach der Testdurchführung mit Hilfe des jeweiligen Tools des Debugger-Herstellers.

Anwendungsbeispiele

Test eines Lenkradschloss-Steuergerätes

In einem HIL-Testprojekt für das Steuergerät eines Lenkradschlosses stand für den Variablenzugriff keine XCP-Unterstützung zur Verfügung. Es war innerhalb des Projektes auch nicht möglich dies zu implementieren. Da die Einstufung als sicherheitskritisches System eine tiefgehende Testabdeckung erforderte, war es nötig einen alternativen Weg für den Variablenzugriff zu finden.

Mit Hilfe der BlueBox und winIDEA konnte, unter Verwendung der Python-Schnittstelle von winIDEA, ein Zugriff in die Steuergeräte-Software realisiert werden. Die Integration in die Testumgebung erfolgt, wie in Kapitel *Variablenzugriff* beschrieben.

Codecoverage-Messungen bei einem Steuergerät für eine Wankstabilisierung

Im Rahmen eines HIL-Testprojektes für eine Wankstabilisierung wurde die Messung der Codecoverage für die ausgeführten HIL-Tests notwendig. Bedingt durch die Einstufung als sicherheitskritisches System kam eine Instrumentierung des Codes des Steuergerätes nicht in Frage.

Als alternative Lösung wurde die Messung der Codecoverage mittels Debugger ausgeführt. Dabei kam sowohl winIDEA/BlueBox also auch Trace32/PowerDebug zum Einsatz. Die Einbindung in die Testumgebung erfolgte, unter Verwendung der Automatisierungsschnittstellen der beiden Tools, wie im Kapitel *Codecoverage* beschrieben.

Es konnte damit, ohne eine Änderung an den Testfällen oder an der Steuergeräte-Software, die Codecoverage der HIL-Testfälle ermittelt werden. Der Implementierungsaufwand für die Umsetzung der Codecoverage-Messungen kann als gering bezeichnet werden. Des Weiteren kann die erarbeitete Lösung in jedem anderen HIL-Testprojekt, welches das iTestStduio verwendet, eingebunden werden.

Zusammenfassung

Die Einbindung von Debuggern in den HIL-Test erweitert die Möglichkeiten des HIL-Tests auf White Box- und Grey Box-Tests, ohne dass Protokolle wie XCP verwendet werden müssen und damit die Laufzeit der Steuergeräte-Software beim Test verändert wird. Weiterhin erlaubt die Durchführung der Codecoverage-Messung mit Hilfe der Debugger die exakte Bestimmung der Testabdeckung auf Codeebene, ohne dass die Software des Steuergerätes instrumentiert werden muss. Die Messung hat somit keinen Einfluss auf die Codegröße, den Speicherverbrauch und die Laufzeit. Bei der Realisierung der Einbindung der Debugger in die HIL-Umgebung und der Anbindung an die Testautomatisierung wurde von Anfang an auf eine generische Schnittstelle Wert gelegt. Es hat sich gezeigt, dass damit der Austausch der Debugger ohne Anpassung der Tests möglich ist.

Quellenverzeichnis

[1] ASAM e.V., „ASAM MCD-1 XCP," ASAM e.V., 27 05 2015. [Online]. Available: https://wiki.asam.net/display/STANDARDS/ASAM+MCD-1+XCP. [Zugriff am 12 10 2016].

[2] iSYSTEM AG, „iSYSTEM AG," iSYSTEM AG, [Online]. Available: http://www.isystem.com/. [Zugriff am 12 10 2016].

[3] Lauterbach GmbH, „Lauterbach GmbH," Lauterbach GmbH, 12 10 2016. [Online]. Available: http://www.lauterbach.com/frames.html?home.html. [Zugriff am 12 10 2016].

[4] K. Trenkel, „From developer's tool to HIL test system," *CAN Newsletter,* Nr. 03/2016, pp. 28-31, 03 2016.

Autor

Dr.-Ing. Kristian Trenkel studierte an der FH Jena Elektrotechnik/Technische Informatik. Nach erfolgreichem Abschluss als Dipl.-Ing. (FH) im Jahr 2005 folgte eine mehrjährige Tätigkeit als Entwicklungsingenieur im Bereich Industrieautomatisierung. Ab 2008 folgte eine Tätigkeit als Testingenieur bei der iSyst GmbH und eine kooperative Promotion in Zusammenarbeit mit der TU Chemnitz. Seit 2013 betreut Herr Trenkel die Forschungs- und Förderprojekte sowie der studentischen Arbeiten bei der iSyst.

Kontakt
E-Mail: Kristian.Trenkel@isyst.de
Telefon: +49(0)911/37665–114