Implementation of AFR Controller in an Event-driven Real-time Language

Arkadeb Ghosal, J. Carlos Zavala J., Marco A. A. Sanvido, J. Karl Hedrick

Abstract— **The control of emissions has been addressed in the past to comply with environmental regulations. In particular air-to-fuel ratio control is key to reach the allowed pollution levels. The aim of this work is to present an alternative approach which allows for more flexibility to account for the type of signals and requirements of automotive applications, specifically, handling of time and event triggered tasks. An Air-Fuel Ratio nonlinear controller is developed for an automobile engine. The controller is then implemented using the event-driven real-time programming language xGiotto on the OSEK platform provided by WindRiver. Special attention is given to show the advantage that can be gained from using an event driven paradigm for implementing automotive controllers.**

Keywords-- AFR control, non-linear control, emissions control, real-time programming language

I. INTRODUCTION

The regulations of automotive emissions have progressively become more strict over the last years, resulting in a decrease of the amount of pollutants created by new vehicles. Increasing demands on minimization of hydrocarbons require improvement of the technologies to reduce pollution and of the strategies used to control these technologies.

In particular, accurate engine air-fuel ratio control is crucial to reach the standards of automotive engine emissions. Currently, engine controllers attempt to eliminate hydrocarbon (HC) emissions by regulation of the air-to-fuel ratio (AFR) of the engine. A catalytic converter is used in most modern passenger vehicles to treat unburned gases in conjunction with the AFR controller. When there is an excursion of the air to fuel ratio, the deviation of the mix with respect to the stoichiometric level results in a temporary unbalance of oxygen that can be used to counteract the next opposite oscillation of the air-fuel ratio.

Many AFR controllers are capable of providing good tracking of the desired air-fuel ratio under steady conditions, however, under transients the performance is not guaranteed. Even though the catalytic converter operates to soften the variations in the AFR, there are two circumstances in which it is convenient to have a more precise control. One is related to large variations in the speed of the engine, which would not allow the catalytic converter to recover and stabilize the AFR to the desired reference. The other is the necessity of having a tighter control of the AFR when the engine is still cold. In the last case the oxygen storage capability of the catalyst does not allow for the correction of the variations of the AFR since the catalyst is not fully operating at these conditions.

The use of xGiotto with non-linear sliding control is presented in this paper with the objective to improve the performance of AFR controller during transients. The value in the use of this approach is the possibility of using an event-triggered frame offered by xGiotto for a problem that also is expressed in terms of events. Specifically, the signals produced by the crankshaft depend on the angular speed of the engine, rather than on a fixed period. The control action will be established in two levels, the first one is the calculation of the control law as a periodic task, whereas the other is the update of engine parameters every camshaft cycle as part of an adaptive control strategy. This division gives the possibility of handling tasks with different timing requirements in similar manner. On the other hand, the adaptive control, which is operating only on the parameters of the controller, can be executed according to events such as the camshaft cycles.

Overview. The next section describes an air-fuel ratio controller of an automobile engine. Section III describes the original implementation of the controller in OSEK for a Ford Taurus test-bench. Section IV briefs the programming paradigm, syntax and semantics of xGiotto language in the

Financial support from Conacyt, the Department of Mechanical Engineering and the Department of Electrical Engineering and Computer Sciences of UCB made possible this work.

Arkadeb Ghosal is a graduate student at University of California, Berkeley, CA 94020. (phone: 510-642-4552; fax: 510-642-5745; e-mail: arkadeb@eecs.berkeley.edu).

Jose Carlos Zavala Jurado is a graduate student at University of California, Berkeley, CA 94720. (phone: 510-642-6933; e-mail: czavala@ me.berkeley.edu).

Marco A.A. Sanvido was a postdoctoral researcher at University of California, Berkeley, CA 94025. (phone: 510-643-5212 ; fax: 510-642-5745; e-mail: msanvido@eecs.berkeley.edu).

J. Karl Hedrick is a professor at the Mechanical Engineering Department of the University of California, Berkeley, CA 94720. (phone: 510-642-2482; e-mail: khedrick@ me.berkeley.edu)

0-7803-9098-9/05/$25.00 ©2005 AACC

context of the AFR controller. Section V discusses the integration of the xGiotto run-time system with OSEKWorks. Section VI presents the results and observations of the experiment. Section VII concludes the paper.

II. AFR CONTROLLER

A. The engine

The main objective of an AFR controller is to maximize the performance of the corresponding catalytic converter by maintaining a precise ratio between the air and fuel. An oxygen sensor is used in closed loop feedback to measure the presence of air in the exhaust gas. Its function is to provide an indication of how close to stoichiometric level is the mix of air and fuel. Binary oxygen sensors are commonly used in automotive and output high or low voltage when the air-to-fuel ratio is above or below stoichiometric, respectively.

The controller design presented here is based on a mean value model used by Souder [1] and adapted from Aquino [2] and Moskwa [3]. This model includes states for the air flow, the fuel flow and the rotational inertia of the engine. Check Hedrick and Cho [4] for reference of the engine and powertrain model .

The first state of the model is the mass air flow through the intake manifold. It can be defined as the difference between the mass flow rate into and out of the manifold

$$\dot{m}_a = \dot{m}_{ai} - \dot{m}_{ao} \quad (1)$$

The mass flow rate running into the manifold is calculated as the maximum flow rate multiplied by throttle angle and pressure ratio scaling factors

$$\dot{m}_{ai} = \dot{m}_{ai,max} \cdot TC(\alpha) \cdot PRI(P_m, P_a) \quad (2)$$

The mass flow rate out of the manifold is a function of engine displacement, V_e, intake manifold volume, V_m, intake manifold pressure, P_m, mass of air in the manifold, m_a, engine speed, ω_e, and volumetric efficiency, η_v and is given by

$$\dot{m}_{ao} = \frac{V_e \eta_v (P_m, w_e) m_a w_e}{4\pi V_m} \quad (3)$$

A 2-D look-up table with inputs ω_e and P_m is used for the volumetric efficiency.

The fueling dynamics are considered as well. As the fuel is injected into the intake ports, part of the fuel vaporizes and part of it deposits on the intake manifold as a liquid. The fuel deposited on the intake manifold wall will affect the in-cylinder air-fuel ratio as it becomes part of the air stream. A model of these dynamics is given by

$$\begin{aligned} \dot{m}_{fo} &= \dot{m}_{fv} + \dot{m}_{ff} \\ \tau_f \ddot{m}_{ff} + \dot{m}_{ff} &= (1-\varepsilon)\dot{m}_{fc} \\ \dot{m}_{fv} &= \varepsilon \dot{m}_{fc} \end{aligned} \quad (4)$$

ε represents the portion of the fuel that enters the cylinder directly as vapor, $1-\varepsilon$ is the portion of fuel that is deposited on the manifold walls, $\dot{m}_{fc}$ is the commanded fuel mass flow rate, $\dot{m}_{fv}$ is the mass flow rate of fuel entering the cylinder directly as vapor, and $\dot{m}_{ff}$ is the mass flow rate of fuel entering the cylinder from the fuel puddle on the manifold walls. The time constant τ_f is modeled as a constant here for simplicity, however adaptive non-linear control techniques can be used to obtain a good approximation.

The equations in (4) can be combined into a single equation for use in the fuel controller design

$$\ddot{m}_{fo} + \frac{1}{\tau_f}\dot{m}_{fo} = \varepsilon \ddot{m}_{fc} + \frac{1}{\tau_f}\dot{m}_{fc} \quad (5)$$

The final state equation of the engine model incorporates the engine rotational dynamics

B. Air to Fuel Control Model

A sliding surface is needed to design a sliding mode AFR controller. Since the goal is to reduce the error between the actual air-fuel ratio and the desired air-fuel ratio to zero, a sliding surface, S, is defined as the difference between the actual and desired air-fuel ratios $S = \frac{\dot{m}_{ao,actual}}{\dot{m}_{fo,actual}} - \frac{\dot{m}_{ao,desired}}{\dot{m}_{fo,desired}}$. As the air-fuel ratio converges to the desired value, S converges to zero. Using β as a constant desired air-fuel ratio to express the desired sliding surface definition gives:

$$S = \dot{m}_{ao} - \beta \dot{m}_{fo} \quad (6)$$

This surface is used to design the sliding mode controller.

Using the sliding surface above, define a positive definite Lyapunov function candidate as

$$V = \frac{1}{2}S^2 \quad (7)$$

Differentiating the Lyapunov function candidate, V, gives

$$\dot{V} = S\dot{S} \quad (8)$$

If $\dot{V}$ is negative definite, global asymptotic stability is guaranteed via Lyapunov's Direct Method, since V(x) is radially unbounded, or V(x) $\rightarrow \infty$ as $\|x\| \rightarrow \infty$. Differentiating the sliding surface and using the fueling model gives

$$\dot{S} = \ddot{m}_{ao} - \beta\varepsilon\ddot{m}_{fc} - \frac{\beta}{\tau_f}\dot{m}_{fc} + \frac{\beta}{\tau_f}\dot{m}_{fo} \quad (9)$$

The definition of the sliding surface S is used to replace the unknown quantity $\dot{m}_{fo}$:

$$\dot{S} = \ddot{m}_{ao} - \beta\varepsilon\ddot{m}_{fc} - \frac{\beta}{\tau_f}\dot{m}_{fc} + \frac{1}{\tau_f}\dot{m}_{ao} - \frac{1}{\tau_f}S \quad (10)$$

Choosing the control law to cancel out the known terms and adding an extra term for robustness gives

$$\beta\varepsilon\ddot{m}_{fc} = \ddot{m}_{ao} - \frac{\beta}{\tau_f}\dot{m}_{fc} + \frac{1}{\tau_f}\dot{m}_{ao} + \eta \cdot \mathrm{sgn}(S) \quad (11)$$

Also, it is noted that given the binary nature of the oxygen sensor, its output can be used as follows:

$$\mathrm{sgn}(S) = \mathrm{sgn}(\dot{m}_{ao} - \beta\dot{m}_{fo}) = y \quad (12)$$

where y is the output of the oxygen sensor. This leads to the modified control law

$$\beta\varepsilon\ddot{m}_{fc} = \ddot{m}_{ao} - \frac{\beta}{\tau_f}\dot{m}_{fc} + \frac{1}{\tau_f}\dot{m}_{ao} + \eta \cdot y \quad (13)$$

If uncertain quantities in (13) are present, η can be replaced by K, where $K = \eta + \|\Delta f\|$, and $\|\Delta f\|$ is the norm of the uncertainties. Using the control input in (8) an (10), the result is:

$$\dot{V} = S\dot{S} \le S(-K\,\mathrm{sgn}(S) - \frac{1}{\tau_f}S) < 0 \quad (14)$$

which means $\dot{V}$ is negative definite and the sliding surface S is globally asymptotically stable. This powertrain model was calibrated to represent a 3.0L Ford Taurus engine. The data were provided by Ford.

III. Original Implementation

Nonlinear sliding mode control techniques in the frame of model based embedded system design have been used to successfully derive AFR controllers. In the final report of the Mobies project [5], the authors utilized the same experimental platform to implement AFR control running under an OSEK based operating system.

The powertrain test facility consists of a 1996 Ford Taurus engine (3.0L, 24 Valve, DOHC) and 4-speed automatic transmission. A dynamometer is implemented to control the output of the transmission. Also, to better represent the inertial load of a vehicle, a flywheel is added downstream of the dynamometer. The engine and dynamometer setup are shown in Figure 1.

The target processor used for implementation of the controls is the Motorola MPC555 [6] with an OSEK compliant real time operating system. The engine setup was designed to allow tests to be done on individual subsystems of the engine while the rest of the engine is controlled by the stock control (the production control from the engine manufacturer).

In the conclusions of [5] the authors pointed out the need of a program structure that allows for managing time and events integrally, with capabilities of handling event based sensing and actuation.

Fig. 1. The actual motor set up in the Hesse hall.

IV. xGiotto and AFR Controller

xGiotto [7] is an event-triggered programming language for hard real-time systems. The implementation scheme of an xGiotto program with respect to a real-time operating system (RTOS) has been shown in Figure 2. The xGiotto program and the run-time system fits in between the application and the RTOS (which in turn interacts with the hardware). The control program is written in xGiotto and describes the interaction of events from the environment and tasks (or functionality code).

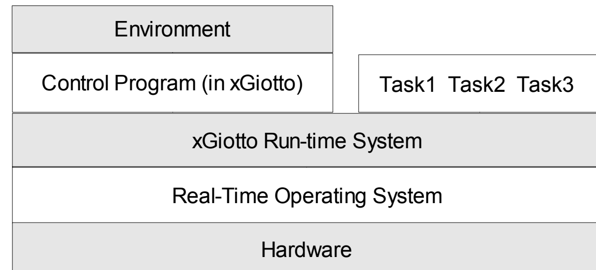

Fig. 2. Implementation Scheme.

The AFR controller is implemented as follows. There are three tasks: *task1* is the controller routine and is invoked every millisecond, *task2* is the injection routine and is invoked every cam shaft cycle and *task3* is the adaptive controller routine (invoked every 10th cam shaft cycle) used to improve the AFR controller. The RTOS used here is

OSEKWorks, an OSEK compliant RTOS from VxWorks. The target platform (the hardware) is the MPC555 controller of the FordTaurus engine discussed in Section III.

A. xGiotto

xGiotto is built around the notion of *Logical Execution Time* (LET) [7] model of task execution. In the LET model, for every task invocation an LET is provided which implies the time when the computation should be available at the output. Even if the output is ready early, they are made visible only when the specified execution time expires. If the computation cannot be completed by the LET then a run-time exception is generated.

```
event
  start, stop, ms, cam

task
 task1 // controller routine released every ms
 task2 // injection routine released every cam cycle
 task3 // adaptive controller release every 10 cam cycle

react R1 {release task1;}
react R2 {release task2;}
react R3 {release task3;}

react time() {
  whenever remember [ms]  react R1 until [ms];
}

react cycle() {
  whenever remember [cam] react R2 until [2ms];
}

react ten_cycle() {
  whenever remember [10cam] react R3 until [2ms];
}

/* main */
{ when [start]
   react time()  until asap [stop] ||
   react cycle() until asap [stop] ||
   react adapt() until asap [stop];
}
```

Fig. 3. AFR controller in xGiotto.

The xGiotto program for the AFR controller is shown in Figure 3. An xGiotto program consists of four types of declarations: *ports*, *events*, *tasks* and *reaction blocks*. *Ports* or program variables are used for inter-task communication and for storing the evaluation of the tasks. The *events* are raised by interrupts. The interrupt may be a time tick (internal to the RTOS-hardware system) or an aperiodic event from the environment. An event is defined by a name associated with an interrupt that raises it. For example, in the program (shown in Figure 3) the event *ms* is raised every millisecond and the event *cam* is raised every cam shaft cycle.

An xGiotto *task* consists of a name, a set of input ports, a set of output ports and a task body. The task body is a standard sequential program (or a link to an actual functionality description) which evaluates a computation on the values of the input ports and updates the output ports at the end of logical execution time. For example, *task1* links to the controller routine.

A *reaction block* is the basic programming block for an xGiotto program. The construct for a reaction block is *react b until e*, where *b* is the body of the reaction block and *e* is an event. The body *b* of the block declares triggers and releases tasks. A trigger maps an event *e* with a reaction block *r* and the construct is *when e react r*; when the event *e* occurs the reaction block *r* is invoked. For example, reaction *cycle* defines a trigger which invokes reaction *R3* at the event *cam* (the repetition construct of *when* is *whenever*). The construct for a task release is *release t (in) (out)* where *t* is the task released, and *in* and *out* are the set of input and output ports respectively. The input ports are read at the time of release and the output ports are updated at the instance of termination. The event *e* denotes the terminating instance of a reaction block; a reaction remains active from the time of invocation till the event instance *e*. When a reaction block is invoked it activates the triggers and releases the tasks. On termination the triggers are made inactive and the tasks are terminated. Thus the active time span of a reaction block denotes the logical execution time for the tasks released. For example, reaction *R2* is invoked at the event *cam* and is terminated at the second instance of *ms* following the *cam* event; the LET for task *task2* is the inter-arrival time which is greater than 1 ms here.

The termination event and the events of the triggers of a reaction blocks defines a *scope*. When a reaction block is invoked the corresponding scope of the called reaction block is made *active* while the scope of the callee reaction block is made *passive*. This is implemented by maintaining a stack of scopes; when a reaction is invoked the corresponding scope is pushed upon the stack and the scope of at the top of the stack is active. The events of an active scope are handled as soon as they arrive. An event in a passive scope can be handled as follows: ignored (which is the default action), remembered or acted upon as soon as possible; they are specified by the keywords: *forget*, *remember* and *asap*. If an event is remembered, the associated action is taken when the corresponding scope becomes active again. If an asap event occurs the trigger queues of the sub-tree rooted by the scope is emptied so new trigger action can take place. Reaction blocks can be invoked in parallel and hence a tree of scopes exist at any instant of execution; the leaf nodes being the active scopes and the non-leaf nodes being the passive scopes.

The program starts by calling three reaction blocks -- *time*, *cycle*, and *adapt* -- in parallel. The controller is started by event *start* and runs until event *stop* is raised. Reaction block *time* defines a trigger that invokes reaction *R1* every *ms* event. Similarly *cycle* and *adapt* define triggers to invoke reaction blocks *R2* and *R3* every event *cam* and ten counts of event *cam*. Reaction blocks R1, R2 and R3 release tasks *task1*, *task2*, and *task3* respectively.

B. *Embedding Constraints and xGiotto Analysis*

The scope structure and the different forms of events allow one to express interaction of events, to add control in an efficient way and to embed time intervals in the program succinctly. For example, the form associated with event *cam* implies that *R2* (and thus task *task2*) would be invoked every *cam* event. The inherent structure of xGiotto allows easy modification of the control structure. For example, consider a parallel controller task needs is to be added to the above program. This can be done by adding a parallel mode in *start* mode or releasing a second task in *R3*. Note this is done without modifying any other functionality or control-flow description. Embedding inter-arrival times can be performed in two ways. First, explicit timer definitions can be linked with events; e.g. *ms* is defined to be triggered every millisecond. Second, assumptions on minimum time available for tasks can be expressed explicitly through scoping e.g. by introducing an empty reaction block parallel to the task scheduled. For example, in the above example the task *task2* gets at least one ms to complete its execution.

xGiotto structure and semantics allows three checks to be performed on any xGiotto program. Most RTOS do not support effective dynamic memory management; however event handling (through tree of scopes) require dynamic data structure. *Resource size prediction* analyses the memory requirements for executing the program on a real-time platform and helps in efficient memory management. Determinacy in the values of program variables is crucial for real-time applications. For xGiotto programs *race condition detection* can be performed by the compiler to check whether a port can be updated non-deterministically during program execution and thus ensuring determinacy of (data dependent) program execution. The time information embedded into the program and the worst case execution time estimation of the tasks on a given platform can be used to perform *schedulability check* which verifies whether all tasks released by the program can complete execution by their respective deadlines (relative to the given platform).

C. *xGiotto Run-Time System*

The xGiotto compiler checks for syntactic conformity, performs the required analyses and generates code for a virtual machine (referred to as Embedded Virtual Machine or EVM). The EVM [8] along with the real-time platform and environment makes the xGiotto run-time system. An implementation of the EVM requires three components: a dynamical data structure to keep track of the scopes (event filter), a processor which computes the reaction to an event (reactive machine) and a scheduler (to handle execution of released tasks).

V. IMPLEMENTATION WITH XGIOTTO

The xGiotto run-time system implementation for the OSEKWorks real-time system is done by generating source files that are then compiled and linked with the OSEK development tools. At present the event filter and the reactive machine have been integrated with the scheduler of the OSEKWorks tool set. The Embedded Machine [9] has been used as the reactive machine; however the original implementation of the Embedded Machine has been modified to incorporate handling of scopes.

The application (ports, events, tasks and reaction block declarations) and the EVM (the integrated event filter and the reactive machine) are passed on as C files. A third C file is used to allocate and manage memory. The task codes are put in C file wrappers and linked with EVM. Finally the application, EVM and the tasks are integrated with OSEK development tools (compiled and linked via an OSEK Implementation Language (OIL) file) and downloaded to the target platform MPC555 board of the Ford Taurus test stand (Figure 1).

A. *Analysis*

The OSEKWorks system does not support any dynamic memory. Unfortunately the dynamical nature of an event-based system needs some sort of dynamical memory. The xGiotto compiler is used to compute resource bounds (upper bound of scopes, and associated trigger queues and task releases) required by a run-time implementation. This information is used to pre-allocate the required memory and assign scopes dynamically without the need of dynamic memory allocation.

The WCET estimation of the tasks are as follows:

	Periodicity (in ms)	Execution Time (in ms)
Task 1 (Controller loop)	1	0.572
Task 2 (Fuel Injection Task)	Between 12 – 24 ms	0.169
Task 3 (Parameter Adaptation Task)	Between (12- 24)*N ms where N is an integer between 1 - 50	0.017

A schedulability check was performed for the controller program (in xGiotto) with the above data and the program was found to be schedulable.

The injection task and parameter adaptation tasks are released every cam cycle and 10 cam cycles. However The minimum arrival time information for two consecutive cam events is required to perform a reasonable schedulability analysis.

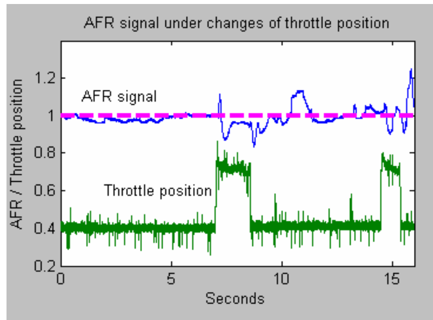

Fig 4. Factory Controller

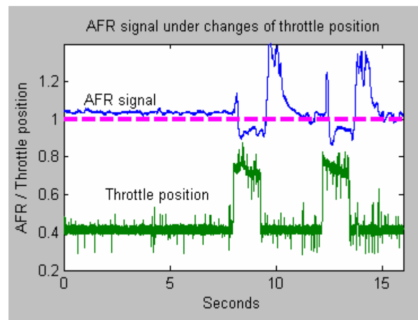

Fig. 5. Standard Non-linear Implementation

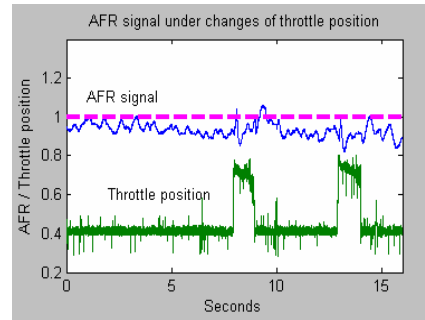

Fig. 6. Implementation with xGiotto

VI. Results And Observations

The objective of this section is to assess the features of the implementation of an AFR controller with respect to traditional approaches.

To have a reference for comparison, the implementation of two other controllers will be shown. The first one is the factory AFR controller, which is operated from the Electronic Control Unit which is part of the original vehicle functionality. The second one is a sliding mode AFR controller without any adaptive control and without extensive parameter tuning.

The third controller that is presented uses an adaptive task to modify the value of a time constant parameter. The adaptive task is executed at a certain number of cam shaft cycles. It is convenient to point out that the all the controllers that are used for comparison use the factory oxygen sensor that is part of the standard control system of the vehicle. A broadband lambda sensor is used for monitoring purposes.

The result of the execution of the production controller is shown in Figure 4. The lower signal (color green) represents the throttle position and its response to a step input, simulating the torque demand of a driver. The same plot shows the behavior of the corresponding AFR signal which is the upper signal (color blue). It is possible to see that changes in the torque demand will cause variations in the AFR signal. The dotted line (color pink) represents the desired stoichiometric AFR level.

Figure 5 shows the performance of the nonlinear sliding mode controller. Non-extensive tuning was provided for the controller. It can be noticed that the optimal AFR value is reached some time after the torque demand stabilizes; however, undesired peaks are present in the signal.

Figure 6 shows the performance of the nonlinear adaptive controller under xGiotto. Small variations are still present in the AFR signal; however a smoother response is obtained with respect to the previous two controllers. The addition of the adaptive control in xGiotto is orthogonal to the AFR implementation which made the integration of the task much easier.

VII. Conclusion

The paper shows how the event-driven paradigm of xGiotto helps in implementing the controller with different tasks being written in parallel, namely the fuel injection task, the control task and the parameter adaptation task. It also shows how it helps in studying engine performance and improving overall user-controller interface.

In future we would like to take the full advantage of the xGiotto programming structure for the controller implementation and check the performance measurement. Besides, the present way of implementation may not be the most optimal approach since it involves integrating the EVM on top of the hosting RTOS. An optimal but complex solution would be to implement the EVM “bare bone" on the target platform, reducing the overhead to the minimum

Acknowledgment

We would like to thank Jason Souder and Dan Lamberson of the Department of Mechanical Engineering, University of California, Berkeley for help and collaboration in this work.

References

[1] Jason Souder. Powertrain Modeling and Nonlinear Fuel Control. Masters Thesis. University of California, Berkeley, 2002.

[2] C. F. Aquino. Transient a/f control characteristics of the 5 liter central fuel injection engine. SAE 810494, 1981.

[3] J. J. Moskwa. Automotive Engine Modeling for Real Time Control. PhD thesis, Massachusetts Institute of Technology, 1988.

[4] D. Cho and J. K Hedrick. Automotive powertrain modeling for control. *Journal of Dynamic Systems, Measurements and Control*, vol. 111:568{576, December 1989.

[5] Mobies final report. University of California, Berkeley, 2003.

[6] MPC555: (http://e-www.motorola.com)

[7] Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A.: *Event-driven programming with logical execution times.* In: Hybrid Systems Computation and Control. (2004).

[8] A. Ghosal, M. A. A. Sanvido, T.A. Henzinger, *A preliminary report on the embedded virtual machine.* Technical Report UCB//CSD-04-1322, University of California, Berkeley (2004).

[9] Henzinger, T.A., Kirsch, C.M.: The Embedded Machine: Predictable, portable real-time code. In: Proceedings of Programming Language Design and Implementation, (2002)