11AE-0082

Platform Independent System Integration

Author, co-author (Do NOT enter this information. It will be pulled from participant tab in MyTechZone)

Affiliation (Do NOT enter this information. It will be pulled from participant tab in MyTechZone)

ABSTRACT

This paper presents and demonstrates a paradigm to implement automotive systems based on their specifications in a manner that is platform independent. The advantage is to have the same software used in simulation as on different types of micro-controller in a vehicle as well as to ease the integration of different systems. The paradigm is to model the system's components as reactive objects and to use the Timber kernel to schedule their (re)actions. The demonstration is done by developing an anti-lock braking system within the simulation software CarSim and Simulink, which is then evaluated on a braking maneuver over a surface with different coefficient of adhesion from side to side (split mu).

INTRODUCTION

A modern car has now a collection of electronic control systems ranging from engine control to electronic stability program, adaptive cruise control and collision avoidance systems. One can easily include electrical parking brakes, which can be activated by a door opening. Each of these systems is very complex and requires much attention to avoid any lurking undesired behavior. Much work is done to verify their individual performances, robustness and failsafe mechanisms. As if this was not difficult enough, these systems must work together in complete harmony without a conductor to lead them. The picture becomes totally overwhelming when one considers that several assortments of systems are offered for a single vehicle model. For example, there might be five different engines to choose from with different gearboxes among other options. The task to certify a vehicle is impressive.

There is also the issue of cost when considering distributed systems versus centralized ones. Having several systems work on different processors that communicate with each other over a network is one option, which can have its advantages as some nodes are optional. This might come at a higher cost than with a centralized processor. However, transferring code for real time systems from one platform to another is quite expensive in time and funds. The issue is also quite interesting when considering developing and testing the automotive system within a vehicle simulation environment. The exact same code should be used within the simulation tool and within the car, because as soon as there is a difference, one should question the validity of the simulation.

To propose a potential solution, we present a standard automotive system with its simplest form of specifications. We will look at two ways to implement the software using different paradigms and demonstrate the second one in a vehicle simulation. As we proceed, we add to the system specification and point to system integration. The system is evaluated in a vehicle simulation. The system's software or source files are available on the Internet to enable others to evaluate the work [1].

The automotive system used here is an anti-lock braking system (ABS) for a passenger car, which is nowadays a standard component. The vehicle simulation environment used is Mechanical Simulation's CarSim, which cooperates with MathWorks' Simulink.

IMPLEMENTING A SYSTEM

SPECIFICATIONS AND HARDWARE DESCRIPTION

An ABS seeks to preserve the lateral force at the interface between the tire and the road while allowing the vehicle to decelerate in the tire's longitudinal direction. The lateral force enables steering on the front wheels and prevents spin out on the rear wheels. In the longitudinal direction, the ABS seeks to keep the coefficient of friction at its maximum between static and dynamic coefficient of friction. The coefficient of friction is often referred as mu or μ. Its peak value is usually found when the slip between the tire and the road is at 20% of the vehicle speed [2].

The ABS must therefore measure the wheel speeds, estimate the vehicle velocity and control the hydraulic pressure at each brake if the slip exceeds the 20% mark. To control the pressure, there is a hydraulic control unit (HCU) which has two 2-way solenoid valves for each brake or channel; a total of eight valves. The HCU can isolate each brake from the master cylinder and thereby hold the pressure constant with the isolation valve. Once isolated from the master cylinder, the brake pressure can be reduced by opening the relief valve, which leads the brake fluid to a low pressure accumulator where a pump forces the fluid back to the master cylinder. To raise the pressure at the wheel, the isolation valve needs to be opened.

The valves and pump are controlled by the electrical control electronic unit (ECU). The inputs of the ECU are the wheel speeds. These are not an analog signal but a pulse train whose frequency is proportional to each wheel speed. They originate from steel-toothed wheel at the hub of the wheels whose rotation is sensed by a stationary reluctance sensor or Hall Effect sensor. In other word, every time there is a positive (or a negative) edge of the pulse train the wheel has traveled a proportional amount.

The micro-controller that executes the software code will either be interrupted by each pulse rising edge or automatically keep track of their occurrence and timestamp trough an interrupt capture mechanism. The time tracking is related to the clock, oscillator or crystal of the ECU. A related issue is the time when a valve is activated: a countdown timer is loaded with an initial value or a counter is compared to a value until it reaches that value. In either case, when the time has elapsed, an interrupt is generated to address the valve handling. These timing values are ECU specific.

ONE IMPLEMENTATION OF THE SOFTWARE

There are many ways to implement a system to meet the specifications through software code. When the system is simple, one can have a simple loop that gets executed for example 250 times a second. During each iteration, the wheel speeds are calculated based on their respective recorded external interrupts. From the wheel speeds, the software can estimate the vehicle speed and therefore the slip at each wheel. If the slip is excessive, the isolation valve is activated, and the relief valve is pulsed to reduce the excessive pressure at the brake. When the pressure is sufficiently reduced, the wheel accelerates back to the vehicle speed, at which point the brake pressure is raised by pulsing off the isolation valve.

The main loop is calculated fast enough so that the micro-controller is idle a few moments before the next iteration. This allows to keep the main loop execution at a constant rate, which in this example is 250 Hertz.

INCREASING THE COMPLEXITY

This simple electronic system has been successful for the past 35 years. But there is always room for improvement. There are the fail-safes, which are features to detect failures of components such as the wheel sensors or the valves' coils. There are also graceful degradation modes which allow the system to function as best as possible when some of the system components have failed. Performance specifications can also increase to include braking control with yaw-moment buildup delay, i.e. to have an initial pressure hold mode on the high mu wheel during a split mu (different coefficient of friction from left to right side of the car) [2]. An additional specification would be a low mu select on the rear axle to improve stability, i.e., the same pressure control is used on both rear brakes based on the rear wheel with the least adhesion. Yet another specification would require both front wheels to lock up when aquaplaning is detected. These increases of complexity also increase the risk of software implementation errors. Adding new sensors and inputs (e.g., acceleration, gyros and 4x4 switches) raises further the complexity of the system. To this, one should not forget the increase of communication on the ECU such as SPI, I2C as well as inter-ECUs communication such as over the vehicle CANbus.

To handle this increased complexity, real time operating systems (RTOS) have evolved. The operating systems run as kernels on embedded systems and schedule tasks to get the system to meet the specifications. The real time aspect strives to meet the specified time requirements. It is quite easy nowadays to find an RTOS for a specific micro-controller. The prioritizing of tasks when scheduling is an interesting research topic, which is kept beyond the scope of this article.

CHANGING PLATFORM

As the complexity rises, there is often a need for more powerful micro-controllers while at the same time new lower cost alternatives reach the market. Automotive systems must at some point migrate to new platforms with different clocks such that all time designs have to be re-designed and re-verified repeatedly until the system can be signed off on. Simulations are becoming attractive and improving all the time. However, a fundamental issue is the fact that a simulation is performed on a computer with single or multiple cores or on a cluster of computers rather than on a micro-controller. When the developed software is moved from one platform to another, how much of it has to be modified or redesigned to fit on the next platform when the specification of the systems remains the same?

CHANGING PARADIGM

On one hand are clear specifications that describe the automotive system and on the other hand is the reality of computer hardware. The development process is then a continuous translation from the specification to flowcharts, state machines and universal modeling language (UML) representations down to software code. All this before a complier is ever invoked.

But, there could be a different approach to the system implementation. One could model the software with objects that react to events. The state of these objects should be protected from others objects to prevent accidental memory corruption. The objects' interactions and their timing could stem right from the specifications, and may even be expressed in SI units. Fig. 1 shows a model for such a paradigm.

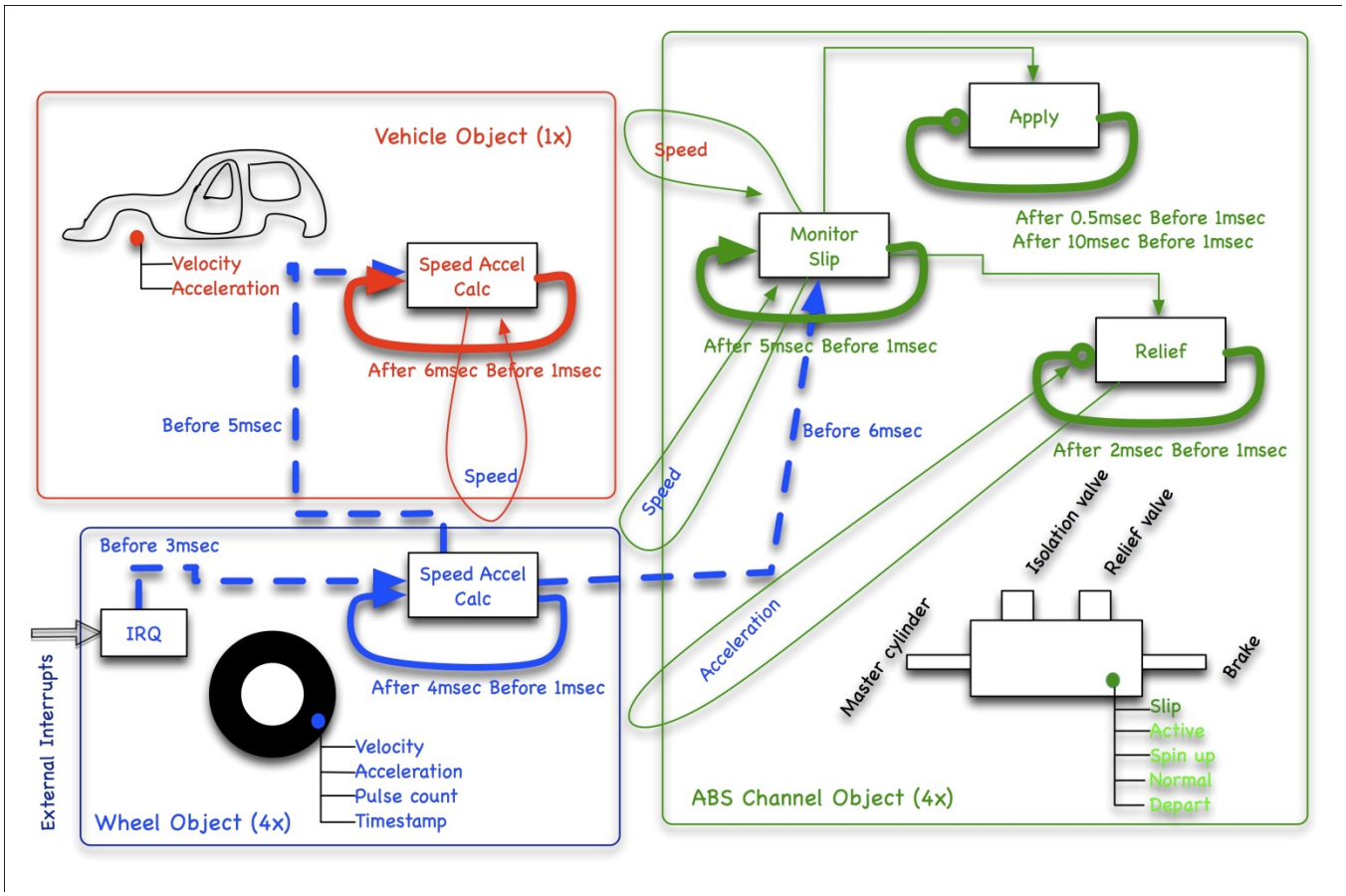


Figure 1 – A model of reactive objects forming an anti-lock braking system.

To illustrate this, one can consider a wheel with a toothed ring with 50 teeth and a circumference of 2 meters. Every time the micro-controller receives a positive edge, the wheel's traveled distance is 0.04 meter. The velocity in meters per seconds is just a function of the number of interrupts times 0.04 meters divided by the time necessary to accumulate them. The acceleration is the change in velocity between two instants in time.

Another example is a vehicle. To estimate its velocity, the information of the wheel speeds is needed. However, this should not be done by accessing memory directly, but rather by requesting it from every wheel. Starting to grant access to the memory space of other objects can lead to accidental tempering of the states. An interesting issue here is the need of update rates: does a vehicle's speed and acceleration with its larger inertia need to be updated as often as that of a wheel with its smaller inertia? The point being that the dynamics of the system should guide the design.

With the information of vehicle speed and wheel speeds, the ABS design continues by following the system specification. None of the examples are platform dependent. The design focuses on components that stand alone but communicate with each other. Adding a new specification or input to one component should not alter another. For example, adding a vehicle accelerometer or radar should not have anything to do with the wheel or the ABS logic, but only with the vehicle and its representation. Adding a split mu surface detection at the onset of ABS activity has nothing to do with the vehicle or the wheels; it is an ABS specification and is therefore related to that object.

This is the type of implementation that is done here for demonstration purposes. The following sections describe the simulation environment used for the demonstration, the underlying tool for the implementation and the implementation itself.

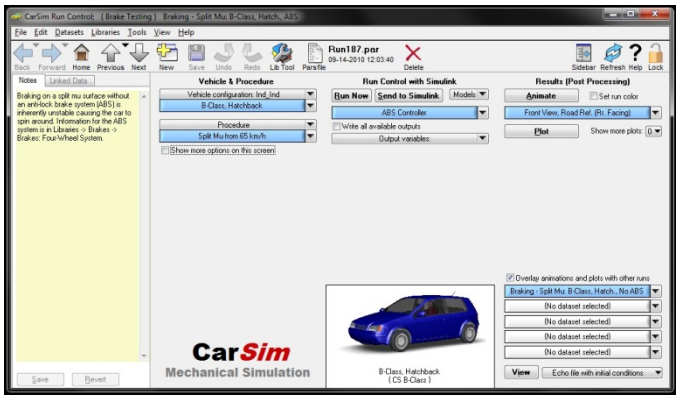


Figure 2 – The CarSim interface.

SIMULATION ENVIRONMENT

The best way to let people experience this type of paradigm and system implementation is to use a standard automotive simulation program and provide freely all the associated software code such that they can try it on their own [1]. The Simulation program used is CarSim 8.0 and to make things completely reproducible none of its settings have been changed. The vehicle chosen is a type B hatchback performing a braking maneuver on a split-mu from 65 km/h (Fig. 2, left section). The analysis can be executed within CarSim or within Simulink (Fig. 2, middle section). The latter is chosen and the Simulink interface appears with CarSim represented by an s-function block (Fig. 3). The outputs from the CarSim block are the true vehicle and wheel speeds as well as the master cylinder pressure. The inputs to the CarSim block are the wheel brake pressures from the hydraulic control unit.

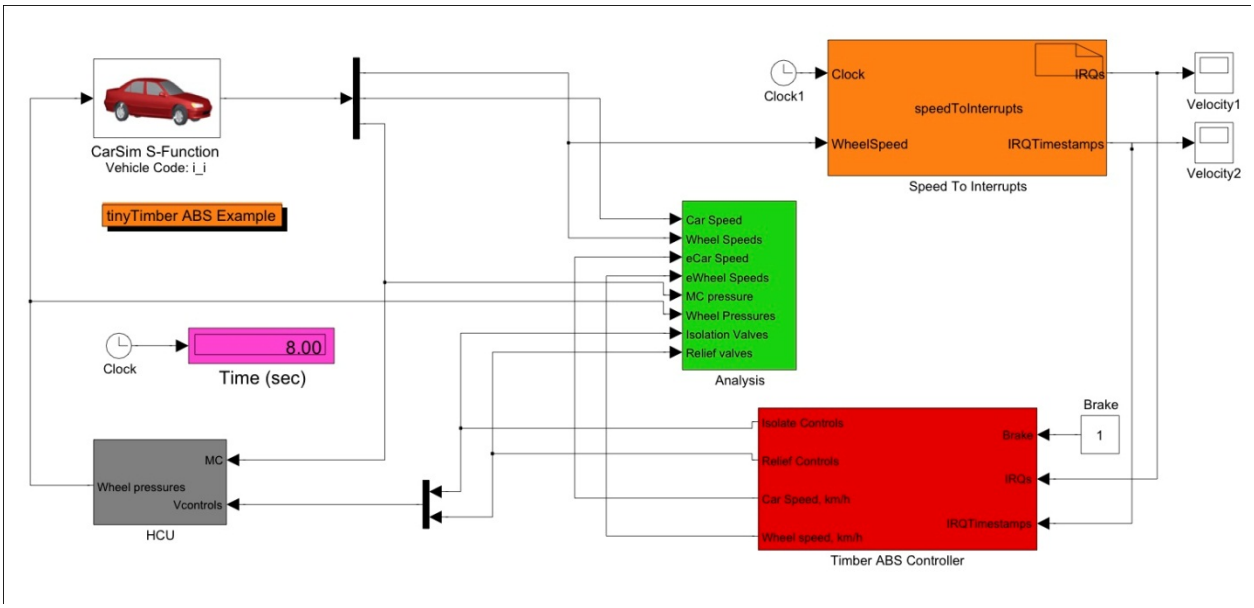


Figure 3 – Within the Simulink environment.

The CarSim original HCU and logic control must be replaced. The original HCU uses on signal per brake channel, which either applies or reduces the brake pressure. The system desired is one that can hold or isolate the pressure and relieve the pressure. Fig. 4 shows an implementation of the set of two valves. This simple implementation switches to the master cylinder in the normal case. If the isolation solenoid is activated, the input is the brake pressure. If the relief valve is activated, a zero pressure is the input to the valve. To keep things simple, both valves have the same orifice which results in a single first order behavior for the brake channel.

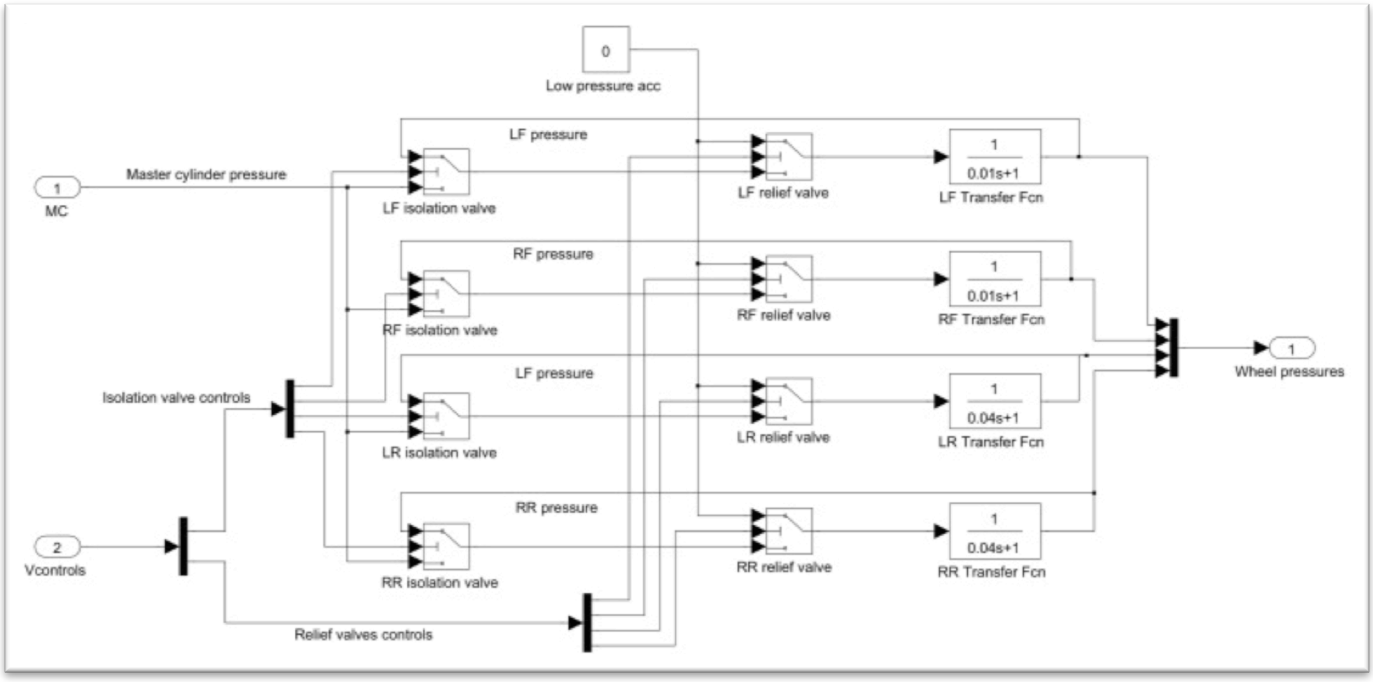


Figure 4 – The updated version of the hydraulic control unit.

A new s-function has to be added to transform the wheel speeds into a pulse train of interrupts and their timestamps for each wheel. This brings the simulation even closer to a real implementation. And at last there is the control block which contains the C code for the ABS.

There is no pump in the model since that CarSim does not model the limited volume of a master cylinder nor is it implemented in Simulink.

TIMBER AND TINYTIMBER

The implementation of the system in software is based on Timber. Timber is a real-time reactive programing language. The compiler for Timber is freely available from www.timber-lang.org. From Timber code, the Timber compiler generates C code for the application, which, together with a platform-specific kernel, can be deployed as a program under a general-purpose operating system or even directly on a hardware platform.

A simplified but less powerful version of Timber is tinyTimber, which is written by a programmer directly in C. It includes a set of rules for writing object-oriented code in C and a minimalistic run-time system that takes care of scheduling, memory management, etc. The run-time system support concurrent execution of messages sent from one object to another (hardware interrupts are also handled as messages to some specific software objects) while ensuring state integrity of the objects. Scheduling of execution is based on timing constraints for each message, defined in the code; these include a baseline, i.e. the earliest time when the message may begin execution, and a deadline. In this way, we avoid the volatile task of explicitly coding parallelism in terms of traditional processes /threads /semaphores /monitors, and side-step the delicate task of translating timing requirements on system operations into thread priorities [3].

To develop a better understanding of what is under the hood of the Timber language, one should refer to its home page. One can also begin with tinyTimber through the work of Eriksson [3].

SOFTWARE MODEL AND PERFORMANCE

For this implementation, the software consists of three types of objects and their associated methods. The three types are a wheel, a vehicle and an ABS channel. There are one instance of the vehicle type and four of the wheel type. There are four instances of the ABS channel type to create an ABS with four independent brake channels, therefore opting away the rear select low feature. Fig. 1 shows a model of the three types of objects. Within the three type blocs are the objects with some of their states listed and their associated methods. The lines between the methods refer to messages, i.e., asking a method to be executed. The thick lines refer to messages scheduling methods with a deadline (to be executed *Before*) and potentially a new baseline (delayed with the word *After*). The scheduled messages that are dotted are only executed once. The thin lines are asynchronous messages such that the next method is invoked immediately without being scheduled. The thin reflecting lines refer to synchronous messages that request other objects' state information, e.g., asking for a speed from a wheel.

Following an initial boot up (which includes port and pin assignment), the system is idle until external interrupts from the wheel speed sensors trigger the system to react (IRQ method in Fig. 1). After two interrupts per wheel, the method that calculates wheel speed is scheduled to be executed within 3 milliseconds (msec) (thick dotted blue line). After calculating the wheel speed and wheel acceleration, the method schedules itself after 4 msec with in a deadline of 1 msec (thick solid blue line).

As soon as one wheel speed is available, a vehicle speed can be estimated by requesting each wheel's speed information. The estimation in this demonstration is kept simple: the highest wheel speed sets the vehicle speed. There is a vehicle deceleration limit to address the case when all wheels are in deep slip. The method is then scheduled anew 6 msec later, at which point it should be executed within a 1 msec deadline.

Once the vehicle speed and a wheel speed are available, the wheel's associated ABS channel can monitor the slip by requesting the vehicle and wheel information (two thin green lines). If excessive slip is detected, the method will asynchronously initiate the methods that control and schedule activation and deactivation of the solenoid valves. The slip monitoring method is scheduled again 5 msec later and is then executed within a deadline of 1 msec.

The relief method is called upon when the wheel is decelerating much faster than the vehicle. Instead of pulsing the relief valve, the method keeps the valve open until the wheel stops to decelerate. It knows that by requesting the information from its respective wheel and schedules itself to check every 2 msec with a1 msec deadline. This is for demonstration purposes since the wheel acceleration is updated every 4 msec but the wheel speed and acceleration calculation rate could be changed during the pressure relief to a rate of 500 Hertz. The thick green line representing posting a message does not terminate with an arrow but a circle. The idea was to point out that there is an end to this self-scheduling: the wheel is accelerating back to the vehicle speed.

When the wheel has reached again the vehicle speed, the monitoring method reacts by invoking a pressure reapply method that schedules the subsequent pulse and hold mode. This self-scheduling terminates when 16 apply pulses are sequentially executed without requiring a relief of the pressure. At that point the ABS channel deactivates itself.

When adding to the specification requirement of braking control with yaw-moment buildup delay, it is only the ABS channel logic that is affected by that. When the ABS channel on one side of the car becomes active, it informs the channel on the opposite side of the car, which if not already active will isolates its brake pressure and schedules a pressure pulse up half a second later in the event that its wheel did not go into excessive slip by then. This feature is in the code but not in Fig. 1 keeping the figure clearer.

SIMULATION RESULTS

The true vehicle and front wheels speeds are presented in the appendix along with the calculated wheel speeds and estimated vehicle speed. The master cylinder and associated wheel brake pressure is also provided along with the valve signals. The wheel speeds are calculated correctly and the estimated vehicle velocity is acceptable. One can see at the beginning of the stop that the vehicle deceleration is limited. The average pressures clearly show that one wheel can hold a higher pressure since it has a higher coefficient of adhesion with the road. The CarSim animation can be viewed on http://www.youtube.com/watch?v=hnDkQLct7Ws.

ON-GOING AND FUTURE PLANS

Through this work, it is shown that tinyTimber, and as a matter of fact Timber, can be used to develop with ease automotive systems based on the system specifications. Timing requirements can be specified directly in the code in a platform-independent manner.

But this last step, the same code on different platform, can be accepted but it has not been demonstrated. Short of having a real car, a one-fifth scale remote control car from FG-Modellsport is being modified to have independent brakes to demonstrate that the same software can be used on different micro-controllers and in the simulation environment.

SUMMARY/CONCLUSIONS

The use of a modeling reactive programing language such as Timber enables engineers to easily develop automotive systems based on their specifications without the limitations of a fixed platform. This has been demonstrated in this article and the code is made available for others to evaluate the concept. This also leads the path to an ease of system integration as objects interact with each other through messages and proper timing.

REFERENCES

1. "EIS Architecture homepage", Demonstrators, http://www.ltu.se/csee/research/eislab/areas/eis_architecture?l=en, September 2010.
2. Bosch Automotive Handbook, 6th edition, Robert Bosch GmbH, 2004.
3. Eriksson, J, "Embedded Real-Time Software using TinyTimber - Reactive Objects in C", Licentiate thesis, Computer Science and Electrical Engineering department, Luleå University of Technology, 2007, ISSN: 1402-1757, http://epubl.luth.se/1402-1757/2007/72/index.html.

CONTACT INFORMATION

All the authors work for the division EISLAB (www.ltu.se/eislab) at Luleå University of Technology in Northern Sweden. Their email addresses are *firstname.lastname*@ltu.se.

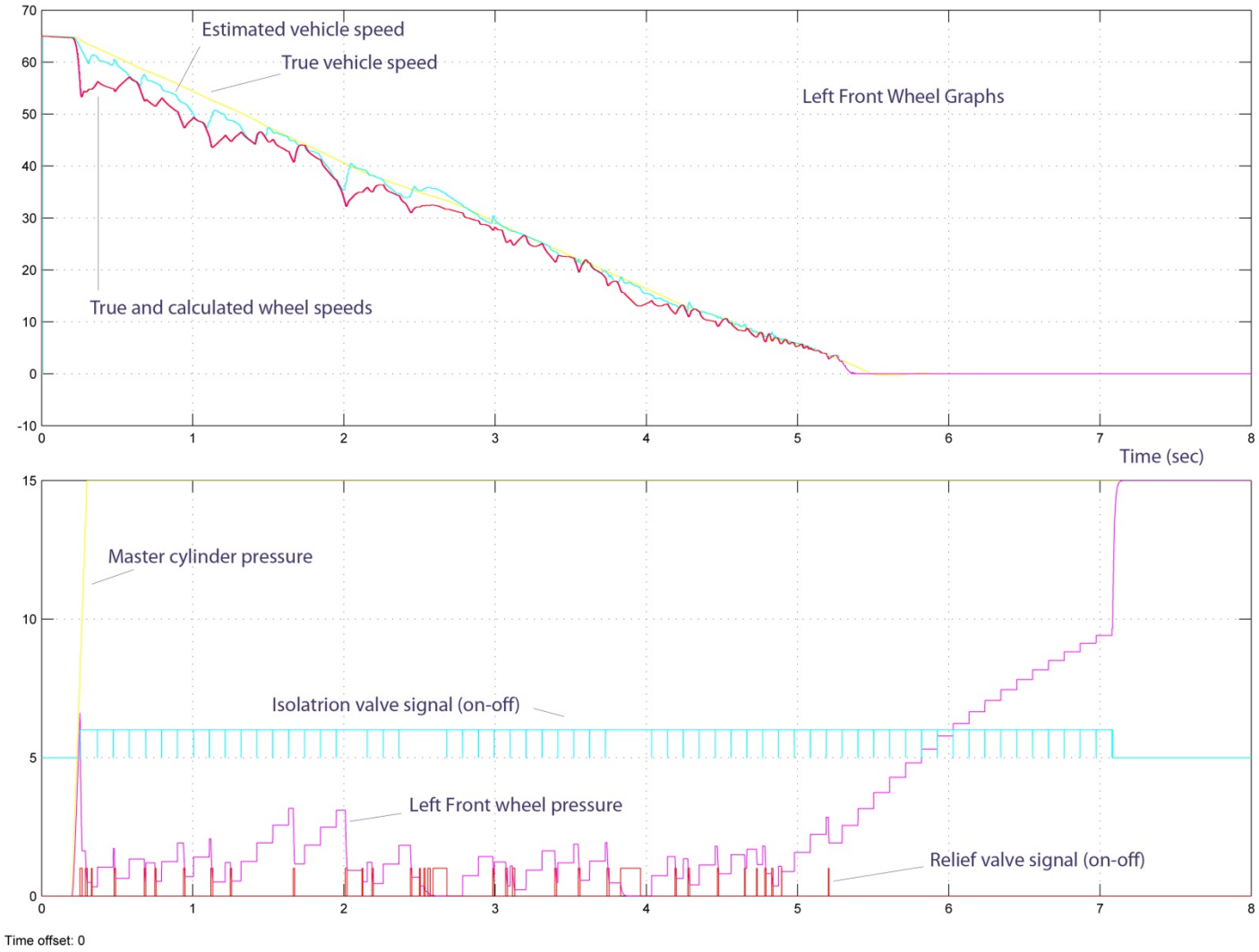
ACKNOWLEDGMENTS

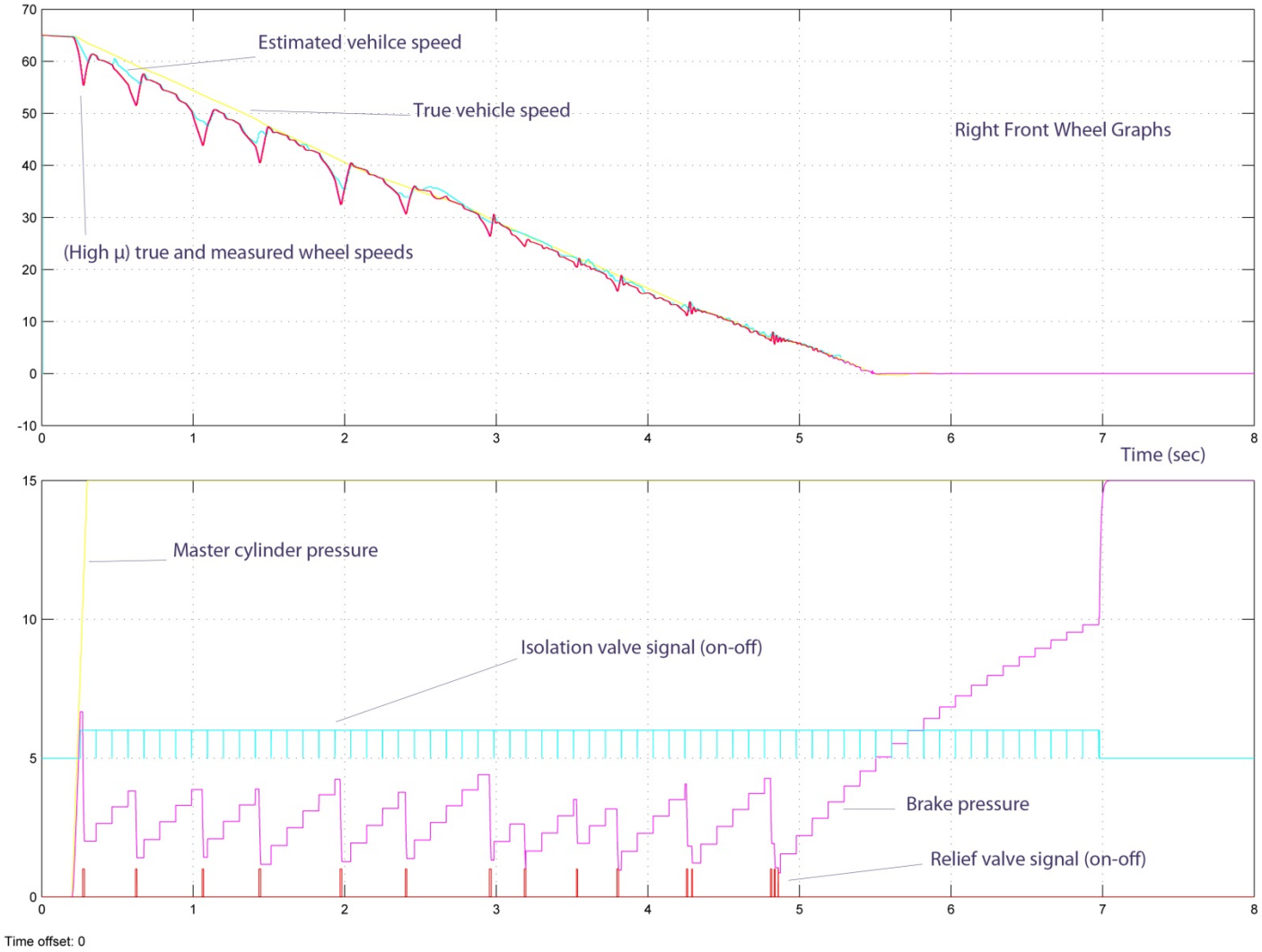
The Authors are grateful to the Center for Automotive Systems Technologies and Testing (CASTT) for its financial support in this research.

DEFINITIONS/ABBREVIATIONS

ABS	Anti-lock Braking System
ECU	Electronic Control Unit
HCU	Hydraulic Control Unit
RTOS	Real Time Operating System

APPENDIX


Estimated vehicle speed
True vehicle speed
Left Front Wheel Graphs
True and calculated wheel speeds
Time (sec)
Master cylinder pressure
Isolatrion valve signal (on-off)
Left Front wheel pressure
Relief valve signal (on-off)
Time offset: 0


Estimated vehilce speed
True vehicle speed
Right Front Wheel Graphs
(High μ) true and measured wheel speeds
Time (sec)
Master cylinder pressure
Isolation valve signal (on-off)
Brake pressure
Relief valve signal (on-off)
Time offset: 0