Testing Embedded Control Systems with TTCN-3

An Overview on TTCN-3 Continuous

Ina Schieferdecker[1,2] and Jürgen Großmann[1]

[1] Technical University Berlin, Franklinstr. 28/29, D-10623 Berlin
[2] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin

Abstract. TTCN-3 has gained increasing significance in recent years. Originally developed to fit the needs for testing software-based applications and systems in the telecommunication industry, TTCN-3 has shown its applicability to a wide range of other industrial domains in the mean time. TTCN-3 provides platform-independent, universal and powerful concepts to describe tests — especially for discrete, interactive systems — on different levels of abstraction. However, TTCN-3 addresses systems with discrete input and output characteristics only. In the automotive industry — as well as in other industries that deal with highly complex software-based control systems — this is not sufficient. Control systems often interact with their environment trough sensors and actuators using continuous signals. A test environment that adequately supports the specification, execution and evaluation of tests for embedded control systems has to provide concepts to handle this kind of signals. Moreover it has to support the test engineer with suitable abstractions that ease signal specification and signal evaluation.

1 Introduction

Embedded systems play an ever increasing role for the realization of complex control functions in many industrial domains — resulting in a big variety of requirements with respect to their functionality and reliability. Especially software-based control systems have specific characteristics, which — at least in their combination — are unique: they are typically embedded, interact with the environment using sensors and actuators, supervise discrete control flows, obtain and process simple and complex structured data, communicate over different bus systems and have to meet high safety and real time requirements. While different, model-based development processes and methods for embedded systems exist, a generally recognized test technology for the analysis and evaluation of these systems, which lead to qualitatively high-quality, safe and reliable systems, is missing. Such a test technology has to address the different aspects of embedded systems and it has to enable the testing of discrete behaviors for the communication sequences, continuous behaviors for the regulation sequences, and hybrid (i.e. combinations of discrete and continuous) behaviors for the control sequence in interaction with sensors/actuators, with other system components and the user.

R. Obermaisser et al. (Eds.): SEUS 2007, LNCS 4761, pp. 125–136, 2007.

TTCN-3 provides a standardized test environment that was originally tailored to satisfy the requirements of testing systems in the telecommunication industry, including also embedded systems. In order to use however the full potential of TTCN-3 to test embedded systems also, it needs to be extended. This paper presents concepts especially dedicated to the testing of embedded, hybrid control systems in the automotive domain.

We start with the overall definition of requirements for an integrated test environment that is viable for the automotive domain in section 2. Section 3 provides the respective TTCN-3 integration and section 4 presents an example application as a proof of concepts. A summary concludes the paper.

2 Testing Automotive Control Systems

Test processes in the automotive industry are tool-intensive and affected by technologically heterogeneous test infrastructures. The established test tools from e.g. dSPACE [1], Vector [2], MBtech [3] etc. are highly specialized, rely on proprietary languages and technologies and are closed in respect to portability, extension and integration. In recent years the application of model-based specifications in development and the establishment of powerful code generators have led the development process to be noticeably more effective, automated, and reaching a higher level of abstraction. Due to the availability of executable models, tests and analytical methods can be applied early and integrated into subsequent development steps. The positive effects — early error detection and early bug fixing — are obvious.

Nevertheless, model based approaches have to be integrated into existing development processes and combined with existing methods and tools. Hence, in the industrial practice an embedded control system has to pass several test levels such as Model-in-the-Loop- (MIL), Software-in-the-Loop- (SIL) and Hardware-in-the-Loop- (HIL) tests. Normally, different test systems are used for this purpose and almost each test system has its individual requirements on methods, languages and concepts. Moreover the whole development process is highly distributed and fragmented. The OEM, i.e. the system integrator and solution provider, is responsible for specification and integration whereas software and hardware of the control systems are normally provided by different suppliers.

However, to keep the whole development and test process efficient and manageable, the definition of an integrated and seamless approach is required. Such an approach especially addresses the subjects of test exchange, autonomy of infrastructure, methods, and platforms and the reuse of tests. The basis constitutes a domain specific test language, that is executable and that unifies tests of communicating, software-based systems in all of the automotive subdomains (telematics, power train, body electronics etc.), and that unifies the test infrastructure as well as the definition and documentation of tests.

TTCN-3 has the potential to serve as such a testing middleware. It provides concepts for local and distributed and for platform- and technology-independent testing. A TTCN-3 based test solution can be adapted to concrete testing

environments and to concrete systems to be tested by means of an open test execution environment with well-defined interfaces for adaptation. However as mentioned before: while the testing of discrete controls is well understood and available in TTCN-3 [4], concepts for specification-based testing of continuous controls and for the relation between discrete and the continuous system parts are not. TTCN-3 lacks especially concepts for specifying tests for continuous and hybrid behavior.

3 Continuous TTCN-3

To enhance the core language to the requirements of testing continuous behavior we introduce[1]:

- the notions of streams, stream ports and stream variables,
- the notions of time represented by a global clock and its sampling in test behaviors, and
- the definition of control flow structures to support the guided provisioning and evaluation of continuous behaviors (in combination with discrete behaviors).

3.1 Type Definitions

TTCN-3 provides a complex type system to define data structures and the structured assembly of testing components. To interact with the environment, TTCN-3 uses the notion of ports and distinguishes different communication characteristics for ports. To support continuous system testing we supplement TTCN-3 with so called *stream ports*. A stream port is a named variable with a history that — unlike message-based and procedure-based ports — receives a value at every step defined by the sample time. We are able to access current values and the history of a stream by using the operators `@` and `[]`. With `x_1@timevalue` we access the allocation of `x_1` at a certain point in time and with `x_1[i]` we access the i^{th} value written to `x_1`. Example stream port type definitions and their usage in component type definitions for stream ports are shown in listing 1.1.

Listing 1.1. TTCN-3 Stream Port Definition

```
type port FloatOut stream {out float}
type port FloatIn stream {in float}

type component MyComponent {
  port FloatOut x_1,x_2,x_3;
  port FloatIn y_1,y_2,y_3;
}
```

In addition to stream ports we also allow the definition of *stream variables* and *stream constants*. Whereas a port represents a connection to the outside world, stream variables and constants are the in memory representation of streams. Listing 1.2 shows the declaration of stream variables and constants and the definition of their respective types.

[1] We already presented parts of the approach shown below in previous articles [5,6].

Listing 1.2. Stream Variables and Constants

```
type stream of float FStrm;
type stream of boolean BStrm;

var FloatStrm myStrm_1;

const FStrm referenceStrm_1(t):={1.0}; // 1.0 for all t
const FStrm referenceStrm_2(t):={sin(t)}; // sinus of t
const BStrm referenceStrm_3(t):={t>10.0 ? true : false};
      // true for all t larger 10.0, otherwise false
```

Besides the basic access operators we additionally provide arithmetic operations on streams. We allow to calculate with streams (e.g. `myStrm_1+myStrm_2`), to directly compare streams (e.g. `myStrm_1>reference`) and the assignments of streams to other streams and stream ports (e.g. `x_1:=myStrm_1`).

3.2 Control Flow Structure for Continuous Behavior

TTCN-3 is a computational language. Test behavior is defined by computational algorithms that typically assign messages to ports. The evaluation at ports is realized using statements that obey the TTCN-3 snapshot semantics [4,7]. Whereas the snapshot semantics provide means for a pseudo parallel evaluation of ports, there is no notion of simultaneous stimulation and sampled evaluation on ports.

The *carry-until-statement* serves as an environment that provides a local time property `t`, sampling, and enables pseudo simultaneous stimulation and evaluation. Listing 1.3 shows the base structure of the carry-until construct.

Listing 1.3. Carry Until Construct

```
carry name_1{
  statement_1;
  statement_2
}
until {
  [] event_1 {statement_3}
  [] event_2 {statement_4; repeat}
}

carry name_2{
  statement_3
}
until {
  [] event_3 {statement_5; name_1()}
  [] event_4 {statement_6; goto name_1}
}
```

The statements enclosed by the carry block are executed iteratively, once at every step defined by the sampling rate. This repeats as long as no event is triggered in the until block. When events are triggered, the execution of the

carry block is stopped and the statement block that is defined in conjunction with the triggered event specification is executed.

The carry-until construct can be named (similar to functions and test cases in TTCN-3). The name may be used as a label, so that it may serve as a destination for goto statements. This provides the ability for more complex control flow specifications (see listing 1.3). The concise semantics of carry-until are best explained using terms of existing TTCN-3 statements. Listing 1.4 shows the mapping result of the first carry-until-statement defined in listing 1.3.

Listing 1.4. Carry Until Mapped to TTCN-3 Constructs

```
label:=cu_name_1;
var boolean continue:= true;
var float t:=0;
timer step;                                                    4
do {
      step.start(sample_rate);
      statement_1;
      statement_2;
      alt {                                                    9
      [] event_1 {
                   statement_3;
                   continue:= false;
                 }
      [] event_2 { statement_4; }                             14
      }
      step.timeout();
      t:= t+sample_rate;
} while continue;
```

We can use the carry-until construct to realize equation systems with TTCN-3. We are able to simultaneously assign new values to stream ports. Usually this is done inside the carry part by using the assignment operator (e.g. `port@t:=2*t` or `port@t:=var@(t-10)` etc.). We can also evaluate ports. The latter is done inside the until part by using compare operations (e.g. `[]port@t>var@t`). The symbol `t` represents the current execution time and is updated according to the definition in listing 1.4. For the evaluation of inputs at ports we consider delayed effectiveness, i.e. each assignment to an output port is available for input not before the next iteration. This condition holds if for every t, the values of the output ports are defined by use of the values of the input ports for $t' \leq t$ and by use of the values of the output ports for $t' < t$ only. For more details on the power and expressiveness of the carry-until construct see [5,6].

3.3 Construction of Streams

The most elementary form of a stream definition is given by an expression over time. The expression my contain a property called `t` to express time progress. The property `t` represents the local time of a stream (it starts with the value `0.0` whenever a

stream is started). For stream expressions, we use the full range of TTCN-3 expressions including the use of variables, functions and the newly introduced concepts defined above (e.g. streams and stream access operators). Listing 1.5 presents the definition of a constant integer stream and of two time dependant float streams.

Listing 1.5. Simple Streams

```
myFirstStrm:=4;
mySecondStrm:=sin(t)+100.0;
myThirdStrm:=mySecondStrm@(t-10.0)*4.0;
```

Furthermore, we allow the part-wise definition of streams. In principle, this is similar to the definition of $\langle m_k \rangle$ by use of an TTCN-3 array (see listing 3.3). Concerning the large amount of data normally necessary to represent a sampled signal, this approach is not feasible for signals with a significant length. To forgo the explicit specification of each individual value, we enable the definition of larger structures called substreams here. To define substreams we use the range expression — normally used to express template values in TTCN-3 — to address a multitude of numeric or time-related index values. For example, `myStrm@(0..100):=4` denotes the assignment of the value `4` to the first 101 time steps (and indices) of stream m̈yStrm:

Part-wise Definition of Streams

```
myFirstStrm:={1,2,45,66,223};
// finite stream with 5 values
mySecondStrm:={
  @(0..100)          :=sin(t)*100.0,
  @(100..2000)       :=4.0,                                    5
  @(2000..infinity) :=4.0+sin(t/20.0)}
// infinite stream with 3 phases
myThirdStrm:={
  [0..100]           :=sin(t)*100.0,
  [100..2000]        :=4.0,                                    10
  [2000..infinity]  :=4.0+sin(t/20.0)}
// the same as mySecondStrm only if the sampling rate is 1
```

In order to concatenate multiple stream assignments into one individual stream we use the shorthand notation for ranges presented in listing 3.3.

3.4 Stream Templates

In TTCN-3, especially for the definition of reference values, the use of templates is encouraged. A template describes a pattern that is used to characterize values. An arbitrary value is either matched by a template or not. We extend the notion of streams to include also streams of template matches, where every match or mismatch is represented by a Boolean true or false. Besides for numerical values, TTCN-3 encourages the use of templates for values of each type: for base types as well as for user defined types. We apply the notion of templates to

streams, but restrict it to *stream templates* for numerical streams and to the definition of upper and lower ranges for the stream value only. More complex stream templates will be subject to further research.

A stream template defines a pattern that characterizes a stream or a multitude of streams. The most elementary stream template is an individual stream itself (see listing 3.4). Such a template matches only if the stream that serves as template definition is exactly the same as the one the template is compared with.

Stream Templates

```
type stream of float FS;
template FS t_1:=100.0*t;
template FS t_2:=(100.0..200.0);                                    3
template FS t_3:=(sin(t)+100.0
                  ..sin(t)-100.0);
```

Similar to scalar value templates for numerical types (e.g. `float`, `integer`), we allow the definition of stream templates that denote the upper and lower bounds for numerical streams. Listing 3.4 presents example stream templates. The template `t_1` is defined by a stream itself, template `t_2` is defined by a range with the constant lower bound `100.0` and the constant upper bound `200`, and template `t_3` uses dynamic evolving boundaries defined by stream expressions.

Moreover, ranges — especially in the field of signal processing — are often used to indicate tolerances, either as a fixed value tolerance or a relative (percentage) value tolerance. To express tolerances explicitly, we introduce an additional syntactical construct for range definitions. With `(strmval|tol)` we denote a range with a fixed value tolerance `tol` and with `(strmval|tol%>` we express percentage value tolerances `tol%`. Example template definitions are shown in listing 1.6 and their application is shown in section 4.

Listing 1.6. Tolerance Templates

```
template FS t_4:=(100.0|5.0);
template FS t_5:=((sin(t)+100.0)|5%);
template FS t_6:=(template1,(4.0|4%));
```

In order to apply stream templates on streams received via stream ports, we use the match operator already known in TTCN-3. A match expression can either be applied to a complete stream (e.g `match(t_2, strmval)`) or to single values on a certain point in time (e.g.
(`match(t_2@t,strmval@t)`). For usage within the carry-until construct, we propose `x_2.match(t_2)` as a shorthand to apply the current template value of `t_2` to the current value at a stream port `x_2`.

4 Case Study

In order to demonstrate the usage of the introduced concepts and to show their applicability to automotive test tasks, we provide a small case study that represents a TTCN-3 realization for testing an *Adaptive Cruise Control* (ACC[8]).

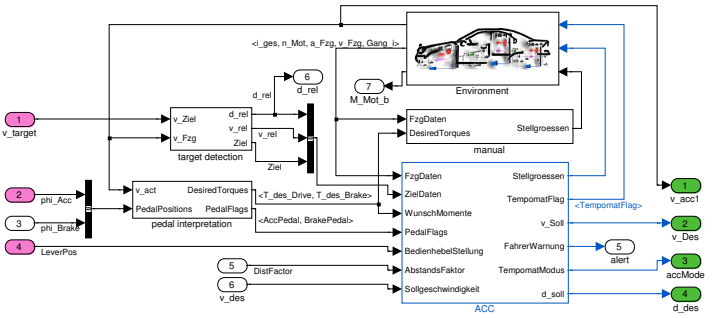

Fig. 1. Simulink Model of an Adaptive Cruise Control

4.1 The System Under Test

An ACC is a cruise control that automatically detects vehicles running ahead and — in the case of a slow vehicle ahead (the target) — it adjusts the actual speed (`v_acc`) so that a safe distance to the detected vehicle is guaranteed (the distance control mode). When there is no vehicle ahead, an ACC works like any other cruise control (the velocity control mode). Today there are different variants of ACC's available. We use a simplified example here (see Figure 1) that does not match the requirements of a real product but that is sufficient to demonstrate the new features that we have introduced for TTCN-3.

The ACC consists of three major parts. The ACC-control unit provides the main functional behavior. It is responsible to calculate the desired velocity (`v_soll`), the destination distance to the vehicle ahead (`d_des`) and it provides a warning signal to inform the driver when the safe distance is violated. The ACC-control is supplemented by the target detection unit that preprocess sensor information on the velocity of the vehicle ahead (`v_target`) and the pedal interpretation unit that preprocess the driver's input (`phi_brake, phi_gas`). For testing purposes we use the test interface specified in table 1.

Table 1. ACC Test Interface

symbol	dir	unit	datatype
`v_target`	in	m/s	double
`phi_gas`	in	%	double
`leverpos`	in	-	enumeration
`v_des`	out	m/s	double
`d_des`	out	m/s	double
`accMode`	out	-	boolean
`v_acc`	out	m/s	double

4.2 The Informal Test Specification

In this example we test the changeover between distance control mode — when there is a slow vehicle ahead — and velocity control mode — without any vehicle ahead. This forms one of the more complex tasks of an ACC and can be tested by the following test behavior (for a similar test specification see [8]).

1. **Init:** Accelerate the vehicle `phi_gas:=100` until velocity `v_acc` rises to more than 40 m/s. Than switch on the cruise control `leverpos:=HOLD_ACC`.
2. **Activate ACC:** Now, we introduce a vehicle ahead by setting the target velocity to `phi_target:=35+sin(t)` m/s. The initial distance `d_init:=90` m is set using the parameter interface that is out of scope here. The ACC should switch to distance control mode after a few seconds. The safe distance to the vehicle ahead can be calculated with `v_acc/2*df`. The symbol `df` represents a distance factor that is set to the value of `2` here.
3. **Accelerate Target:** To test whether the ACC switches back to velocity control mode, we accelerate the target vehicle using a smooth ramp. The ACC should now adjust the actual velocity according to the vehicle ahead as long as the velocity excess `40` m/s. We allow a tolerance of `10%` here. Afterwards, the ACC should switch back to velocity control mode.

4.3 The TTCN-3 Test Specification

We use TTCN-3 and the concepts specified in section 3 on page 127 to implement the test case. We start with the specification of the test system architecture and the test interface. Please note that TTCN-3 uses a test system centric perspective, i.e. system inputs are declared as outputs here and system outputs as inputs. We declare continuous stream ports to cover the velocity (v_des, d_des, v_target), the pedal output (phi_gas), and the input of the actual acc status (accMode). To set the lever position we choose a port with a message based communication characteristics.

Listing 1.7. Test Architecture

```
type port FloatOut stream {out float};
type port FloatIn stream {in float};
type port BoolIn stream {in boolean};

type enumerated Lever { MIDDLE, HOLD_ACC,                    5
                        HOLD_DEC, OFF };
type port LeverOut message
  { out Lever };

                                                             10
type component ACCTester {
  port FloatOut v_target,phi_gas;
  port FloatIn v_des,d_des, v_acc;
  port BoolIn accMode;
  port LeverOut leverpos;                                    15
}
```

After the definition of the structural setup, we define constant values and streams that are used for the stimulation of the system later on.

Listing 1.8. Constants and Stream Constants used during Stimulation

```
type stream of float FS;
const integer INIT_SPEED:=40;
const integer INIT_T_SPEED:=35;
const integer KICKDOWN:=100;                                    4
const integer TIMEOUT:=20000;

const FS target_speed:=INIT_T_SPEED+sin(t);
const FS accelerate_slow:=t/1000.0;
```

For the evaluation of system behavior we define stream templates. The template `s_dist_fail` covers an essential safety requirement. The desired distance shall never under-run the minimal safety distance. The template `v_des_fail` will be used later on to monitor the destination velocity in proportion to the velocity of the target.

Listing 1.9. Template Definitions

```
template FS s_dist_fail(integer df,
                        float vel):=
  complement(vel/2*df..infinity);

template FS v_acc_fail(float vel ):=                            5
  complement( vel|10%);
```

Similar to the altstep-construct already available in TTCN-3, we use the untilstep-construct to define reusable evaluation statements that are activated as defaults and so applied to multiple carry-until statements in the following.

Listing 1.10. Definition of an Untilstep

```
untilstep tout_and_safety runs on ACCTester{
  []d_des.match@t(s_dist_fail(df,v_acc@t)){
    setverdict(fail)}
  []t>TIMEOUT {                                                 4
    setverdict(fail)}
}
```

Now we are able to define the test case itself. We start with the init phase and activate the ACC when `INIT_SPEED` is reached.

Listing 1.11. The Test Case Definition

```
testcase ACC_Mode_Test() runs on ACCTester
  setverdict(pass);
  var integer df :=2;

  carry init{
    phi_gas@t:=KICKDOWN;
  }
  until{
    []v_acc@t>INIT_SPEED{
      leverpos.send(ON)
    }
  }
...
```

In the following we activate the untilstep defined in listing 1.10. This guarantees the detection of safe distance violation and timeouts. For test behavior we use carry-until to introduce the vehicle ahead (`v_target:=target_speed@t`). We also check for the beginning of the distance control mode (`[]acc_mode@t==true`).

Listing 1.12. Test ACC Mode Activation

```
  var default safety_default :=
    activate(tout_and_safety);

  carry activate_acc{
    v_target@t:=target_speed@t;
  }
  until{
    []acc_mode@t==true{};
  }
  ...
```

In the end, we check whether the ACC holds the correct velocity during the acceleration of the target and switches back to velocity control mode when the destination velocity is reached again.

Listing 1.13. Test ACC Mode Deactivation

```
  carry accelerate_target{
    v_target:=v_target@t+accelerate_slow@t;
  }
  until{
    []acc_mode==false{}
 [] v_acc@t.match(v_acc_fail (v_target@t))
    {setverdict (fail)}
  }
  deactivate(safety_default);
}//testcase
```

Occurring errors were detected during test execution and logged using the `setverdict(fail)` statement. After the test execution we are able to obtain the test result by examining the verdict value provided by the test case.

5 Summary and Conclusions

This paper reviews the general requirements for a test technology for embedded systems, which use both discrete signals (i.e. asynchronous message-based or synchronous procedure-based ones) and continuous flows (i.e. streams). It compares the requirements with the only standardized test specification and implementation language TTCN-3 (the Testing and Test Control Notation [4]). While TTCN-3 offers the majority of test concepts, it has limitations for testing systems with continuous aspects.

Hence, this paper introduces basic concepts and means to handle continuous real world data in digital environments. Therefore, we introduce streams that can be created, calculated and examined by means of continuous and potentially discretized data. Moreover, TTCN-3 is being extended with the concepts of stream-based ports, sampling, equation systems, and additional control flow structures to be able to express continuous behavior. The paper demonstrates the feasibility of the approach by providing a small example. In future work, the concepts will be completed, implemented and applied to real case studies in the field of automotive software engineering and the development of ECUs (electronic control units).

References

1. dSpace AG: Web pages of the dSpace corporation (2005)
2. Vector Informatik GmbH: Web pages of the Vector Informatik GmbH (2007)
3. MBtech Group: Web pages of the MBtech Group (2007)
4. ETSI: ES 201 873-1 V3.1.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language (2005)
5. Schieferdecker, I., Großmann, J.: Testing of Embedded Control Systems with Continous Signals. In: 2nd Dagstuhl-Workshop MBEES 2007: Model based Development of Embedded Systems (2006)
6. Schieferdecker, I., Großmann, J., Bringmann, E.: Continuous TTCN-3: Testing of embedded control systems. In: 3rd International ICSE workshop on Software Engineering for Automotive Systems (2006)
7. International Organization for Standardization: Information technology - Open systems interconnection — Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN), ISO/IEC 9646-3, 2nd edn. (1998)
8. Conrad, M.: Modell-basierter Test eingebetteter Software im Automobil. PhD thesis, TU-Berlin (2004)