

Technical overview of the expanded powertrain challenge problem

Mobies PI meeting, July 24-26, 2002, New York

Tunc Simsek, simsek@eecs.berkeley.edu

1. Introduction

During the Mobies PI meeting, July 16-18, 2001, Jackson Hole, the Electronic Throttle Control (ETC) problem was developed as an introductory powertrain challenge problem. This problem has served mainly as a workhorse to understand applications of Phase I tools. In particular, Paul Griffiths provided a baseline realization of the ETC specifications and Phase I participants provided analysis and alternate routes to this baseline tool-chain. A detailed description of the baseline solution may be found in Paul's MS thesis [1]. Reviews of the Phase I applications are discussed during the Mobies PI meeting, July 24-26, 2002, New York.

While the ETC problem revealed several interesting aspects of the model-to-software tool-chain automation objective, the simplistic nature of the problem omits the discussion of important software questions relevant to powertrain control. In this paper we discuss these problems and introduce an expanded powertrain challenge problem. Our hope is that this problem brings us closer to a full-blown powertrain scenario while still keeping it simple enough to be addressed by the Phase I participants.

1.1. Why is the ETC problem simple?

Consider a simple description of the engine. The engine produces torque in proportion to the amount of charge in the cylinders during combustion. The throttle opening determines the volume of air and hence the volume of charge that enters the cylinder. The fueling system is responsible for spraying the right amount of fuel so that immediately before combustion the air-to-fuel ratio (AFR) is in 14.69:1 to 14.71:1.¹ That is, there are two basic components to the engine: the ETC which regulates the throttle and hence the volume of charge, and the AFR which ensures that the charge in the cylinder has the desired properties (it is assumed that the spark advance and other engine functions are given and invariant).

The ETC operates in response to the human driver. That is, the driver issues a desired throttle opening (and hence torque) by depressing the accelerator pedal, and the ETC system issues actuator signals to produce the desired throttle signal (note on other characteristics). Furthermore, the manifold acts as a low-pass filter for the airflow dynamics to the intake ports. Effectively, this decouples the airflow dynamics around the throttle from the airflow dynamics around the intake ports. Thus, the performance requirements of the throttle may be specified independent of other engine units and with respect to the perception of the driver. If, for example, the ETC system is too slow, the actual throttle will not track the drivers desired throttle fast enough so that the driver feels like the vehicle is not responding to his/her commands.

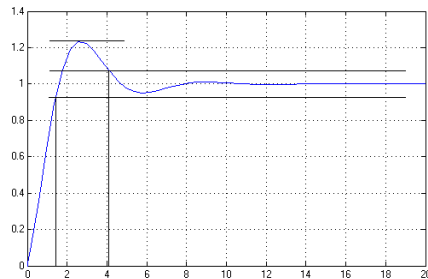


Figure 1: Sketch of a generic 2nd order system tracking a step input. The lines show the overshoot (12.3%), rise time (1.6s) and settling time (4s).

¹ The number 14.7:1 is the ideal air-to-fuel ratio for complete combustion. It is called the stoichiometric ratio.

Denote the drivers desired throttle as $\alpha_d(t)$ and the actual throttle controlled by the ETC as $\alpha(t)$. Then the performance criteria are given with respect to a step change in $\alpha_d(t)$ as illustrated in the Figure 1. The simple (almost linear) dynamics of the physical throttle body makes it possible to realize these specs using a uniformly sampled discrete-time sliding mode controller. That is, a controller that acquires input data, performs a control computation and issues a new actuator signal only on a periodic time interval in the order of a few milliseconds. The human driver is not sensitive to the timing interval of the ETC controller. He/she will only feel the difference if the performance criteria are not met.

1.2. What makes the AFR controller different?

The engine, unlike the human driver, is quite sensitive to the timing of the AFR controller. In particular, the AFR controller, has to issue updates to the fueling actuators (ie. The fuel-injectors) so that the right amount of fuel is sprayed to the intake ports once the valves are closed and with enough slack so that the mixture forms before the valves open. Since the valve timing is determined by the engine speed, the AFR controller will have to issue updates at aperiodic time intervals.

2. The AFR controller timing

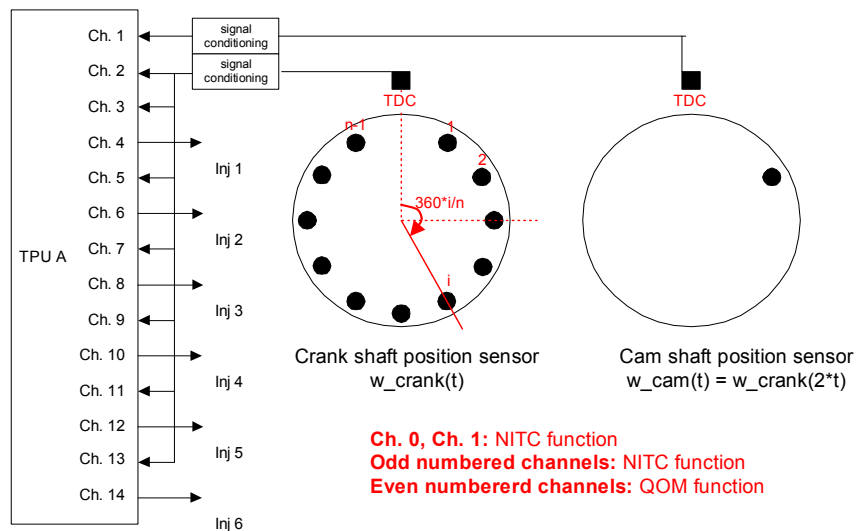


Figure 2: AFR controller hardware

The AFR controller outputs a quantity called the dM_{fc} . This is the rate of the fuel mass. The actuation software is responsible for driving the fuel-injectors so that the amount of fuel sprayed to the intake ports is a prescribed scale of dM_{fc} . The fuel-injectors are on-off actuated. The injectors will spray fuel at a constant rate while a high voltage (or logical 1) is applied.² The MPC555 is equipped with a sub-processor called the Time-processor Unit (TPU). The TPU is a special purpose micro controller that is capable of producing and capturing high resolution digital signals. We will use the TPU to realize the actuation software. Figure 2 illustrates the TPU configuration used by the AFR controller.

We will discuss the operation of a single injector. For each engine cycle (which corresponds to two revolutions of the crank shaft and a single revolution of the cam), the injector sprays once. Initially we assume that the engine is at top dead center (as illustrated in Figure 2). We assume that at this point in time the AFR controller has already computed dM_{fc} – ie. how much fuel is needed for the cylinder in the next engine cycle -- and also knows when the valves of the cylinder are open and closed.

² For the purposes of this discussion we omit the injector response time and other non-linear effects.

The function of the TPU is to issue the rising edge of the injector fueling pulse (relative to an absolute position of the crank and cam shafts) and the length of this pulse (see Figure 3). The camshaft pulse is used as a reference to obtain the numbering of the crankshaft teeth (which we have chosen to be 5 + 1 missing) as shown in Figure 3. Once the crankshaft teeth have been converted to pulses via the signal conditioning circuitry and the desired teeth number have been obtained, the fueling algorithm for the i^{th} injector follows:

- 1) when the crank shaft reaches the end of the engine cycle, denoted by tooth number 11, the TPU initiates the serviceFuelParams routine which asks the controller for the fueling parameters for the next engine cycle,
- 2) the AFR controller is assumed to have already computed or will compute these parameters before the crank shaft reaches tooth number 1,
- 3) shortly before tooth number 1, the TPU checks the parameters for the i^{th} injector and schedules the i^{th} injector pulse,
- 4) the i^{th} injector pulse schedule is given by the rising edge of the pulse and the falling edge of the pulse. The rising edge of the pulse is given by the α_i parameter for the i^{th} injector and the falling edge by the rising edge plus an absolute time T_i (assumed proportional to the mass of fuel). The schedule is such that the rising and falling edge fall all happen while the valves for the i^{th} cylinder are closed.

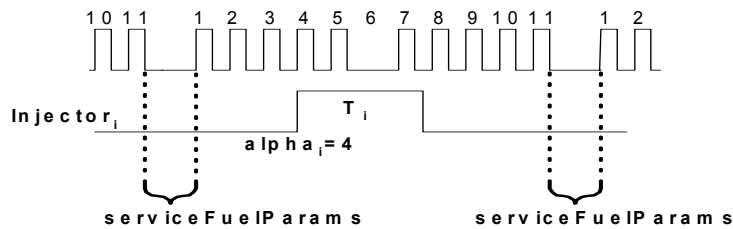


Figure 3: sample timing for the i^{th} fuel-injector actuation pulse

2.1. Acquiring engine speed

Production engines are not equipped with an engine speed sensor. The AFR controller, however, relies on the engine speed to compute the fueling parameters. The crank-shaft signal is used to obtain an approximation of the engine speed. In the ideal case, the controller would ask for the engine speed at some time t and the software would respond at time $t+dt$ with the desired quantity where dt is small. In general, the time-delay dt is given by the software execution time (assumed negligible) and the sensor acquisition time. For the ETC problem, all the sensory signals had a small and negligible acquisition time. The acquisition time of the engine speed is however not negligible. The roughest approximation would be obtained by measuring the time-interval l between m consecutive teeth on the crank-shaft (suppose for simplicity that there is no missing tooth). Then, $w_{\text{approx}}(t) = m360/nt$ (degrees/second) where n is the number of crank teeth taken to be 6 in the example of Figure 3 and t is the time at which the measurement is taken. At an engine speed of 1000 rpm, with $n=6$ and $m=1$ this corresponds to an acquisition delay of roughly 10 ms.

3. Baseline solution for the ETC/AFR control problem

By a baseline solution we mean a reference implementation of the ETC and AFR controllers on the MPC555 target. As our starting point we have chosen the TargetLink code generation review for the observer based AFR controller described in [2]. We make use of the code generation methodology described in this report together with a stylized methodology for mapping Simulink subsystems to OSEK tasks. The end result is an almost automated process for realizing the ETC/AFR controller implementation problem. The automation aims to start with the ETC/AFR models and produce target code for the controllers.

3.1. Baseline ETC/AFR models

Jason Souder prepared a simplified Simulink/Stateflow model for the ETC/AFR problem. This model was prepared in two steps. In the first, the complete powertrain models were stripped down to include only the AFR controller and related systems. In the second, this stripped model was merged with the ETC models used in the ETC challenge problem.

Stripping down the powertrain models is motivated by at least three factors : reducing the possibility of bugs, increasing simulation speed, and most importantly, reducing the number of state variables.. We know that model checking and verification tools have the state explosion problem. That is, the number of states explored by the verification tools is exponential in the number of original system states. Thus, by keeping the number of states in the ETC/AFR models to a minimum we hope to provide a feasible challenge problem for the Phase I analysis tools. The actual components that were removed from the original powertrain models include:

- transmission is locked in 4th gear, so no transmission models,
- torque converter is locked,
- exhaust-gas recirculation is removed,
- coolant temperature dynamics is removed,
- the idle-air and transmission controllers are removed.

These simplifications require the implicit assumptions that the engine is warm and running in 4th gear.

The merging of the ETC and the stripped down AFR controller produced several interesting challenges regarding variable naming, unit conversions, library links, duplicate blocks, solver parameters and initialization specifications. Jason's notes on his experience are included in the appendix. The joint model at this stage is still a mean-value model. This means that the AFR controller output (denoted by dmfc) has the units of *fuel mass/time*. We desire that the controller produce *fuel mass/engine stroke*.

Tunc Simsek added a crank/cam sensor to the model and an aperiodically triggered controller called the serviceFuelParams. This control algorithm is input the dmfc and produces 6 outputs. These outputs are the fuel-injector parameters in the unit *fuel mass/engine stroke*.

3.2. Baseline controller structure

Figure 4 illustrates the 3 basic elements of the controller software. These are the *scheduler*, the *ISR scheduler* and the *controller software*. The controller software is further illustrated in Figure 5.

The ISR³ scheduler is responsible for counting the crank and cam signals and issuing an *asynchronous* trigger at a reference crank and cam position. In the example discussed above, the reference position was the last falling edge of every other crank revolution cycle. The scheduler is responsible for triggering the controller subsystems periodically and in such a way to ensure the intended data flow order.

The controller software carries slightly deeper meaning. The data lines represent data stores. The data store denoted by dmfc is special since it lies at the boundary of a periodic and asynchronous task. We do remark, however, that in the simulation semantics an explicit data store is not needed since the subsystems require zero execution times. The data store denoted by mfc_per_stroke is also special since it denotes a particular hardware register.

The *etc_v2_0* and *afr_v1_0* subsystems represent the cluster of tasks that perform the ETC and AFR functions respectively. Finally, the *sfp_v1_0* subsystem represents an interrupt service routine that performs the serviceFuelParams functions described above.

³ ISR is an acronym for Interrupt Service Routine.

3.3. Baseline software methodology

The baseline software for the ETC/AFR controllers uses a stylistic method to map the controller components to OSEK objects and the methodology presented in [2] to obtain the functional code for these objects. As discussed above there are 3 basic controller components: scheduler, ISR scheduler and controller software. The mapping of these components to OSEK objects is discussed here.

For simplicity of the presentation we will map all the data stores (ie. All the wires in the controller software) to global variables. Due to the asynchronous access, the dmfc datastore is mapped to a shared OSEK resource. The mfc_per_stroke datastore is mapped to a prescribed memory location; in particular, to the parameter RAM of the fuel-injector TPU drivers. We will not comment on the shared resource arbitration for the TPU parameters.

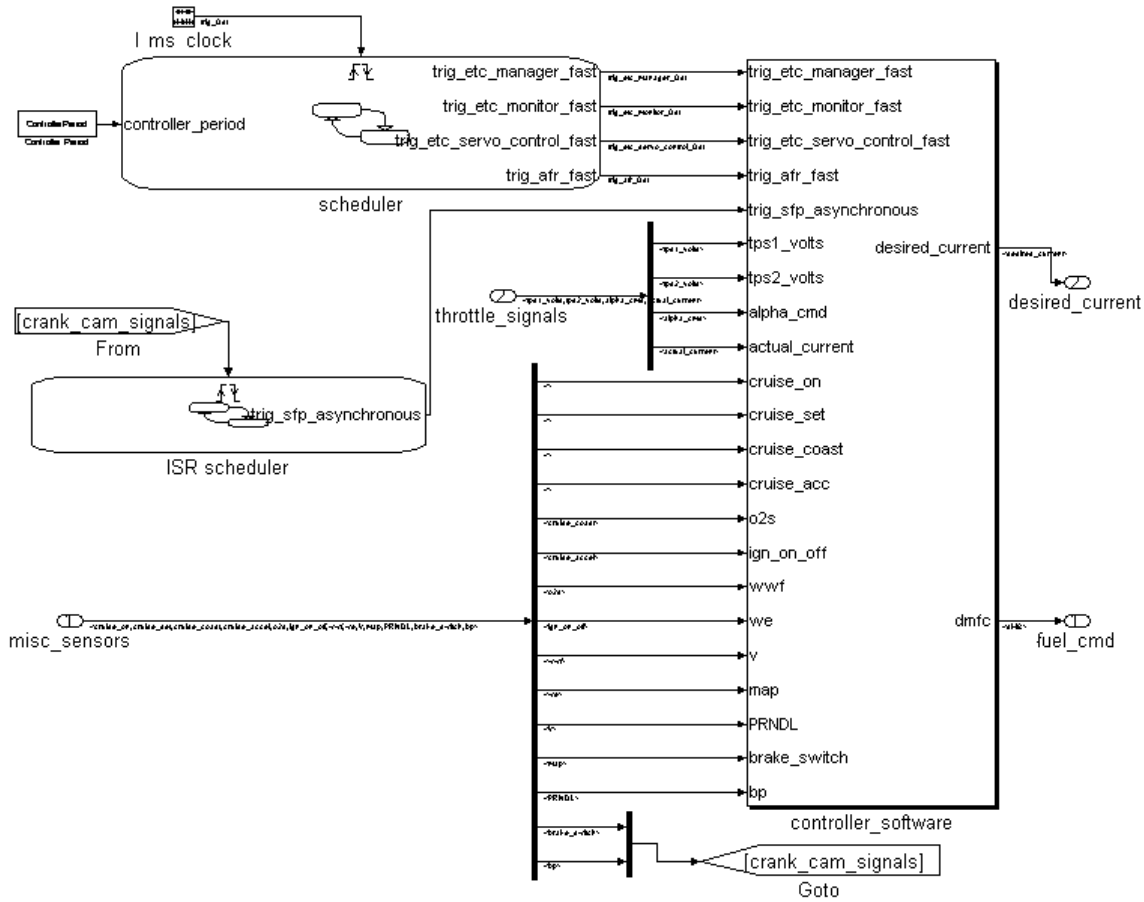


Figure 4: Top level controller software structure

The rough idea is to map each controller software subsystem to an extended OSEK task. The first input of these subsystems that represents the triggering is mapped to an OSEK event. The task will then wait for that event in an endless while loop and upon receiving the event, will produce an output. For example, the afr_v1_0 is mapped to an OSEK task that looks like:

```
TASK(afr_v1_0) {
    while(1) {
        WaitEvent(trigger_afr_fast);
        ClearEvent(trigger_afr_fast);
        AcquireInputs_afr_v1_0();
        GetResource(dmfc_lock);
        OutputUpdate_afr_v1_0();
        ReleaseResource(dmfc_lock);
        SetEvent(afr_v1_0_states, stateUpdate);
    }
}
```

```

TASK(afr_v1_0_states) {
    StateUpdate_afr_v1_0();
    TerminateTask();
}

```

The `afr_v2_0` task waits for the event denoted by `trig_afr_fast`. Upon receiving this event it immediately acquires its inputs⁴ and performs the output update functions. It is assumed that these two computations take very little time. To ensure that the outputs are produced without delay, this task is made non-preemptible. Thus, it is important that the input acquisition and the output updates do not take a long time. Finally, a trigger is sent to an auxiliary preemptible task that performs the state updates. The priority of the auxiliary task need not be high but it is assumed that it should complete the state update phase by the time `afr_v2_0` loops again.

The code generation methodology described in [2] is used to automatically produce the output update and state update code for these tasks. The noticeable changes to the referenced methodology includes: use of global variables to represent inputs and outputs, the separation of state and output update functions, the separation of input acquisition functions and use of floating point instead of fixed point computations.

The scheduler component is mapped to a periodically triggered OSEK task. The task is triggered with period 1ms, and is responsible for triggering (or setting the event of) the scheduled controller task. The code for the scheduler looks like:

```

TASK(scheduler) {
    static TaskType nextTask;
    static EventType nextEvent;

    SetEvent(nextTask,nextEvent);
    nextEvent=scheduleNextEvent(...);
    nextTask=scheduleNextTask(...);
    SetRelAlarm(triggerScheduler,1,0);
    TerminateTask();
}

```

The actual code for the functions that compute the next event and task is obtained by invoking `TargetLink` on the scheduler Statechart. The alarm denoted by `triggerScheduler` expires in 1 ms and initiates the scheduler task. This task should be non-preemptible.

Finally, the ISR scheduler component models the interrupt request produced by the TPU (see Figure 3). The TPU should request an interrupt at the falling edge of the last crank tooth transition (at the end of every other crank-shaft revolution). Currently, we do not know how to automatically generate microcode for the TPU. This component is handcoded to the following OSEK objects:

- the actual TPU microcode, the discussion of which is out of the scope of this paper,
- the TPU interrupt request configuration,
- an ISR routine that triggers the `sfp_v1_0` task.

The TPU interrupt request configuration is performed in the OIL file by adding an ISR to the external interrupts and denoting this ISR by `ISRscheduler`. The OSEK code for the ISR looks something like:

```

ISR(ISRscheduler) {
    SetEvent(sfp_v1_0, trig_sfp_asynchronous);
}

```

where the structure of `sfp_v1_0` looks like:

⁴ The inputs are always acquired first. This is to ensure that the inputs are acquired at the sampling times $kT+\Delta$ with the minimum jitter Δ . In comparison, the RTW semantics may delay the input acquisition if there is no direct feedthrough connection. Also note that the acquisition delay of the engine speed, w_e , may not be negligible.

```

TASK(sfp_v1_0) {
  while(1) {
    WaitEvent(trigger_sfp_asynchronous);
    ClearEvent(trigger_sfp_asynchronous);
    GetResource(dmfc_lock);
    AcquireInputs_sfp_v1_0();
    ReleaseResource(dmfc_lock);
    OutputUpdate_sfp_v1_0();
    SetEvent(sfp_v1_0_states, stateUpdate);
  }
}

TASK(sfp_v1_0_states) {
  StateUpdate_sfp_v1_0();
  TerminateTask();
}

```

We note that the ISR described above must be an OSEK level 2 or level 3 ISR. This requirement is imposed because the ISR uses the OSEK system call SetEvent. We also note that one could have directly embedded the code for the sfp_v1_0 task into the ISR, but this would complicate the methodology of our approach.

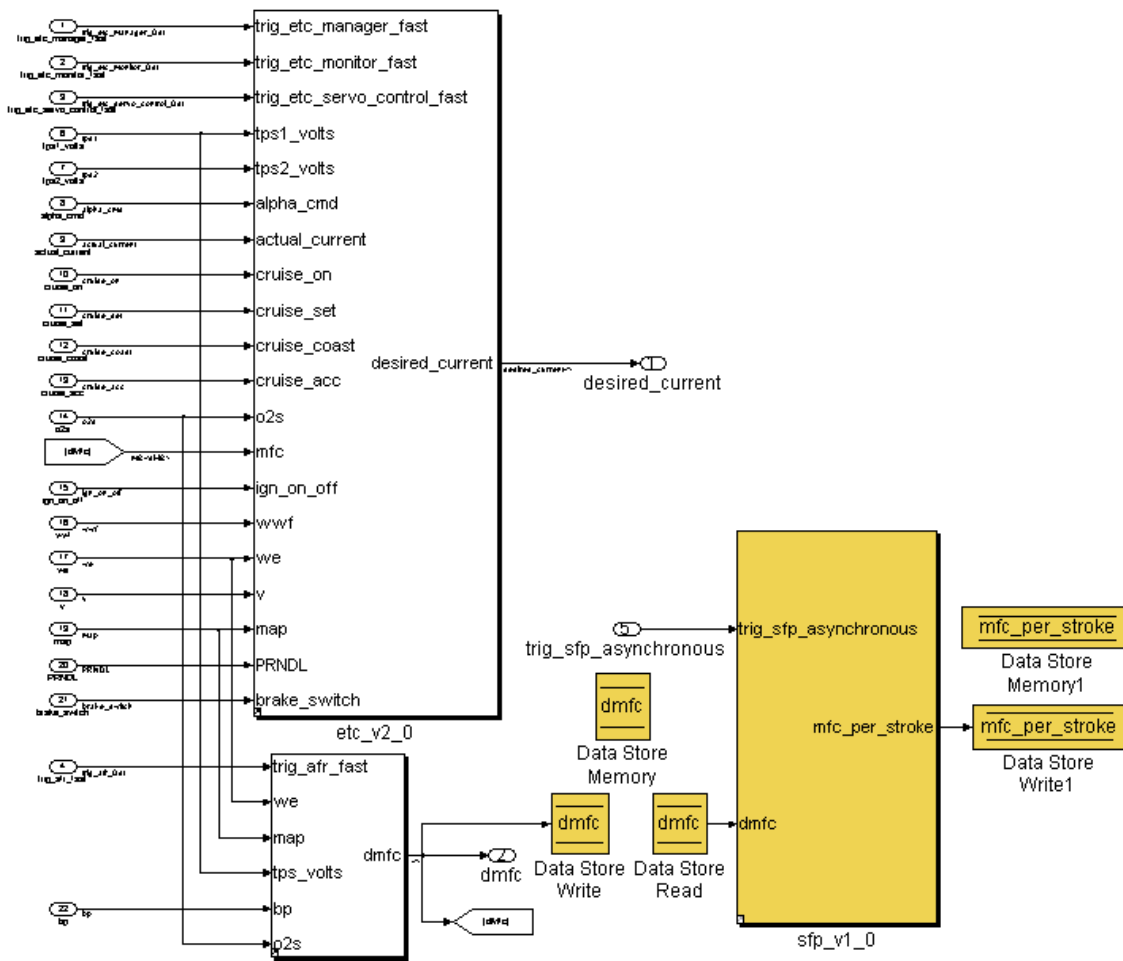


Figure 5: Controller software (asynchronous components shown in yellow)

4. Conclusion

Currently, we have available the joint ETC/AFR models implemented in Simulink/Stateflow (see Section 3.1), a methodology for mapping model components to OSEK objects (see Sections 3.2 and 3.3) and the TPU micro code for managing crank/cam signals and producing fueling pulses (see Section 2). We are

envisioning a completely automated tool chain for model analysis, for cross-compiling the models to target code and for analysis and verification of the target system.

The ETC/AFR control problem both builds on and adds new challenges to the ETC problem. This paper is intended to serve as both a design document and to provide technical grounds to discuss specific challenge problems, [3]. In particular, we hope that a MoBIES toolchain is discussed before proceeding to realize baseline solutions for the new powertrain challenge problems.

References

1. Paul Griffiths, MS Thesis, UC Berkeley, 2002.
2. Bostic, D., Milam, W., Wang, Y., Cook, J. *SmartVehicle Baseline Report: Embedded Software Analysis and Generation*, Ford Motor Company, May 7, 2002.
3. Milam, W. *Suggested Challenge Problems*, version 0.0, July 2002.