

# Chapter 4

## Dependability Benchmark for Engine Control Applications in Automotive Embedded Systems

### Abstract

The pervasive use of commercial off-the-shelf ECUs (Electronic Control Units) in automotive systems motivates the increasing interest of the community in methodologies for quantifying their dependability in a reproducible and cost-effective way. Although the core of modern vehicle engines is managed by the control algorithms running inside such units, no practical approach has been proposed so far to characterise the impact of faults on their behaviour or to compare alternative solutions concerning dependability features. This chapter specifies a dependability benchmark for engine control systems. It illustrates through a prototype, how this specification can be implemented and at which cost and effort. First, the essential features of automotive engine control systems are captured in a general model, which is then exploited in order to define a standard procedure to assess dependability related measures. It is worth noting that these measures are aimed at comparing different control applications in order to guide the purchase decisions that industrials must make when integrating such applications in their automotive systems. Second, the chapter illustrates how the benchmark specification can be implemented and it identifies the most important problems that one must solve during that phase. Third, the prototype is exploited in order to benchmark and compare the dependability of two different diesel engine electronic control units. This practical work is then used in order to reason about the portability of the benchmark, its low intrusiveness in the system under benchmarking, the repeatability of the experiments and the representativeness of the measures. Finally, the chapter presents conclusions.

## 4.1. Introduction

In order to reduce temporal and economical development costs, automotive system developers are increasingly resorting to commercial off-the-self components (COTS), even when these components are to be integrated in critical parts of the vehicle. This situation motivates the need of new methodologies for quantifying, at least partially, the dependability of these components with comparison purposes, and this, despite the lack of information issued from their development.

Electronic control units (ECUs for short) are at the core of the development of most modern automotive control systems, such as engine and anti-lock braking systems. These units are typically manufactured as systems-on-chip (SoCs) in order to increase their scale of integration and their performance [Leen and Heffernan 2002]. Among other advantages, they are easier and cheaper to program, calibrate and maintain than the older (most of times mechanical) control systems. The wide spectrum of possibilities offered by these ECUs motivates their success in the automotive industry.

This chapter focuses on the specification of dependability benchmarks for automotive control applications running inside engine electronic control units. The goal is to provide means to support the purchase/selection decisions made by automotive engineers when integrating such applications in the ECUs they produce. In order to perceive the importance of this work, one should understand that engine control applications are corner stones in the construction of powertrain systems that harness the energy produced by the engine of a vehicle in order to produce motion. Their role in such systems is critical and it consists in managing the core of motor vehicles in order to maximize the power obtained from the combustion of air and fuel, while minimizing the production of pollutants [Heywood 1988]. Hence, incorrect control actions on the vehicle engine may not only damage the engine itself, but also have a negative impact on the safety of the vehicle passengers.

The importance of engine control applications in modern combustion vehicles motivates the need of methodologies and tools for evaluating the impact of their failures over the engines they control. In our context, the notion of dependability benchmarking must be thus understood as the evaluation, of the safety of the ECU software from perspective of the engine. Most of the existing research in the domain concentrates on the verification of engine control algorithms according to their requirements [Fuchs et al. 1998; Leveson 2000]. Some research has also been performed on how to implement robust control strategies for tolerating the mechanical or electrical faults that may perturb the behaviour of electronic control units in general [Spooner and Passino 1997]. More focused on benchmarking, the Embedded Microprocessor Benchmark Consortium (EEMBC) has developed a meaningful set of performance benchmarks for embedded processors and compilers. The EEMBC benchmarks are composed of dozens of algorithms organized into benchmark suites targeting telecommunications, networking, automotive and industrial, consumer, and office equipment products. Among these benchmarks, AutoMark [EEMC 2003] is devoted to the evaluation of the microprocessors performance when running automotive and industrial algorithms, such as road speed calculation and angle to time conversion. The main difference with the type of

benchmark we consider relies on the fact that AutoMark focuses on the hardware in spite of the software running inside the ECU.

From our viewpoint, benchmarking the dependability (safety) of engine control solutions mean, at least, studying to what extent (i) the external faults that may affect the execution of the ECU control algorithms, may perturb the control produced by this unit; and (ii) the (eventual) impact of this faulty control over the controlled engine. It is obvious that this study is conditioned by the viewpoint of the benchmark expected users. As previously stated, the goal is, in our case, to support the selection decisions made by automotive engineers when integrating engine control software in their power train systems.

The rest of the chapter is structured as follows. Section 4.2 refines the notion of engine control application introduced above and defines a general model for such kind of applications. Section 4.3 exploits this model in order to specify a standard procedure to assess dependability related measures of an engine control application. Then, Section 4.4 exemplifies this benchmark specification through a prototype implementation for diesel engine ECUs. Through this practical work, we illustrate the cost of implementing the proposed benchmark and the time required for its execution. Section 4.5 discusses to what extent the general properties described in Chapter 1 are verified by our benchmark and Section 4.6 presents conclusions.

## **4.2. Basics on Automotive Engine Control Systems**

As stated in chapter 1, categorising a dependability benchmark means specifying its benchmarking context. The amount of different automotive embedded systems existing in today's vehicles and their heterogeneity prevent the definition of a unique methodology for benchmarking any sort of automotive control application. Thus, a first step towards the specification of useful automotive benchmarks must be the clear definition of the benchmarking context. This context defines (and restricts) the application domain of the resulting benchmark specification. Then, it is important to understand the characteristics of the automotive embedded systems included in the considered domain. This enables the definition of general models that are representative of a large spectre of these automotive systems. These models are finally the abstractions on which dependability benchmarks for such systems can be then specified.

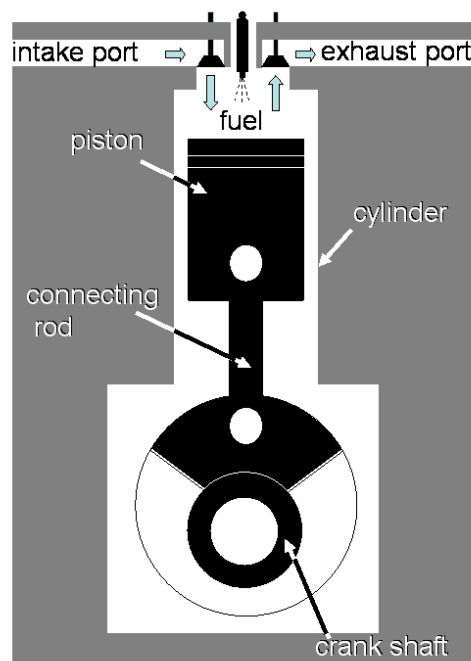
According to the above reasoning, this section introduces the basics required for understanding the type of automotive systems to which our benchmark can be applied. These systems are four-stroke engines. Then, it is proposed a general model that captures the main characteristics of the control applications managing such type of systems.

### **4.2.1. How Engines Work**

Engine control applications are responsible for the injection of a certain amount of fuel (gasoline or diesel) in a small, enclosed space and its combustion. As a result, a certain amount of energy is released in the form of expanding gas. Basically, an engine works in a cycle that allows setting off this combustion process thousands of times per minute. Then, the produced energy is harnessed in order to generate motion.

Most modern vehicles embed engines working with a so called four-stroke combustion cycle: These four strokes are the intake, the compression, the combustion and the exhaust stroke. The following points summarize what happens as the engine goes through its cycle:

1. The piston starts at the top, the intake valve opens, and the piston moves down to let the engine take in a cylinder-full of air. This is what happens during the intake stroke in diesel and direct injection gasoline engines. In the other types of gasoline engines, the intake stroke fills the cylinder with a gasoline-air mixture.
2. Then, the piston moves back up to compress the cylinder content. Compression makes the explosion more powerful. Diesel engine compression ratios are greater than gasoline ones.
3. When the piston reaches the top of its stroke in a diesel engine, the fuel is then injected into the compressed air. The heat of the compressed air lights the fuel spontaneously. In non-direct injection gasoline engines, the fuel has been already injected during the intake stroke. Thus, when the piston reaches the top of its stroke, the spark plug emits a spark to ignite the gasoline. In both gasoline and diesel engines, the combustion of the air/fuel mixture drives the piston down.
4. Once the piston hits the bottom of its stroke, the exhaust valve opens and the smoke leaves the cylinder to go out the tail pipe. Now the engine is ready for the next cycle, so it intakes, depending on the type of engine, another charge of air or air and fuel.



**Figure 4-1. High-level view of a (diesel) engine**

Typically, the linear motion of the piston is converted into rotational motion by the engine crank shaft, which is connected to the piston by a connecting rod (see Figure 4-1). The rotational motion is the one needed to rotate the car's wheels.

As this brief description shows, all four-stroke engines present similarities, despite the type of fuel (gasoline or diesel) they combust, that can be exploited in order to define a general model of the type of systems required to control them. Interested readers can find further details about how vehicle engines work in [Pulkrabek 1997].

#### 4.2.2. Engine Control System Model

According to the general view of a four-stroke engine provided in the previous section, our goal here is to identify the main characteristics of the systems controlling these engines. These characteristics conform the general model on which the specification of our dependability benchmark is based (see Figure 4-2).

The notion of engine electronic control unit defined in the introduction of this chapter refers to the compound formed by the engine control application and the software and hardware supporting its execution. Typically, engine control applications run on top of a real-time operating system support and a microcontroller- or DSP<sup>1</sup>-based hardware platform [Miller et al. 1998]. As Figure 4-2 shows, these applications can be modelled as a set of control loops handling two types of input data: (i) the acceleration/deceleration data provided by the driver through the throttle, and (ii) the feedback data supplied by the set of sensors connected to the car engine. The first type of data is external to the engine and it defines the speed reference imposed by the vehicle driver. The second one is internal and it is used in order to monitor what is currently happening inside the vehicle engine.

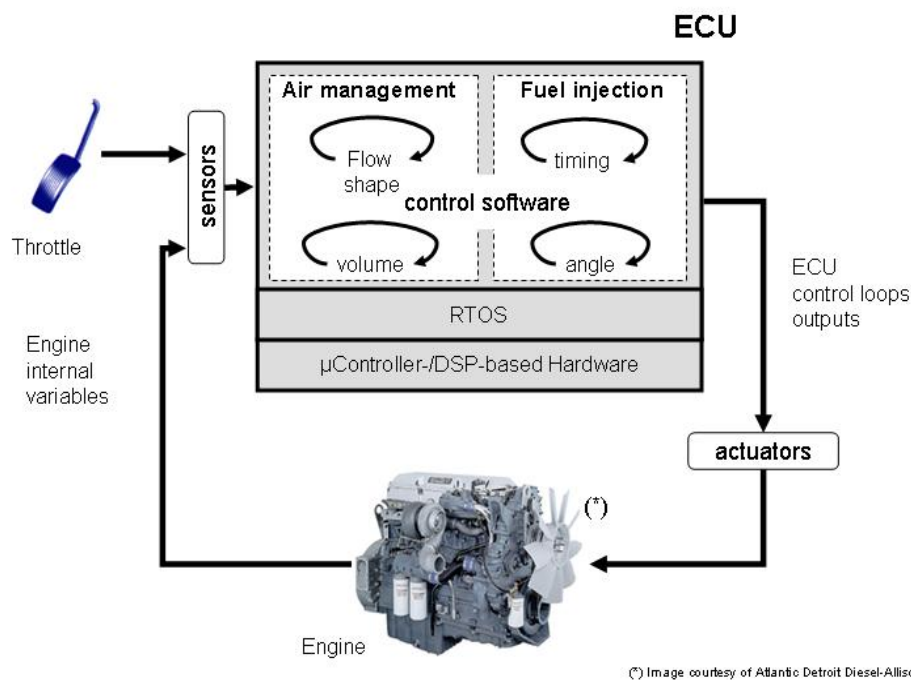


Figure 4-2. Model for an engine control system

<sup>1</sup> The acronym DSP stands for Digital Signal Processor.

The core of the engine control application can be characterized through four different control loops. On the one hand, the *angle* and the *timing* loops determine when the fuel injection must occur and how long it must be. On the other hand, the *volume* and the *flow shape* control loops manage the quantity of air and the way that air enters the cylinder. These four control loops are the ones that any (gasoline or diesel) engine control system should implement. However, one should notice that, due to their specific characteristics, each engine family also implements other specific control loops. Examples are, for instance, the EGR<sup>2</sup> control loop of diesel engines and, the spark control loop of gasoline engines. It is worth noting that this also happens among engines of the same family.

It is important to notice that engine control loops are typically designed to provide robust control. Although similar to fault tolerance, the notion of robust control only refers to the ability of a control algorithm to absorb perturbations in its inputs in order to compute correct outputs. In the automotive domain, this robustness is obtained by (i) applying filters to the inputs in order to eliminate the eventual perturbations and (ii) using predictive techniques that compute outputs based not only in the current set of inputs, but also in the previous state of the system and the dynamics the engine is supposed to follow. In that way, the impact of an incorrect input over the computed outputs is minimized. However, the question persists: what happens if the effect of an incorrect input (or other type of external fault) propagates to the computed control outputs? To the best of our knowledge, engine control applications do not integrate specific mechanisms to recover from the effects of such type of failures. As a result, if the consequence of a fault is the computation of an improper control output, then this output will be applied to the engine as it is. This is the type of unsafe situation whose consequences are evaluated by our dependability benchmark. The best option among several candidates will be the one reducing the consequences and the ratio of occurrence of such unsafe situations. Since our purpose is not to define a benchmarking solution for a particular engine control system, our interest will focus all through this chapter only on the control loops that any automotive control application must provide, i.e. those depicted in Figure 4-2.

### 4.3. Benchmark Specification

Our dependability benchmark is built on top of the model defined in the section 4.2.2 . In this section we introduce the different components of our benchmark and we detail each component in a different section.

#### 4.3.1. Benchmark Overview

In the first chapter of this book, we have identified the set of different dimensions of dependability benchmarks. In this section, these dimensions are instantiated to the functional context defined by automotive engine control applications.

---

<sup>2</sup> EGR is the acronym of *Exhaust Gas Recirculation*, a technique used today in most Diesel engines in order to produce a better combustion of contaminants and ensure in that way a fewer emission of such containants to the atmosphere.

The general specification framework states that a dependability benchmark must clearly distinguish the *system under benchmarking* (SUB) from the *benchmark target* (BT). The former is the system on which the dependability *benchmark experiments* (BEs) are conducted. The latter is the benchmarking object, i.e. the system or system component which is intended to be characterized by the benchmark. In our context, the SUB corresponds to the system depicted in Figure 4-2, and the BT is the software running inside the engine ECU.

The experimental dimension of our dependability benchmark is characterized by its input domain and its output domain.

The input domain corresponds to the stimuli required to exercise the system under benchmarking (the *workload*) and the set of injected faults (the *faultload*). In our case, the workload corresponds to the speed reference supplied by the car driver through the throttle and the state feedback provided by the sensors connected to the engine. On the other hand, the faultload is defined as the set of transient hardware faults that the ECU memory could suffer during its normal operation [Gracia et al. 2002]. The aggregation of these components defines the *execution profile* of our benchmark.

In general, the output domain is the set of observations (*measurements*) that are collected from the benchmark target during the experiments. These measurements are retrieved from the experiments following a so-called *benchmark procedure*. Their processing enables the computation of the benchmark *dependability measures*. As commented in section 4.2.2, engine control applications are designed to be robust to external perturbations that may affect their inputs but they do not integrate any specific mechanism to recover from such type of faults. This is why the measures our benchmark provides basically characterise the robustness of the ECU control algorithms with respect to the faults affecting their inputs. In case of the generation of an incorrect control output (a failure), these measures also reflect the ability of the ECU to limit the impact of such failure on its controlled engine. This latter issue is of great importance since, as next section explains, this impact may vary from the non-optimal behaviour of the engine to a situation in which the engine behaviour becomes unpredictable. Accordingly, the best control system will be the one providing a safer behaviour from the viewpoint of the controlled engine.

Before closing this overview, we want to underline the fact that the high integration scales used in today's electronic components is a major impairment for conducting benchmark experiments on the considered control applications. Although technical, this issue applies in general to any modern engine ECU and it has a deep impact on the definition of a suitable and realistic benchmark procedure for such kind of automotive systems.

### 4.3.2. Dependability Measures

As stated in the previous section, the dependability measures of this benchmark evaluate the safety of an engine control system. This evaluation is not performed on the basis of a single measure, but rather on the basis of a set of measures that estimates the impact of ECU control loop failures over the engine.

We consider that an engine ECU fails when it delivers incorrect control outputs or when it does not deliver any control output at all. Accordingly, one can identify five different failure modes for an engine ECU:

1. The *Engine ECU reset* failure mode models the situation in which an error corrupts the internal registers of the engine control unit. Typically, this problem prevents the hardware platform to continue with the normal execution of the system and requires its reset. Hence, the ECU is not able to provide any control output during a certain amount of time (called the *reset time*). It is worth noting that this failure affects the service provided by the ECU, but it is independent from the computation performed by the engine control algorithms;
2. The “*no new data*” is a value failure mode that represents the case in which the engine control software hangs without providing any new control output. It also models the case in which one or more control loops enter in a computation cycle providing, despite the reads of the sensors, the same control outputs;
3. The “*close to nominal*” and “*far from nominal*” failure modes characterize failure situations in which the control application provides outputs with incorrect values. One should notice that engines are typically able to absorb certain levels of discrepancies between the expected control values (the nominal values) and the ones actually provided by the ECU control software. These discrepancies cannot be defined in general since they vary from one engine to another. Typically, discrepancies of more of the 30 % of the nominal value represent failures of type “far from nominal”. It is thus obvious that the calibration of this discrepancy has a direct impact in the benchmark results;
4. The *Missed deadline* failure mode represents a temporal failure that occurs when a deadline is violated by the control algorithms. This means that the control output is provided out of the temporal bounds fixed by the system specification or it is not provided at all.

In general, vehicle engines react to the different control failure modes defined above according to three different levels of criticality. The most critical level is the one in which the behaviour of the engine becomes *unpredictable*, i.e. we cannot determine the consequence of the control failure for the engine. This is why this situation is the most unsafe for both the engine and the vehicle passengers. The second level of criticality is *noise and vibrations*, in which the engine is noticeably not running according to its specification. Finally, the less critical impact is when the performance of the engine is *non-optimal*. Typically, this is not perceived by the vehicle passengers while driving, but it has a deep impact over the fuel consumption, the power delivered and the pollution generated by the engine.

As Table 4-1 shows, the level of criticality of each control loop failure is different. The most critical failures, from the engine safety viewpoint, correspond to those affecting the control of the fuel injection process, and more precisely the moment at which the fuel is injected in the engine cylinder. In general, the engine behaviour becomes in that situation unpredictable. However, when the fuel injection angle control loop suffers a time failure or a temporal value of type close to nominal, the engine behaves in a non-optimal mode. The table also shows






that the reset of the engine ECU is critical. This is because this failure prevents the engine ECU control loops to perform their computation. Hence, during the reset time, the engine behaviour becomes unpredictable, since it is neither monitored nor controlled by the ECU. The rest of control failures lead the engine to produce noise and vibrations or to behave in a non-optimal mode.

The reader should notice that the impact of these control failures over each engine depends on the tolerance of its mechanical components and the dynamics desired for each particular engine<sup>3</sup>. Thus, this impact varies from one engine to another.

**Table 4-1. Measures characterising the impact of ECU control failures over the engine**

		Engine ECU control outputs				
		Fuel injection		Air management		
		Angle	Timing	Volume	Flow shape	
Control failure modes	Engine ECU reset (ECU internal registers corrupted)	Unpredictable				
	Value Failures	No new data	Unpredictable	Non-optimal	Noise/Vibrations	Noise/Vibrations
		Close to nominal	Non-optimal	Noise/Vibrations	Non-optimal	Non-optimal
		Far from nominal	Unpredictable	Non-optimal	Noise/Vibrations	Noise/Vibrations
	Time Failure (Missed deadline)	Non-optimal	Noise/Vibrations	Non-optimal	Non-optimal	

-  The vehicle engine behaves in an unpredictable way (most unsafe situation);
-  The vehicle engine works, but not properly: it produces some noise and/or vibrations;
-  The vehicle engine behaviour works properly but its performance is not optimal (least unsafe situation)

The safety measures that one can deduce from Table 4-1 correspond to the percentage of failures falling in each table cell. The goal of presenting these measures in a tabulated form is to improve their readability. We also recommend colouring the cells of the table according to their criticality level. In that way, all the information can be perceived at a single glance.

### 4.3.3. Execution Profile

The computation of the benchmark measures specified in Table 4-1 requires the activation of the engine ECU with a particular execution profile. This profile has two main components: the workload and the faultload. The workload defines the type of stimuli needed for activating the engine ECU software during the benchmark experiments. The faultload characterize the faults that may perturb the normal execution of this software. The main challenge here is to define these benchmark components in a representative way, i.e.

<sup>3</sup> It must be noted that the use of electronic units enable engines dynamics to be customised according to the needs. In other words, the calibration of the engine dynamics is performed by tuning the internal parameters handled by the engine control algorithms. This is, for instance, what happens today when different models of the same vehicle, integrating the same engine, supply different horsepower indexes.

according to the set of conditions that drive the execution of engine control applications in the real world.

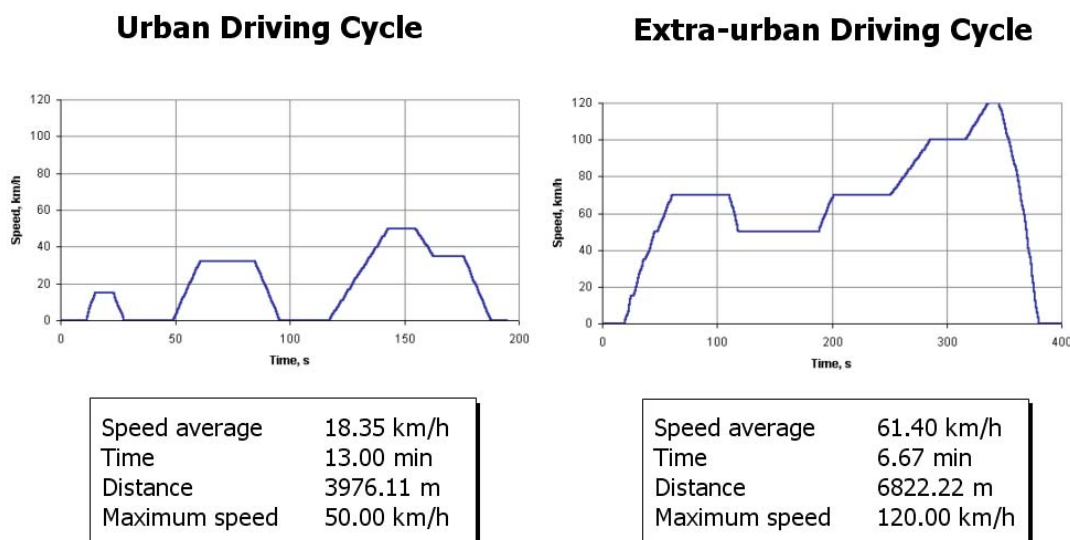
### 4.3.3.1. Workload

Despite what happens in other benchmarks, the workload needed for exercising the control software of an engine ECU cannot be simply defined in terms of a set of programs. As Figure 4-2 showed, engine control applications execute according to two types of information. The former corresponds to the speed reference the driver imposes to the engine through the throttle. The latter is defined by the set of engine internal variables that the ECU monitors in order to feedback its control computation.

#### Throttle Inputs

Defining the throttle inputs consist in modelling different ways of driving. For our purposes, we propose the use of the following driving cycles: the acceleration-deceleration driving cycle, the urban driving cycle and the extra-urban driving cycle. All these cycles are expressed in terms of engine speed, i.e., in revolutions per minute, *rpm* for short.

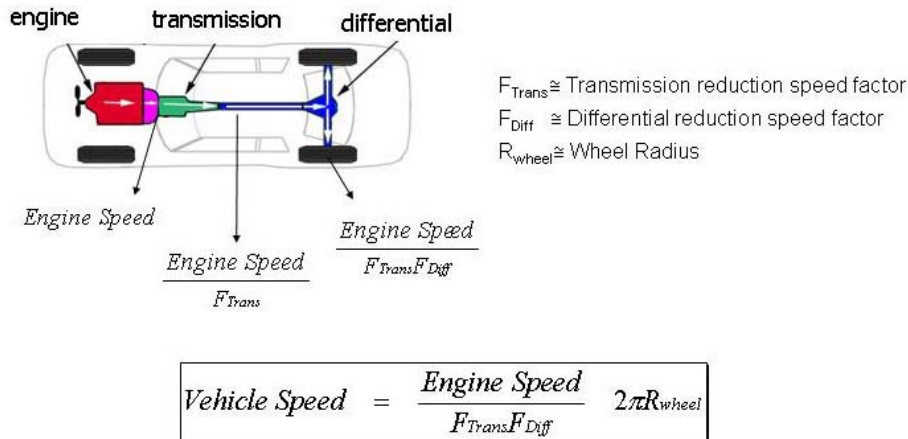
In the acceleration-deceleration driving cycle the throttle is repeatedly pushed and released trying to test all possible regulation conditions of the engine system. At the beginning of the cycle, the engine is idle and it is progressively accelerated, without putting any gear, by pushing the throttle from 0% to 100%. Then, the throttle remains pushed at 100% until the engine reaches a speed of the 80% of the maximum engine rpm. The throttle is then progressively released from 100% to 0% and it is maintained completely released until the engine becomes idle once again. Then, the cycle can be repeated again.



**Figure 4-3. Urban and Extra-urban driving cycles**

The urban and extra-urban driving cycles are standard cycles specified for emission certification of light duty vehicles in Europe [EEC Directive 90/C81/01 1999]. Figure 4-3 depicts their main features. As we can see, the standard reasons in terms of vehicle speed, i.e.

in kilometres per hour (km/h for short). In order to transform this speed into engine speed (expressed in rpm), one must use the mathematical relation presented in Figure 4-4. Conversion from vehicle to engine speeds requires the knowledge of the speed reduction factors applied by the transmission for each gear, the differential and the radius of the vehicle wheels. This information must be supplied by the benchmark user, since it varies from one engine to another.



**Figure 4-4. Parameters for converting the Engine speed (rpm) to Vehicle speed (km/h)**

The reader must understand that for the purposes of benchmarking what is important is not to apply all the driving cycles described above each time the engine ECU is benchmark, but to apply only those of them that makes more sense according to the benchmark user purposes. It must be also noticed that the results obtained when considering different driving cycles are not comparable among them.

### Engine Internal Variables

Although the benchmark experiments can be carried out using a real engine, it is evident that this solution is not suitable due to economical and safety reasons. This is easy to understand if one considers how unsafe can be the consequences of an engine ECU control failure (see Table 4-1).

In order to avoid the damage of the engine during the experiments, we propose to replace the real engine by a model. This model could have a mathematical or an empirical nature. From a practical viewpoint, a mathematical model is only a good choice if it is available and ready to use at the moment of running the benchmark experiments. If this is not the case, it seems more reasonable to use an empirical model. This model can be easily defined, for instance, by tracing the activity of the real engine while running under the working conditions imposed by the throttle.

#### 4.3.3.2. Faultload

In general, benchmarking a system component prevents faults to be injected into this component. In other words, the faultload can only affect the execution environment of the

benchmark target, but not the benchmark target itself. Our target is the software running inside the engine ECU. Consequently, faults cannot be introduced in this software; but in the hardware supporting its execution.

Recent studies [Gil et al. 2002] point out that the likelihood of apparition of hardware transient faults is increasing in integrated circuits (ICs). These results can be also applied to electronic control units, since their manufacturing greatly benefits from improvements of integration scales in IC technologies. Moreover, some fault types that usually have been neglected for such type of systems will have an important impact. This is the case of faults in combinational logic (pulse fault model). At higher working frequencies, the likelihood to latch a combinational fault is going to rise considerably. Other unconsidered fault models like delay and indeterminism are going to increase their influence also due to problems related to high speed working. At high frequencies, skin and Miller effects will make that delay in ICs will not be constant, originating even time violations that can lead to indeterminate outputs. [Gracia et al. 2002] have shown that a basic approach towards the consideration of the above fault models is the bit-flip fault model. In addition, the bit-flip fault model is largely accepted by the dependability community for modelling hardware transient faults.

The faultload we propose is defined as the set of hardware transient faults emulated by bit-flips that will affect the engine ECU memory during its normal use. The reason guiding this choice is two-fold. As motivated above, the high scales of integration used for manufacturing modern engine ECUs make such electronic components very sensitive to the interferences produced by the different types of radiations existing in the environment. On the other hand, faults impacting the ECU internal memory are those with a bigger influence on the service supplied by engine ECUs. This is easy to understand if we notice that this memory holds the code and the data of both the operating system and the engine control software. In addition, all the sensors/actuators physically connected to the ECU store/retrieve their data to/from specific locations of this memory. Figure 4-6 provides a detailed view of the relationships existing between the different elements (sensors, actuators) connected to engine ECUs, the components (memory, processors, A/D and D/A converters) integrated in such type control units and the different interfaces ECUs must provide for enabling the execution of the benchmark. Among these interfaces, the faultload interface is the one through which bit-flips are injected in the ECU memory during the experimentation phase. The description of this interface and the proposed fault-injection methodology are part of the benchmark procedure specification.

#### **4.3.4. Benchmark Procedure**

For a given engine control application, the dependability measures defined in Table 4-1 are deduced from measurements computed by comparing its behaviour in absence of faults (golden run) with its one in presence of faults. In all the benchmark experiments, the same initial SUB configuration is required. From one experiment to another, the difference relies on the definition of the experiment execution profile, and more precisely on the fault selected from the faultload.

#### 4.3.4.1. Experimental Setup components

Figure 4-5 identifies the main components of the experimental setup required for running the benchmark experiments. This setup is built according to the model depicted in Figure 4-2. Within a single experiment, the experiment manager coordinates the activity of all the setup entities. It also sequences the execution of successive benchmark experiments.

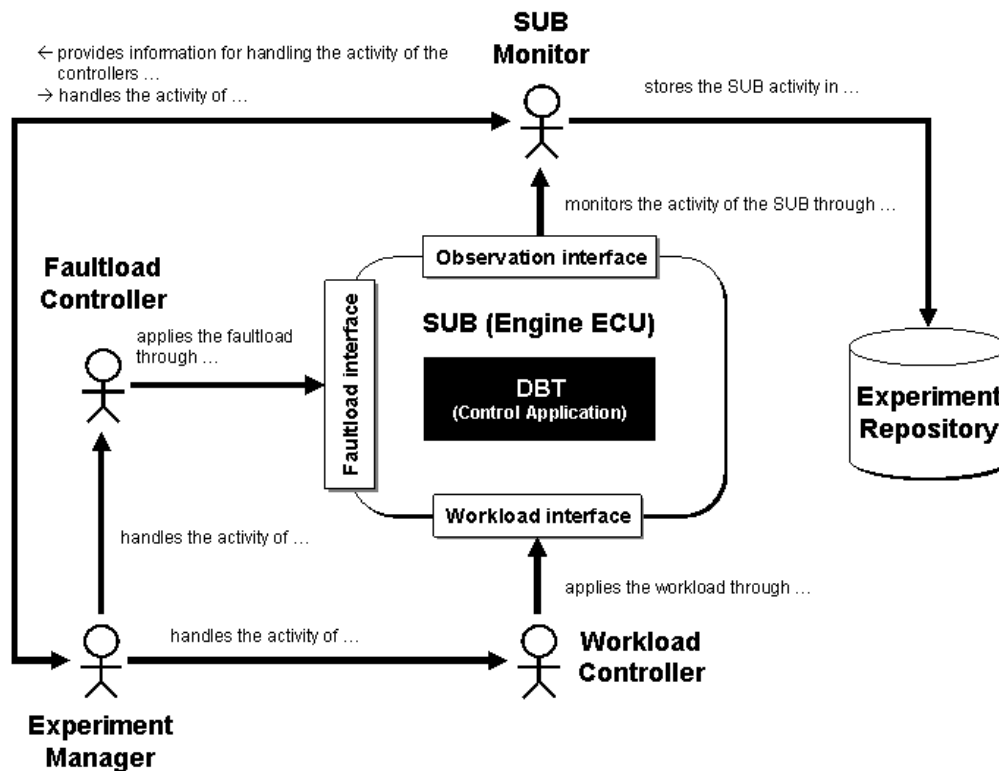


Figure 4-5. Experimental setup components

The workload and faultload controllers are those entities devoted to the activation of the SUB according to the execution profile defined for each experiment. In order to do that, they use two different interfaces: the *workload interface* and the *faultload interface*. The former is defined in terms of the sensors that feed the control algorithms of the engine ECU. The latter is the interface through which the faultload is applied to the target system. It must be noted that the definition of this second interface is not as evident as the definition of the workload interface. Further insights on that issue are provided in section 4.3.4.2.

The *observation interface* is used by the SUB monitor in order to obtain traces reflecting the target activity. We name these observations the *benchmark measurements*. These measurements are those from which the dependability measures specified in Table 4-1 are later deduced. The place where the SUB monitor stores the benchmark measurements is called the *experiment repository*.

The reader must notice that the view of the experimental setup provided here is conceptual, i.e. the different entities depicted in Figure 4-5 can be mapped to a single entity or a set of different entities in the implementation. The experiment repository, for instance, can be implemented, despite its role in the experimental setup, as either a database or a file

directory. These practical issues are however out of the scope of this specification and they will be commented in the prototype section of this chapter.

#### **4.3.4.2. Experimental approach**

The experiments with an execution profile containing one fault are carried out in three successive phases. In the first one, the experiment manager initialises the system under benchmarking, which is led to the initial conditions of the experiment. Then, it asks to the setup controllers to run the execution profile. Finally, it supervises the activity of the SUB monitor, which stores in the experiment repository the information retrieved from the target internal activity. Golden runs are experiments with a simpler benchmark procedure that consists in running execution profiles with and empty faultload.

##### **Phase 1: SUB Initialization**

First, the target application is reset. Then, an initial start-up time must be devoted to the initialization of the system under benchmarking. During this time, the engine control software and the operating system are loaded in memory and its execution is lead to a “*ready to control*” state. From a mechanical viewpoint, this initialization phase enables the engine to be started in order to warm-up and reach the required initial speed. According to initial conditions of our workload (see section 4.3.3.1), the initial speed that we need is of 800 rpm, i.e. the engine is idle at the beginning of the benchmark experiments. The internal temperature of the engine should be the one that the manufacturer considers as “*normal*” for his engine.

The injection of faults during the experiments is triggered through a timer that must be programmed during this initialization phase. Although the fault injection time can be shorter than the experiment start-up time, this situation is only of interest when studying the start-up of the system in presence of faults, which is not our case. We consider that the fault injection time is always longer than the start-up time of the system under benchmarking (see Figure 4-8).

##### **Phase 2: Execution profile running**

As Figure 4-6 shows, running the workload consists in feeding the sensors connected to the engine ECU with the adequate inputs. These inputs are those associated to the selected driving cycle and the feedback supplied by the engine. As discussed during the definition of the workload, both the throttle and the engine can be either real components or models. In this latter case, it must be noted that the sensor data is always digitalized before being stored in a set of predefined memory locations. This issue can be exploited in order to write the sensors readings directly to these locations from another computer system (the workload controller). This is a technical issue that can simplify the design of the benchmark setup since it is not necessary to produce analogical signals for feeding the sensors if we are using neither real throttles nor real engines.

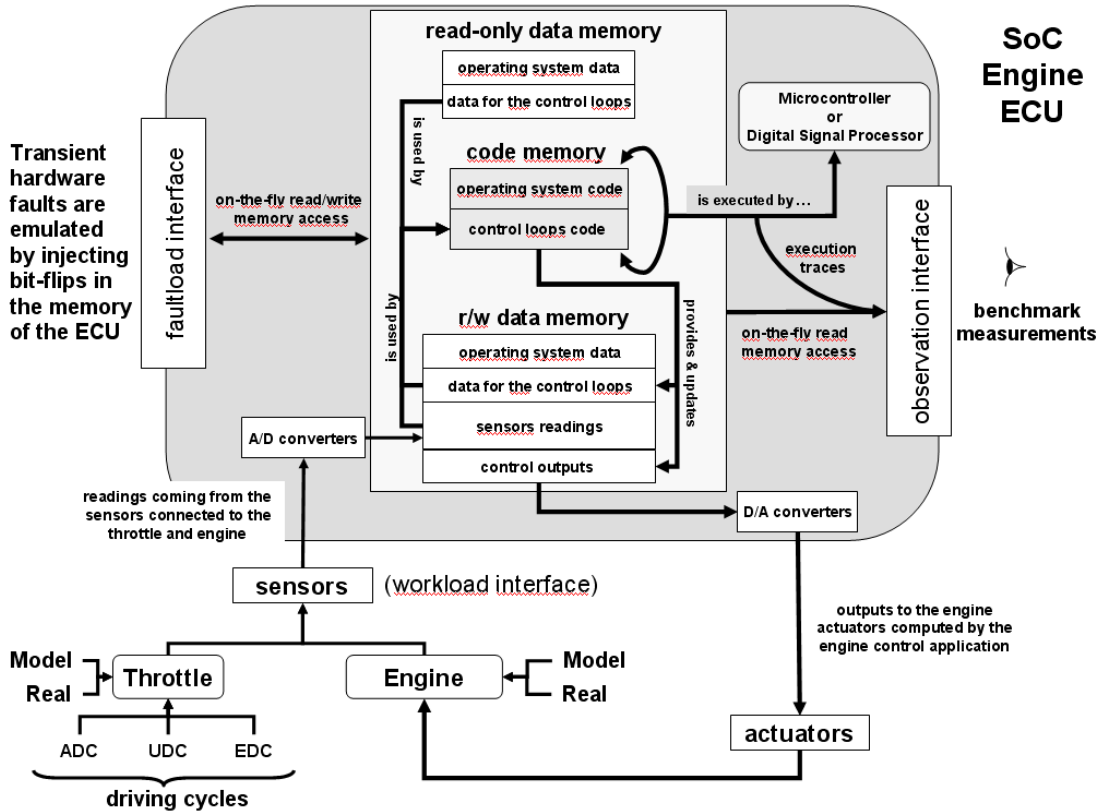


Figure 4-6. Role of the benchmarking interfaces in the experiments

On the other hand, executing the faultload means injecting faults in the normal execution of the engine ECU. In our case, only one fault is injected per experiment. However, flipping bits in the memory of the engine ECU is more problematic that it may seem at a single glance. Ideally, the fault injection procedure should not interfere with the normal execution of the engine control software. In order to cope with that goal, this procedure must be defined taking into account that: (i) the benchmark target has a real-time behaviour that should not be perturbed during the fault injection process, and (ii) if a real engine is used for the experiments, then its dynamics prevents the execution of the ECU software to be stopped. These issues are general and they concern all modern engine ECUs. Basically, a faultload interface suitable for our purposes must provide on-the-fly read/write memory access features. Using these features, the ECU memory content can be accessed and manipulated at runtime minimizing the level of intrusion and without stopping the normal execution of the system. It is worth noting that on-the-fly memory access are nowadays more and more present in automotive systems where this feature is exploited for calibration purposes.

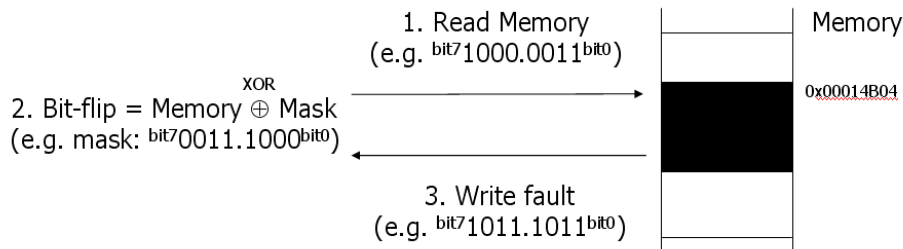


Figure 4-7. Using the faultload interface features for flipping bits in the ECU memory

Figure 4-7 describes how the on-the-fly memory access features of the faultload interface must be used in order to provoke the flip of a certain number of bits in a particular ECU memory position. First, the memory position content is read (by the faultload controller). This controller defines a mask where the bit (or set of bits) to be changed are set to the logical value 1, the rest becomes 0. In the case of a single bit-flip only one bit of the mask is set to 1; in case of a multiple bit-flip several bits in the mask take that logical value. The particular bit (or set of bits) to be set can be predefined or chosen randomly. Once the mask is defined, it is applied on the memory content retrieved in step 1. The logical operation used is an XOR. Finally, the resulting word is written back to the memory, which concludes the fault injection process. The moment at which this fault injection process is triggered (the *fault injection time*) is randomly selected within the duration of a benchmark experiment. As stated before, this time must be greater than the SUB initialization time. It is important to notice that the memory position where the fault is injected to can be selected randomly or predefined by the benchmark performed. More details about this issue will be provided in Section 4.4.2.2.

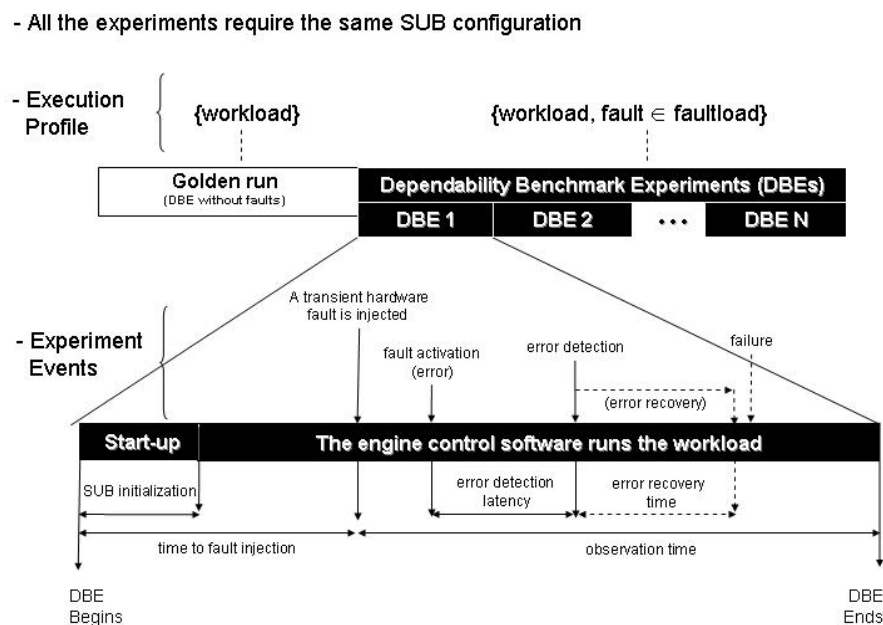


Figure 4-8. Detailed view of the benchmark conduct

If all works fine, the sequence of events depicted in Figure 4-8 must internally occur in the benchmark target. Once one fault is injected in the memory of the engine ECU, it remains dormant until the application exercises the location where it is stored. At this moment (notion of *fault activation time*), the fault is activated and it becomes an error. Then, this error can be detected or not. If it is the case, then the error recovery mechanisms execute and two new measurements can be defined: (i) the error detection latency, which is the difference between the error detection time and the fault activation time; and (ii) the recovery time, which can be defined as the execution time of the system error recovery mechanisms. If an error remains undetected, the activity of the target application must be monitored during the defined observation time. In our particular case, this time must be greater than the time required for running each loop of the engine control software. The observation of these events let to determine whether an error has been detected or not and, in case of error detection, the type



of the detected error. This issue is essential in order to determine when the engine ECU will be reset. As stated in section 4.3.2, only certain error leads the ECU to reset. Most of times, this reset only affects the ECU internal hardware and it cannot be perceived from outside of this unit without being intrusive. If the error finally affects the outputs computed by the engine control algorithms, then a failure raises in the system.

### **Phase 3: Retrieving measurements from the benchmark target**

This phase executes in parallel with phase 2. It starts once the system under benchmarking is running the execution profile.

Through the observation interface, the SUB monitor should have access to the control outputs computed by the ECU internal algorithms. These outputs can be captured internally (by reading their associated memory position) or externally (by reading the analogical signals produced by the digital-analogical converters of the ECU). It is worth noting that this second option requires the analogical signals produced by the ECU to be converted to digital signals by the SUB monitor. In addition, this solution is only viable when every computed control signal is externalized by the engine ECU. If this is not the case, then control outputs must be captured internally which requires the engine ECU to provide on-the-fly read memory access facilities in order to minimize the level of intrusion in the target system.

Due to the real-time nature of the application under benchmarking, being able to obtain the instant of occurrence of a failure is also very important. This temporal information is required for deducing whether a deadline is missed or not. In order to understand how missing deadlines are detected, the reader should understand that in our context, each control loop running inside the engine ECU is executed cyclically with a given frequency. The inverse of this frequency, typically expressed in milliseconds, is the computation period of each control loop. This period represents the amount of time each control loop has to compute new control outputs according to the current set of inputs. If this computation exceeds this amount of time, then the control loop violates its deadline, i.e. the control output is not supplied within the bound of the computation period. In order to detect such situations, it is essential to accurately know the times at which the engine control loops start, finish, resume and stop their execution.

The reader should notice that the internal information depicted in Figure 4-8 is also of great interest from a validation viewpoint. One should not forget that once implemented the behaviour of each benchmark prototype must be always verified according to its specification. This issue has an extreme importance for placing a justified confidence on a particular implementation of the prototype and, by extension, on the dependability measures it provides. One of the objectives of this validation phase must be to check that the benchmark procedure is correctly conducted by the experiment manager. We want to underline that the interest of these measurements is not only limited to the validation of a particular implementation of our benchmark specification. The dependability measures that can be deduced from them, such as error detection latencies and error coverage, can be exploited for guiding the eventual design decisions that each engine ECU developer may adopt in order to improve the dependability measures of their implementations. These tuning

decisions may be oriented, for instance, to the reduction of error detection latencies or to the improvement of error detection coverage.

As Figure 4-6 shows, retrieving all the observations defined here requires the SUB to provide interfaces with features for (i) for tracing the internal activity of the engine control application and (ii) for handling the ECU memory contents on-the-fly. It must be noted that these mechanisms refer to features existing in (most sophisticated) today's debugging standards for automotive embedded systems. This is why their existence is, from our viewpoint, an acceptable requirement for the considered SUB. The benchmark prototype section illustrates how to use one of these debugging interfaces, the one standardised as IEEE-ISTO 5001 [IEEE-ISTO 5001™ 1999], in order to cope with the observation requirements described here.

#### 4.4. Benchmark Prototype

The prototype implementation presented here shows the feasibility of the dependability benchmark specified in the previous section. This prototype enables the comparison of different SoC-based diesel engine ECUs. In addition, it illustrates how the in-built debugging features of a SoC-based ECU can be exploited to facilitate the implementation of the experimental procedure imposed by the benchmark.

Section 4.4.1 introduces the functional specification of the considered diesel engine ECUs. Then, Section 4.4.2 describes the benchmark prototype in detail. Finally, Section 4.4.3 estimates the effort and cost required for its implementation.

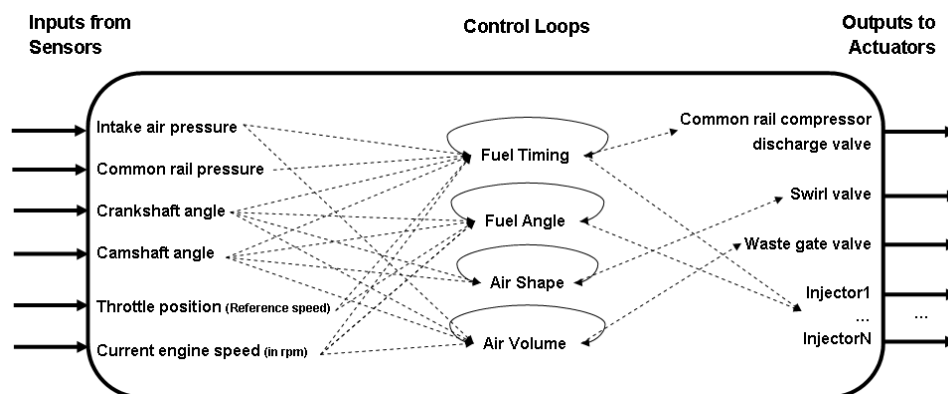


Figure 4-9. View of the case study Diesel Engine ECU

##### 4.4.1. Specification of the case study DBT

Figure 4-9 provides a view of the type of diesel engine ECUs that we consider. These ECUs work in three successive phases:

1. The ECU gets from the sensors six different types of information: (1) the pressure of the air in the engine intake; (2) the pressure of the fuel in the engine common rail; (3 and 4) the current angle of the crankshaft and the camshaft (see these components in Figure 4-1); (5) the current position of the throttle, which defines the engine speed

reference, and (6) the current speed deployed by the engine, which is expressed in rpm. These analogical values are then digitalized and stored in the internal memory of the ECU (as shown in Figure 4-6).

2. According to the sensor readings, the ECU control loops computes: (i) the new pressure in the common rail, which impacts the diesel injection timing; (ii) the outputs required for controlling the swirl and waste gate valves that respectively regulates the air shape flow and volume injected in the engine; and (iii) the duration and the angle of the fuel injection, which are control outputs directly applied on the engine diesel injectors. These control loops are executed cyclically with the following frequency: the fuel timing control loop executes each 20 milliseconds, the fuel angle loop each 50 milliseconds, the air flow shape loop each 20 milliseconds and the air volume loop each 500 milliseconds. Figure 4-9 details the set of inputs that each one of these control loops requires and the set of outputs it generates.
3. Once computed, the control outputs are stored in specific memory locations monitored by a set of D/A converters. These converters are those providing to the actuators the analogical signals they need for regulating the engine behaviour.

#### 4.4.2. Prototype description

Figure 4-10 provides a general view of the prototype benchmarking platform. An Athlon XP 2600 PC manage the execution of the experiments. The considered diesel ECUs run in a microcontroller-based MPC565 evaluation board. The MPC565 microcontroller embeds a built-in debugging circuitry offering a standard debugging interface, formally named IEEE-ISTO 5001 but typically called Nexus. The board provides a port to which a Nexus emulator can be connected. In our case, this emulator is commercial and it is distributed by Lauterbach GmbH. The PC manages the Nexus interface of the microcontroller by means of this emulator.

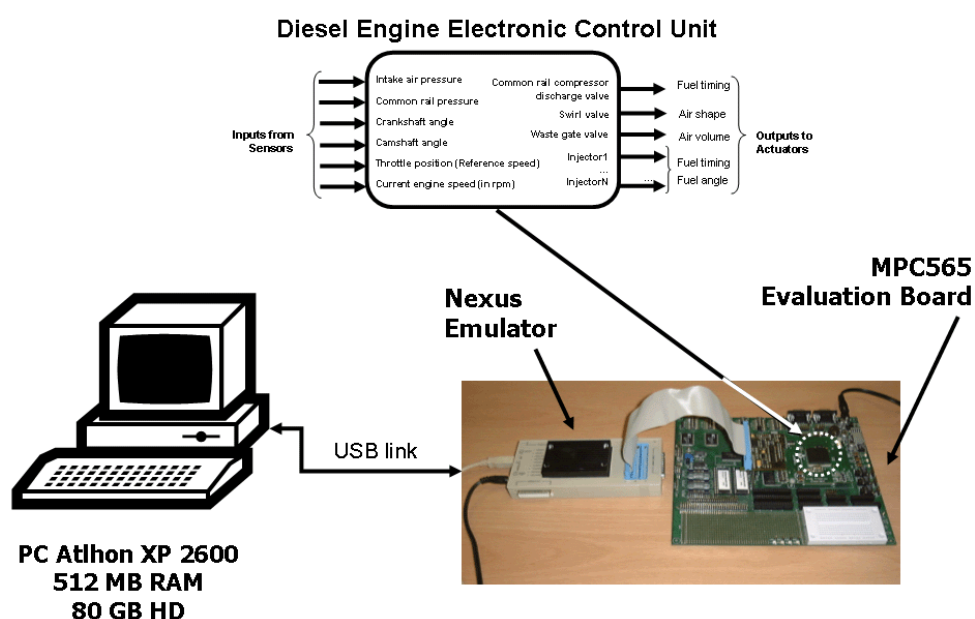


Figure 4-10. Experimental setup

The following sections detail the different elements of the prototype from the perspective of a reader interested in implementing the benchmark specified in the previous section.

#### 4.4.2.1. Workload Implementation

The prototype is designed in order to admit synthetic workloads. These workloads are defined using engine and throttle models.

The definition of engine models are supported by the tracing tool showed in Figure 4-11. Basically, this tool is able to monitor the behaviour of any diesel engine providing the sensor data defined in Figure 4-9. The values showed in Figure 4-11 are, for instance, those supplied by the dynamics of a PSA DW12 engine, an engine with common rail direct diesel injection, also known as HDI, when exercised using the acceleration-deceleration driving cycle. The information monitored by the tool is stored in tables that are also allocated in the memory of the ECU. These tables play the role of the memory locations where the analogue-digital converters store the data coming from the sensors connected to the diesel engine (see Figure 4-6). In that way, the ECU is not aware about the absence of the real engine during the experiments.

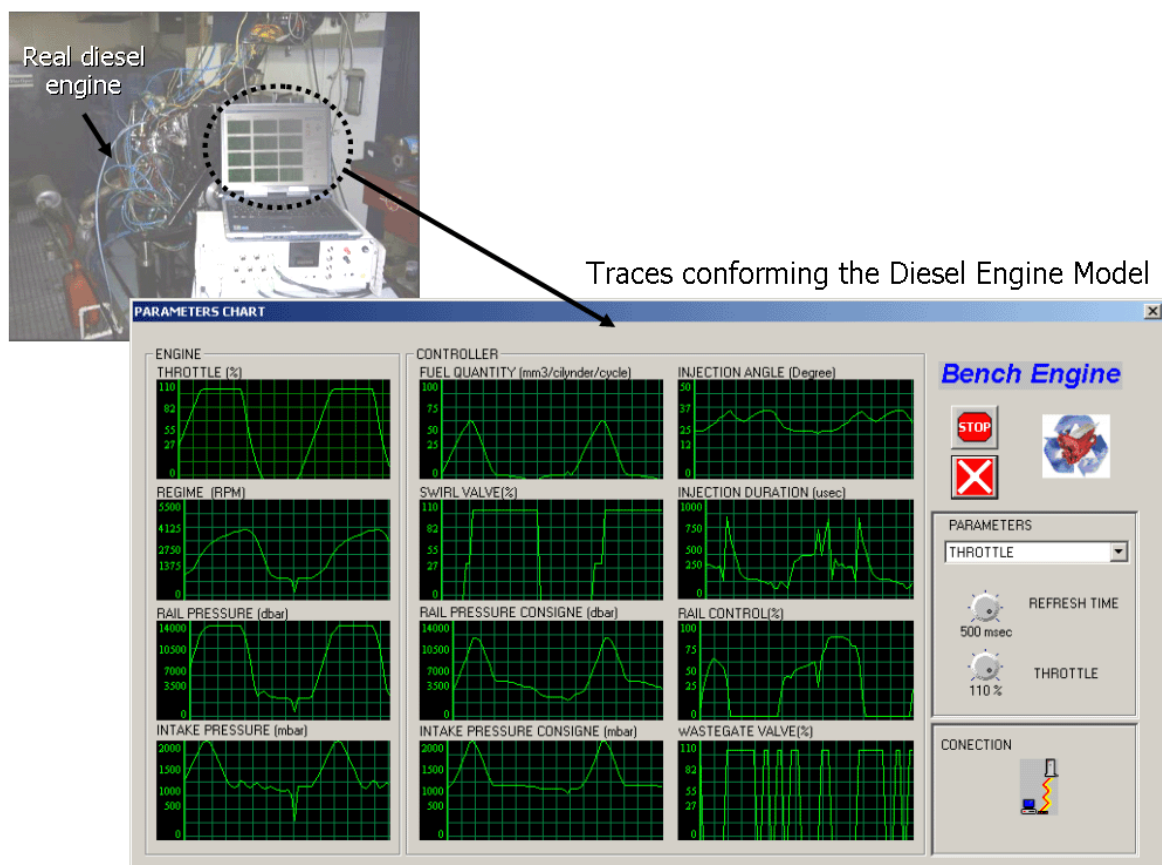


Figure 4-11. Diesel engine modelling tool

The same approach is used in order to replace the real throttle with a model of its behaviour. In this latter case, throttle models are directly deduced from the specification of the driving cycles introduced in section 4.3.3.1.

#### 4.4.2.2. Faultload Specification Tool

In order to support the definition of the faultload, the benchmark prototype provides the tool depicted in Figure 4-12. Basically, it is a graphic tool that aims at the specification of (i) the total number of faults to be injected (one per experiment), (ii) the observation time and (iii) the set of memory locations where the faultload is applied to.

Each memory location corresponds to the set of bytes associated to each symbol of the program. This symbolic information is generated by the compiler when building the executable file of the engine ECU control. In Figure 4-12, the supplied file is called ECU.map. For the sake of clarity, the benchmark user can customize the view of the program symbols. Three possibilities are offered. In the sort by sections view, symbols are sorted according to the memory segment name in which they are allocated. In the *sort by file source* view, they are grouped according to the file in the source code in which they are defined. In the *sort by type* view, symbols are shown according to the type of memory segment (code, read-only data and read/write data) in which they are allocated. Once a view selected, the benchmark user can select one, several or all the memory locations showed by the tool. It is also possible to ponder the importance of each memory location for the fault injection selection process.

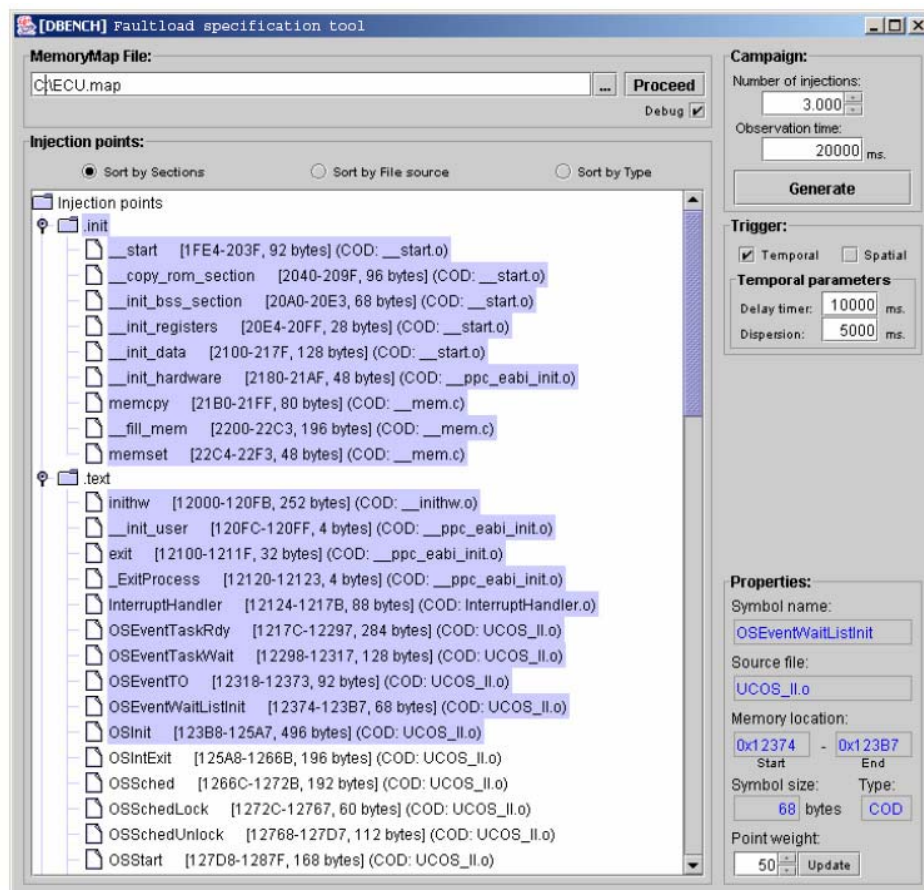


Figure 4-12. Faultload specification tool

The button *Proceed* launches the generation of the faultload. By default, faults are randomly distributed in memory following a uniform probability distribution.

#### **4.4.2.3. Benchmark Procedure Implementation**

Nexus is the corner stone of the implementation of our benchmark procedure. First, Nexus is a debug and calibration standard for real-time embedded systems. Second, the inspection and control features offered by this standard match the requirements identified in the section 4.3.4.2. And third, Nexus is defined in a processor independent way. Thus, solutions built on it can be easily ported and applied to any embedded platform supporting the interface.

In our case, the capabilities of Nexus are exploited in order to inject faults in the memory of the ECU while minimizing the interference with its execution. According to Figure 4-5, the faultload interface of our prototype corresponds to the compound formed by the Nexus emulator, the evaluation board and the MPC565 microcontroller built-in Nexus debugging circuitry. First, a timer is programmed with the fault injection time at the beginning of each benchmark experiment. When the timer expires, a signal is sent, by the Nexus circuitry of the MPC565, to the PC through the Nexus emulator. The on-the-fly memory facilities of Nexus enable the PC to read, without stopping the execution of the ECU software, the contents of the memory position where the fault must be injected. Next, it flips the selected bit and finally it writes back the corrupted contents. The limitation of this approach is that it cannot guarantee that, in all the cases, the steps described before are executed without a simultaneous access of the ECU software to the considered memory location. In other words, we cannot ensure that the memory location contains the injected fault just after the execution of the fault injection process. We solve this problem by not considering for the computation of the benchmark measures those experiments where this situation happens.

In addition, the Nexus circuitry can generate, when programmed adequately, messages that inform the emulator about the data and code memory locations accessed by the ECU software during its execution. These tracing capabilities together with the ones providing on-the-fly read access to the engine ECU memory defines the observation interface of our prototype. The reader must notice that the amount of trace memory of the emulator is limited (in our case to 16 mega-messages). This can become a limitation for the duration of the experiment in case of very long workloads, which is not the case in the driving cycles proposed here.

The interested reader can find in [Yuste et al. 2003a; Yuste et al. 2003b; Yuste et al. 2003c] more details about how to use the Nexus debugging capacities for fault injection and observation purposes.

#### **4.4.2.4. From Experimental Measurements to Dependability Measures**

Deducing the dependability measures from the measurements resulting from the experiments is far from being trivial. First, the trace files of these experiments are difficult to handle due to their sizes. In our case, the size of some files exceeds 1 gigabyte. Second, the information stored in each file is a dump of the emulator tracing message memory. Thus, the format of this raw information is not directly exploitable for our benchmarking (comparison) purposes.

Figure 4-13 depicts the three-step process our data analyser follows in order to deduce the expected dependability measures from the measurements retrieved by the SUB monitor. These measurements are located in the experiment repository, in our case, a file directory. First, the measurements of each experiment are classified according to the type of probe

through which they have been obtained. Then, this *probe-oriented* representation is compared to another one deduced from the golden run. This comparison enables the identification of the relevant events happened during each the experiment. Once this *event-oriented* representation computed for all the experiments, the general measures defined in Table 4-1 are deduced. It is worth noting that, in addition to these general measures, the analysis tool also provides some specific measures regarding error latencies, error coverage and error distribution. As commented in section 4.3.4.2, these specific measures can be exploited with validation and tuning purposes.

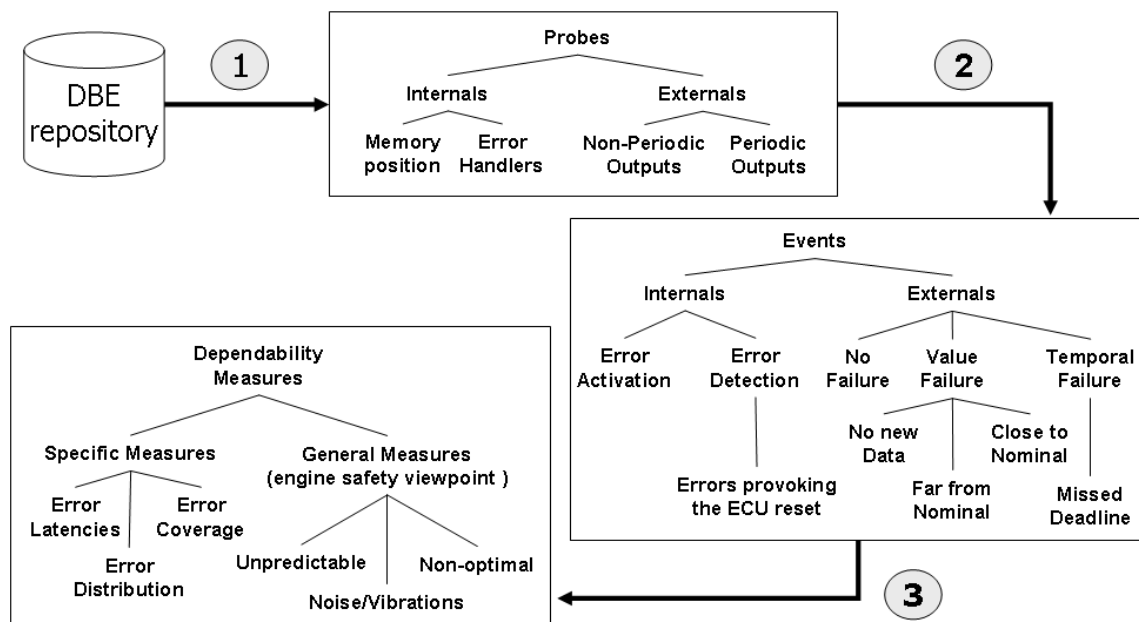


Figure 4-13. Experimental data analysis: From measurements to measures

### 4.4.3. Benchmark experiments

The following sections describe the configuration considered for the experiments carried out with our benchmark prototype. Then, we present and reason about the results obtained from those experiments.

#### *Benchmark targets*

Two different versions of the Diesel ECU (*DECU* for short) described in section 4.4.1 are considered in our experiments:

1. The first version runs on top of a real time operating system, called  $\mu\text{C}/\text{OS-II}$  [Labrosse 2002]. In the rest of this chapter, this *DECU* will be referred as *DECU with RTOS*. Each control loop of this *DECU* is implemented as a different operating system task, the activity of each task is synchronized with the rest by means of semaphores and wait system calls. On the other hand, the operating system is configured in order to schedule the execution of the tasks following a rate-monotonic policy.

2. The second version of our case study DECU is directly implemented on top of the MPC565 microcontroller, so no operating system is available in that case. We will name this second DECU as *DECU without RTOS*. In its implementation, each control loop is deployed in a different program procedure and the main program schedules the execution of each procedure using a scheduling policy computed off-line.

### ***Execution Profile***

For the experimentation purposes, both the urban and extra-urban driving cycles has been considered as workload. The throttle model for these cycles was deduced taking into account the acceleration/deceleration information provided in Figure 4-3 and the dynamics imposed by the considered engine, which was a PSA DW12 engine. The model of this engine was obtained from measures performed using our engine modelling tool (see Figure 4-11).

As far as the faultload is concerned, transient hardware faults has been randomly injected in the memory of the ECU engine using a single bit flip fault model. These faults are selected using our faultload specification tool (see Figure 4-12) and the procedure followed for their injection was the one described in section 4.4.2.3.

### ***Results***

We have devoted one working week (5 days) to the execution of the benchmark experiments. During this time, 600 experiments were carried out: 300 with each driving cycle. 200 seconds were required for running each urban driving experiment, while the double (400 seconds) was required for the extra-urban experiments. The analysis of the raw day obtained from the experiments, although automatically performed, took 16 hours. All the process was automatically performed. The results included in Table 4-2 and Table 4-3 show the measures obtained from these experiments.


Under urban conditions, the DECU without RTOS is definitively better than the version integrating the operating system. First, the DECU with RTOS doubles the failure ratio of the version without RTOS. Second, the percentage of unpredictable failures that the former DECU has shown is 4 times greater than the one showed by the latter DECU. Following the same tendency, the percentages of noise and vibrations and non-optimal failure modes are twice in the DECU with RTOS than in the without operating system. These results do not leave any doubt about which is the most suitable DECU version for engines used in urban conditions. This DECU is the one without RTOS.

Under extra-urban driving conditions, both DECUs have shown similar failure ratios. The study of the failure distribution has shown that, in the case of the DECU version with RTOS, most of the failures (the ratio is 2.78 %) have an unpredictable impact over the engine. In the case of the DECU without RTOS, this ratio is of 1.28 %. However, the ratio of failures producing noise and vibrations is of 1.60 %, while it is null in the case of the version with operating system. As far as the engine non-optimal behaviour is concerned, both ECUs have shown similar failure ratios. The conclusion is that the DECU without RTOS is also better for driving under extra-urban driving conditions. In that case, this does not mean that it has a lower failure ratio than the version with RTOS, it means that the failures are distributed in such a way that their impact is more safe from the viewpoint of the considered engine.



**Table 4-2. Results obtained using the Urban Driving Cycle as workload**

		Diesel ECU with RTOS			
		Fuel injection		Air management	
		Angle	Timing	Volume	Flow shape
Number of DBEs: 300 Failure ratio: 5,76%					
Control failure modes	Engine ECU reset (ECU internal registers corrupted)	1,15%			
	No new data	0,2%	0,77%	0,57%	0%
	Value Failures				
	Close to nominal	0,38%	1,53%	0%	0%
	Far from nominal	0%	0,38%	0%	0,38%
	Time Failure (Missed deadline)	0%	0%	0%	0,4%

Engine Behaviour:  unpredictable behaviour (most critical)  
 noise & vibrations  
 non-optimal (less critical)

		Diesel ECU without RTOS			
		Fuel injection		Air management	
		Angle	Timing	Volume	Flow shape
Number of DBEs: 300 Failure ratio: 2,38%					
Control failure modes	Engine ECU reset (ECU internal registers corrupted)	0,34%			
	No new data	0%	0%	0%	0%
	Value Failures				
	Close to nominal	0%	0,34%	0%	0%
	Far from nominal	0%	0,68%	0,68%	0%
	Time Failure (Missed deadline)	0%	0,34%	0%	0%

#### 4.4.4. Implementation Cost and Effort

The economical cost related to the hardware depicted in Figure 4-10 was of 18000 €. The time required for developing our prototype was of 120 man working days (6 man working months). Basically, ten man working days has been spent in the development of the tools showed in Figure 4-11 and Figure 4-12; fifty more days in order to implement the benchmark procedure using the MPC565 Nexus capabilities; and finally, the automation of the result analysis process has required an additional effort of 60 man working days. It must be noted that this effort estimation includes both the time required for understanding the specification and the time devoted to get skills in the programming of the selected hardware target platform. Our experience shows that programming languages selected for the implementation of the benchmark must be, if possible, function of the programming skills of the developer. Otherwise, if the programming languages are imposed, for instance due to the enterprise practices, then an additional effort must be included to this temporal estimation.

We have also estimated the amount of time required for sequentially running 3000 experiments with each (urban and extra-urban) driving cycle on the considered targets. The time needed such experiments is of one working month and a half in total. Although this time can be significantly reduced when running the experiments in parallel, this increases the economical investment to be performed on the benchmarking platform. Other solution is to reduce the number of experiments, bounding the experimental time to one working week. This is what we have made in the experiments that we have performed. However, one must take into account the impact that this reduction in the number of experiments may have in the accuracy of the measures finally obtained. This interesting issue is a subject for future research.

**Table 4-3. Results obtained using the Extra-Urban Driving Cycle as workload**

		DECU with RTOS			
		Fuel injection		Air management	
		Angle	Timing	Volume	Flow shape
Number of DBEs: 300 Failure ratio: 5,1%		0,68%			
<b>Control failure modes</b>	<i>Engine ECU reset</i> (ECU internal registers corrupted)	0,68%			
	<i>No new data</i>	0%	0,34%	0%	0%
	<b>Value Failures</b> <i>Close to nominal</i>	0%	0%	0%	0%
	<i>Far from nominal</i>	2,04%	2,04%	0%	0%
	<b>Time Failure</b> (Missed deadline)	0%	0%	0%	0%

Engine Behaviour:  unpredictable behaviour (most critical)  
 noise & vibrations  
 non-optimal (less critical)

		DECU without RTOS			
		Fuel injection		Air management	
		Angle	Timing	Volume	Flow shape
Number of DBEs: 300 Failure ratio: 5,76%		0%			
<b>Control failure modes</b>	<i>Engine ECU reset</i> (ECU internal registers corrupted)	0%			
	<i>No new data</i>	0,32%	1,6%	0%	0%
	<b>Value Failures</b> <i>Close to nominal</i>	0%	1,28%	0%	0%
	<i>Far from nominal</i>	0,96%	1,28%	0	0,32%
	<b>Time Failure</b> (Missed deadline)	0%	0%	0%	0%

## 4.5. Benchmark Properties and Their Validation

This section focuses on the verification of five properties of our dependability benchmark: its portability, its non-intrusiveness, its scalability, the repeatability of the experiments and the representativeness of the dependability measures they provide.

### 4.5.1. Portability

The portability of a benchmark specification is the property that refers to its ability to apply to different benchmark targets. Our specification builds on a general engine model that abstracts from the specificities associated to a particular type of engine. Thus, the control loops included in this model are those at the core of diesel and gasoline engine ECUs. From that viewpoint, the specification is applicable to both types of engines. However, one should not forget that each type of engine may have specific control features that have not been considered in our benchmark. In order to study the dependability of these control loops, the approach needs to be extended. This extension restricts the portability of the specification to engines of the same type – if we consider, for instance, specific diesel or gasoline control features, like the exhaust gas recirculation or the spark control loops. The same applies if we consider control features of a particular (diesel or gasoline) engine model.

From the viewpoint of the deployed prototype, its implementation can be plugged on any engine ECU providing a Nexus debugging interface. The only requirement is that the interface must be Nexus class 3 compliant. This issue has not been underlined before because it only influences the portability of the benchmark prototype. The fact is that engine ECUs with Nexus interfaces of classes 1 and 2 do supply neither on-the-fly memory access nor memory tracing messages facilities. Since these are the features sustaining the execution of our prototype, it cannot be ported to those SoC engine ECUs not providing Nexus class 3 compliant ports. This is the only impairment to the portability of our benchmark prototype.

Our experiments show that, as far as the functional interface of benchmark target remains unchanged, different implementations of the same specification can be benchmarked using the same benchmark prototype. This is like in the operating systems domain, when one talks about POSIX and forgets the particular operating system (Windows or Linux) implementing this portable interface. When the interface of the benchmark target is modified (an input/output is added or removed from its specification), the prototype must be then changed in consequence. The experience shows that these (apparently naïf) changes are easy to take into account in the experimental procedure, but they deeply affect the analyzer of the experiment results. Hence, the software architecture of this analyzer must be designed open enough for enabling such kind of extensions.

### 4.5.2. Non-intrusiveness

As we have already underlined in other sections, the strong encapsulation imposed by the SoC technologies to the manufacturing of today's engine ECUs makes difficult the benchmarking of the software running inside them without modifying the code of the SUB or affecting its temporal behaviour.

As commented in Section 4.3, we have chosen the solution of exploiting, with benchmarking purposes, the debugging features provided by most of today's automotive ECUs. These features are those described in section 4.3.4. In order to assess the viability of the approach, we have developed a prototype, whose implementation relies on the class 3 Nexus debugging interfaces. According to its specification, the built-in circuitry embedded in this interface must ensure, by construction, an absence of temporal intrusion when (i) accessing the engine ECU memory contents on-the-fly, and (ii) tracing the activity deployed by ECU software. Since these features are also enough from an observation viewpoint, no code instrumentation is required in order to adapt the code of the ECU software to the purposes of our benchmark. Hence, no spatial intrusion is either required.

This practical work has shown that our specification can be implemented in a non-intrusive way from either a spatial and temporal viewpoint. It must be noted that Nexus has been chosen due to its increasing popularity in the automotive embedded systems domain. However, it is important to underline that our benchmark specification does not rely on Nexus, only our prototype does. Hence, any other debugging interface providing the features identified in the specification can also be used.

### **4.5.3. Scalability**

What affects the scalability of our benchmark specification is the consideration of additional control loops. In that case, the benchmark must be revisited in order to enrich its specification with the goal of covering also the characterisation of the new features related to these control loops. It is obvious that the benchmark prototypes meeting the old specification must be updated according to the new requirements identified in the resulting benchmark. This is an exercise that has been left for future research.

### **4.5.4. Repeatability of the experiments**

Under identical initial conditions, and using the same execution profile, can be the results obtained from experimentation reproduced? This property must be guaranteed in absence and in presence of faults. In this latter case, the property must be verified despite the occurrence of errors and failures.

Basically, the idea is to run the same experiment several times in order to compare the resulting traces. In our experiments, two traces are said to be equal if they reflect the same ECU internal and external activity. The internal activity refers to the events depicted in Figure 4-8 and the external activity concerns the production of control outputs. This information is retrieved from the system using the tracing capabilities offered by the Nexus interface of the hardware platform. The goal is to check that the traced events are the same and they are ordered following the same temporal sequence. As far as a missed deadline failure does not occur, the exact time at which each event happens is not important for the verification of the repeatability property. The reader must notice at this point that the notion of statistical repeatability used by our partners does not apply in our case. This is mainly motivated by the fact that we do not consider in our benchmark the impact that faults have in the performance of the engine ECU, since this type of measures has no sense in our context. Thus, the measures computed by our benchmark do not admit variations from one experiment

to another, i.e. the set of events traced from one experiment must remain the same if the execution profile, the initial conditions and the target system remains the same.

In the experiments, we have checked the repeatability of (i) the golden runs, (ii) the experiments with a faultload affecting an empty memory location, (iii) the experiments with a faultload impacting occupied code/data memory locations that are used by the engine ECU software. In this latter type of experiments, we differentiate those ones leading to an internal error not provoking the failure of the target system and those leading the target system to fail. These experiments have been very fruitful in the first stages of the benchmark prototype development in order to fix some bugs in the implementation. As far as our benchmark prototype matures, it is harder to identify discrepancies between experiments executed on the same target using the same execution profile and the same initial conditions. For instance, the experiments depicted in section 4.4.3 have been reproduced twice and no deviations have been appreciated in the resulting dependability measures.

Further experiments are however required in order to check the difficulty of guaranteeing this level of repeatability when changing the hardware platform or when considering other benchmark targets.

#### **4.5.5. Representativeness**

The workload we have selected is representative for the automotive domain, as shows the fact that it is used a standard workload for emission certification of light duty vehicles in Europe. It is well-known that the combustion process of fuel inside the engine conditions the level of contaminants that vehicles emit to the atmosphere. Thus, our choice makes sense if the workload is applied to the part of the engine ECU responsible for handling this combustion process. As discussed in section 4.2.2, this is our case.

Concerning the faultload, recent publications [Gil et al. 2002; Gracia et al. 2002] has pointed out the likelihood of apparition of hardware transient faults in integrated circuits will increase greatly. We have selected the bit-flip fault model for characterising such hardware transient faults because it provides, from our viewpoint a good compromise between (i) the type of models widely accepted by the dependability community for modelling such type of faults and (ii) the type of perturbations that can be injected in SoC electronic components used in today's automotive systems.

The question that for the time being remains open is the representativeness and the usefulness for the industrials of the dependability measures defined in our benchmark. This is an issue we plan to handle in a near future.

## **4.6. Conclusions**

The omnipresence of COTS in the automotive software industry together with the increasing need of minimizing costs and time-to-market, justifies the interest of the benchmarking approach proposed in this chapter. This approach focuses on the study of the impact of engine control application failures over the most critical part of a vehicle: its engine.

The main contribution of this research is to (i) specify a set of dependability measures for comparing the dependability of fuel-based engine control systems and (ii) identify all the elements required for developing a benchmark prototype for obtaining those measures. The high scale of integration of today's engine ECUs is, without any doubt, the most important technological impairment for the development of such type of benchmarks. On the one hand, one must solve the problems regarding the application of the faultload to the engine control applications running inside the ECU. On the other hand, the control outputs and the internal ECU activity must be observed in order to compute the dependability results and validate the benchmarking platform. The prototype section of this chapter has illustrated how to solve in a portable and non-intrusive manner all these problems. The idea is to use the observability (tracing features) and controllability (on-the-fly memory access) features provided by modern automotive electronic control units. In these units, these features are typically used with debugging and calibration purposes. Our proposal is to exploit them for running the faultload and tracing the engine ECU activity without inducing neither temporal nor spatial intrusion in the target system.

The feasibility of the approach has been illustrated through a real-life benchmark prototype, which requires the target system to provide a class 3 Nexus debugging standard port. This standard has been chosen since it is, to the best of our knowledge, the only one having a processor independent specification that guarantees, by definition, a sufficient degree of observability and controllability without interfering with the target system. These characteristics are essential for coping with the ambitious goals of our benchmark specification.

We are convinced that the technology exposed in this chapter constitutes a step forward to the provision of useful methodologies and tools for comparing the dependability of embedded automotive systems in general, and engine control applications in particular. The technology is currently mature enough to be migrated to the industry where it can enrich the existing engine control software development process. This is one of the directions of our current work. The other direction is further research-oriented and it focuses on extending our benchmark approach in order to characterize the dependability of more sophisticated (distributed) electronic control components that need to cooperate for managing mechanical elements that are de-localized in a vehicle. This is, for instance, the case of the antilock braking control systems.