

Design Space Exploration of Automotive Platforms in Metropolis

Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli
University of California, Berkeley

Sampada Sonalkar, Sri Kanajan, Claudio Pinello
General Motors Research

Copyright © 2005 SAE International

ABSTRACT

Automotive control applications are implemented over distributed platforms consisting of a number of electronic control units (ECUs) connected by communication buses. During system development, the designer can explore a number of design alternatives: for example, software distribution, software architecture, hardware architecture, and network configuration. Exploring design alternatives efficiently and evaluating them to optimize metrics such as cost, time, resource utilization, and reliability provides an important competitive advantage to OEMs and helps minimize integration risks. We present a methodology (Platform-Based Design) and a framework (Metropolis) to support efficient architecture exploration. We have exercised the methodology and the capabilities of Metropolis for developing a library of automotive architecture components and performed design space exploration on a chassis control sub-system.

INTRODUCTION

The electronic content in the automobile is increasing at a rapid rate. In-vehicle electronics are displacing the traditional mechanical interfaces and enhancing the driving experience in every area from safety to telematics. OEMs face a number of challenges related to the integration of this electrical content. Specifically, one of the more significant challenges lies in the architecture or design space exploration realm.

The design of electronic architectures in the automotive domain is presently an ad-hoc activity that relies heavily on the expertise of the architect. Architecture optimization is essentially driven by the architect using a qualitative and manual design approach. Looking at the future, as the complexity increases, there is a clear need for a cohesive methodology that allows the designer to make various design choices quickly and evaluate them systematically using tool-based quantitative and qualitative analysis (see e.g., [4] for a concrete view of

this concept). Considering the role of OEMs as integrators of a heterogeneous set of electronic subsystems fed by a complex and distributed supply chain, it is essential for the design methodology to facilitate the capture of constraints and requirements on the architecture coming from the supplier components.

The platform-based design methodology described in [1], and supported by frameworks such as Metropolis [2], addresses these requirements by advocating the separation of concerns and abstraction refinement.

This paper presents how the platform-based methodology and the supporting framework applies to the automotive domain, and investigates their benefits in terms of design space exploration. The paper is organized as follows: In the next section, we describe the platform-based design methodology and discuss its significance in the automotive domain. Next, we describe the Metropolis tool framework. In the subsequent section, we outline the case study and give details of the function and architecture models. In the design space exploration section, we discuss the different design choices and report results and observations. Finally, we conclude with a summary and avenues for future research.

PLATFORM-BASED DESIGN METHODOLOGY

The basic principle of the platform-based design methodology [1] is the “orthogonalization of concerns”, i.e. separating the various aspects of a design to facilitate greater design productivity. The methodology primarily advocates separation between the following aspects of a design:

1. Function (what the system does) and architecture (how it does it)
2. Computation and communication
3. Behavior and performance

The separation between function and architecture is motivated by the observation that implementing an electronic system is not a purely top-down flow. Instead, a significant portion of the design effort will consist of re-designing functional components to a different architectural platform (e.g. from an FPGA to an ASIC-based processor). Having this separation between the processor type and the functionality through a middleware layer enables the quick exploration of different execution platforms.

Separating computation and communication is driven by the observation that communication schemes are typically more standardized than the computational aspects of an electronic system. If the boundary between these two aspects is made explicit, then a particular standardized communication scheme can be quickly imported into the design. For instance, regardless of the type of ECUs, sensors or actuators in a system, a CAN bus can be used to implement communication between these components. Therefore, the adherence of these components to a CAN interface is crucial during the modeling phase.

The final separation is between behavior and performance of a system. If these two portions of a system are cleanly separated, then the amount of modeling effort required to experiment with different performance alternatives is minimal. For instance, the clock rate of an ECU should not affect the values of messages sent, only their timing. If the system model separates behavior and performance, a system designer can easily experiment with different ECU clock rates and gauge their impact on the overall system.

PLATFORM-BASED DESIGN ACTIVITIES

There are four basic steps in a platform-based design flow: choosing the model(s) of computation (MoC), modeling the functionality of the system, modeling the architectural platform, and carrying out different mappings of functions to architectural elements to evaluate different points in the design space.

The first step involves choosing one or more models of computation and abstraction levels that are appropriate for modeling both the functionality and the architectural platform. A model of computation is a mathematical entity that specifies the interaction between components in the system. The use of formal MoCs facilitates verification and synthesis activities. The model of computation must capture the essential aspects of the functionality that are necessary to ensure correctness [2]. Also, it must be possible to implement the MoC on the architectural platform in an efficient manner – it must be a good “match” for the architectural components. A specific model of computation may be realized at different abstraction levels. The abstraction level can be chosen based on the accuracy of models that are desired. Abstract models may not accurately capture the performance of the system, whereas refined models may cause unnecessarily high overhead. The result of

this step in the design flow is agreement on a set of common services at a particular abstraction level and a set of rules that determine how these services can be composed – the MoC. After this step in the design flow, both the functional and architectural modeling activities can be carried out concurrently.

A functional model is an abstract view of the behavioral aspects of the system. It captures the actions the system will carry out, but does not indicate how these actions will be performed. Therefore, it has no concept of implementation associated with it. A functional model utilizes the common services as specified by the MoC to realize its behavior. However, the functional model does not explicitly rely on an architectural model for completeness – it is an independent executable specification of the behavioral aspects of the system.

The architectural model captures how the common services will be realized using the architectural components. The performance aspects of implementing these services are captured by the architectural model. An architectural model has no concept of the usage pattern of the services it provides, only that the services will be utilized in a manner consistent with the rules given in the MoC. An architectural model may be parameterizable, i.e. it may implicitly represent a family of architectural platform models. For example, a processor model could be simply parameterized with a clock speed, which would enable it to be instantiated to reflect any processor speed.

Mapping is the step in the design flow that allows the services utilized by the functional model to be bound with the services provided by the appropriately configured architectural model. The definition of a legal mapping depends upon the specific MoC used. A mapping represents a particular point in the design space of the system. After a mapping is realized, the performance of the system can be estimated using simulation. An iterative design space exploration procedure can be carried out to explore different points in the design space.

SIGNIFICANCE FOR THE AUTOMOTIVE DOMAIN

The basic idea of the methodology is to hide unnecessary details of the architecture, summarize its important parameters into an abstract model, and to limit the design space exploration to a set of platform instances. This disciplined approach can be used to resolve many issues in the automotive domain.

Due to the clear separation of function and architecture, there is flexibility in distributing functional processes across ECUs. The functional models are unaffected by changes in hardware, software, and network architecture. Hence the designer can explore various design alternatives in an efficient manner. Moreover, both functional and architectural models can be reused more easily across a product line that has different architectures. Once the functional model is mapped to

the architecture, it is then possible to evaluate the architecture with respect to a number of metrics such as time, cost, power, and resource utilization.

METROPOLIS

Metropolis [2] embodies the platform-based design methodology and provides a unified framework for simulation, synthesis and verification of heterogeneous systems. The infrastructure supports many different MoCs and abstraction levels. The specification language for Metropolis is known as the “metamodel”, as any specific model of computation can be obtained by refinement, and supports both imperative and declarative specifications for modeling functionality, architecture, and mapping. Metropolis also provides a set of backend tools, including a simulator, and interfaces with formal verification tools.

FUNCTIONAL MODEL

The functionality of a system is described as a set of concurrent processes that execute independently and communicate with each other. Each process has a single thread of execution. Processes communicate with each other by calling methods on ports. A port is associated with an interface that declares the set of methods associated with that port. Media are passive objects that implement interfaces. In this way, computation and communication are orthogonalized in Metropolis – processes implement computation, while media can be used to describe the communication.

Processes and media can be hierarchically composed into networks.

Figure 1 shows a network of two producer processes and one consumer process that communicate through a medium. Each process executes a sequence of read(), execute() and write() actions. The declarative constraint specifies the mutual exclusion of the producers when accessing the shared resource.

ARCHITECTURE MODEL

There are two aspects to the architecture model: the services it offers to the functional model and the cost of providing these services. At the top level, the services are represented by the Task processes shown in **Figure 2**. The tasks themselves use the services of media such as CPU, bus and memory for providing the services.

The efficiency of implementation is determined by specifying the cost of each service. Each service is broken down into a sequence of events and each event is annotated with a value representing its cost. The cost may be in terms of CPU cycles, time, power, etc. Quantity managers represented in the illustration as diamonds connected to the media perform the

annotation of costs to events and also arbitrate the execution of events.

Since the performance-related information is specified in the quantity managers, there is a separation between the behavior (services) and performance aspects of an architectural model.

MAPPING

To evaluate the performance of a particular implementation, the functional model needs to be mapped to an architectural model. This is done by adding declarative constraints to the specification. These constraints synchronize the events of the functional model with corresponding events of the architectural model.

The synchronization mechanism corresponds to an intersection of the sets of legal behaviors of the functional and architectural models. Functional model executions specify a sequence of events for each process, but usually allow arbitrary interleaving of event sequences of the concurrent processes, as their relative speed is undetermined. On the other hand, architectural model executions typically specify each service as a timed sequence of events, but exhibit non-determinism with respect to the order in which services are performed, and on what data. Mapping eliminates all executions from the two sets, excluding those in which the events that should be synchronized always appear simultaneously. Thus, the remaining executions represent performance-annotated sequences of events of the concurrent processes. Error! Reference source not found. shows a mapping network that combines the functional network from **Figure 1** with the architectural network from **Figure 2**. Events of the two networks are synchronized using the constraint clause in the mapping network. For example, executions of foo, read, and write by P0 have been synchronized with executions of execute, read, and write by T1. Since P0 executes its actions in a fixed order, while T1 chooses its actions non-deterministically, the effect of synchronization is that T1 is forced to follow the decisions of P0, while P0 “inherits” the quantity annotations of T1. In other words, by mapping P0 to T1 we make T1 become a performance model of P0. Similarly, T2 and T3 are made to be performance models of P1 and C, respectively.

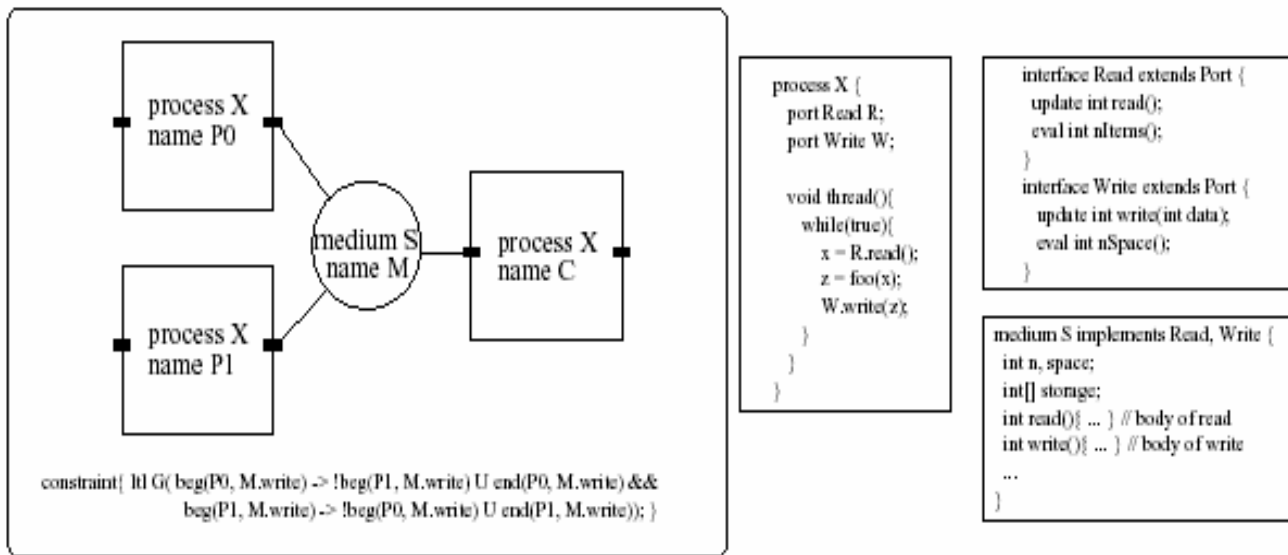


Figure 1 Functional model of two producers and one consumer

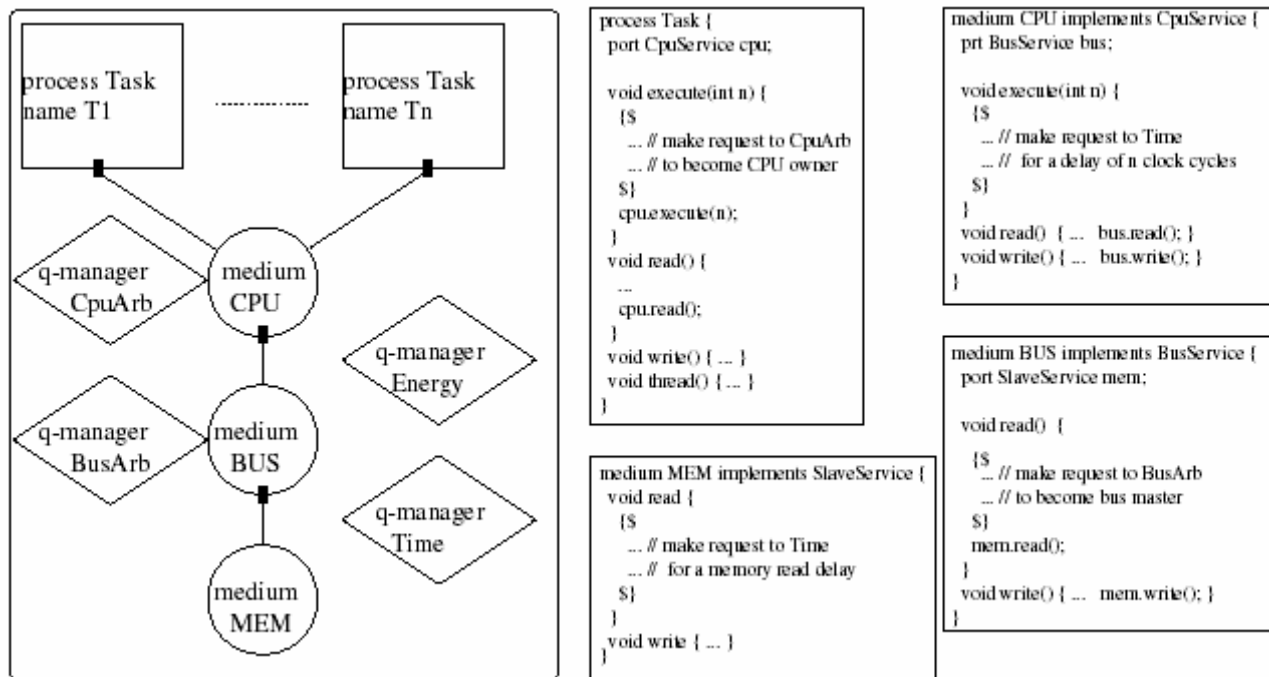


Figure 2 An architectural model

TOOL SUPPORT

The Metropolis framework includes tools to perform a variety of tasks such as simulation, property verification, and synthesis. The Metropolis system description is mapped to SystemC [5] that is used for simulation. Quantity managers compute quantities such as timing and power consumption and associate them to events. Hence, the performance numbers for a simulation run are calculated by quantity managers and then are

summarized at the end of the run. Quantity managers are an important innovation supported by Metropolis which is rather unique in capturing and analyzing non-functional properties of a system. The formal verification tool Spin [6] is used for verifying properties of the functional model. The Metropolis framework supports a special purpose language, the Language Of Constraints (LOC) to express constraints related to timing and performance. An LOC checker is also provided to verify properties.

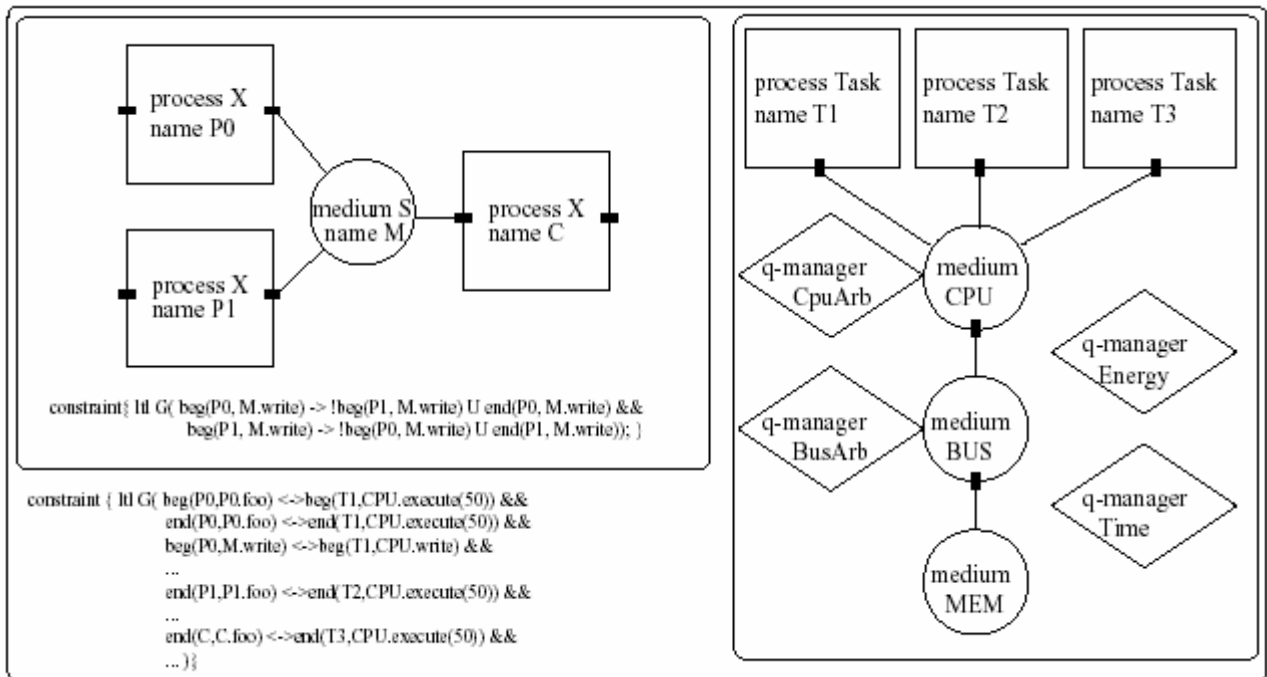


Figure 3: Mapping of function to architecture

CASE STUDY: DISTRIBUTED SUPERVISORY CONTROL SYSTEM

This system is a limited-by-wire system that implements a supervisory control layer over the steering, braking and suspension system. The objective is to integrate active vehicle control subsystems to provide enhanced riding and vehicle performance capabilities.

The high-level view of the functional architecture of this control system is defined in **Figure 4**. Using sensors to collect data on the environment, the supervisor plays a command augmentation role on braking, suspension and steering. This supervisory two-tier control architecture enables a flexible and scalable design where new chassis control features could be easily added into the system by only changing the supervisory logic.

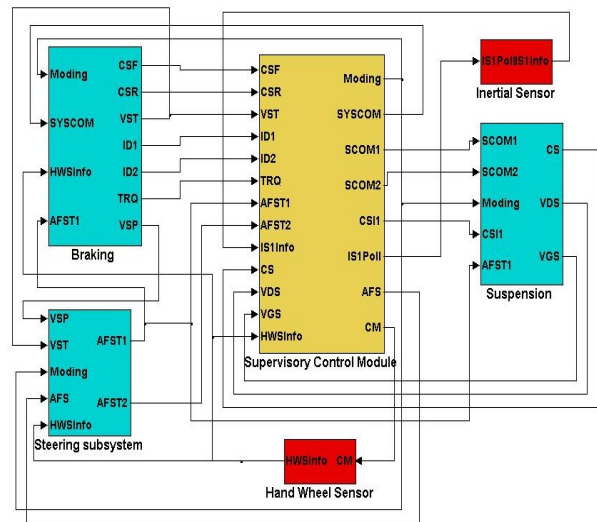


Figure 4: Functional Architecture of Supervisory Control System

The baseline physical architecture consists of 6 ECUs and includes 2 CAN-based smart sensors connected over a high-speed CAN communication bus. The interfaces to the rest of the vehicle subsystems such as Body and Powertrain are ignored at the moment. **Figure 5** describes this physical architecture. It consists of a Supervisory Control Module (SCM), Steering control, Braking control, Suspension control and two sensors viz., Hand Wheel Sensor to obtain the steering position and an Inertial Sensor for Yaw Rate Lateral Acceleration. The Simulink model for the interactions between these units is shown in Figure 4.

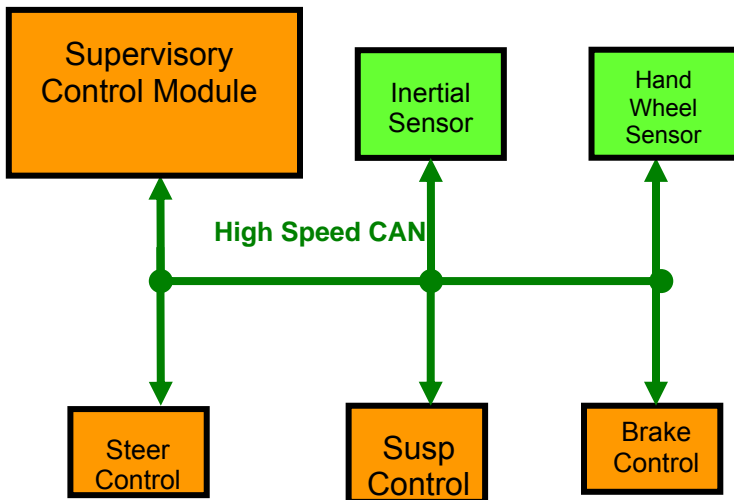


Figure 5: Physical Architecture of Supervisory Control System

THE METHODOLOGY

To develop an appropriate methodology based on PBD for automotive systems, we first examine the shortcomings of current design practice. Simulink is commonly used to develop control algorithms for automotive systems. Real-Time Workshop is used to generate code for these systems. However, RTW assumes that the code will be deployed on a single-processor architecture, whereas it is typically deployed on multi-ECU systems. Since ECUs are not perfectly synchronized and messages may be lost/overwritten in transit, the assumptions made by the code generation tool may not hold. Without any guarantees on system behavior, designers typically rely on two techniques: oversampling and testing. Oversampling mitigates the effects of message loss, at the cost of higher resource utilization. Extensive testing of the physical system is the predominant design validation technique. Clearly, testing on the physical system can only be performed late in the design cycle, especially because the automotive industry is often organized in a multi-tiered supplier/OEM structure. This approach severely limits design productivity by prolonging and hindering the design space exploration procedure and making system verification difficult and expensive.

The main source of problems stems from the mismatch in the assumptions at the functional and architectural levels.

In this paper, we close this gap by including the non-idealities of the architecture in modeling the functionality. We present a framework where important

tasks during testing and validation can be performed long before the physical system is available.

MODEL OF COMPUTATION

To close this gap, we specify the functional aspects of the system with a semantic that is compatible with that of the architectural platform. In this model of computation, *independent processes communicate using unidirectional media*. Processes are activated by an external source. After activation, the processes carry out a predetermined sequence of non-blocking reads and writes on the media interleaved with the required computation. Media have FIFO behavior with signal loss and duplication. Signals on a media are not delivered out of order.

The functional model can be simulated by specifying an activation pattern. At this stage, it is possible to assess the effects of non-idealities on the behavior of the system. Once the functional model is mapped with an architectural platform, task activation and signal/message loss/duplication follow non-idealities such as jitter that are exhibited by the actual architectural components.

FUNCTIONAL MODEL OF SUPERVISORY CONTROL SYSTEM

The processes in the functional model wait until they receive a trigger request through a trigger medium. After the trigger is received, the processes carry out a sequence of non-blocking read and non-blocking write operations to the input and output media respectively. The media in the functional model are initialized with initial tokens. When a token is written to the medium, it is queued behind the existing tokens and a unique sequence number is generated for that token. Correspondingly, a read operation requires a sequence number as input. The token corresponding to the sequence number is returned and all tokens ahead of this token are deleted from the medium, regardless of whether they have been read. This can model signal loss, due, for example, to overwrites. Conversely, the same sequence number can be provided for the subsequent read operation, and the same token will be returned. This corresponds to signal duplication in the medium, due, for example, to missed/slower updates.

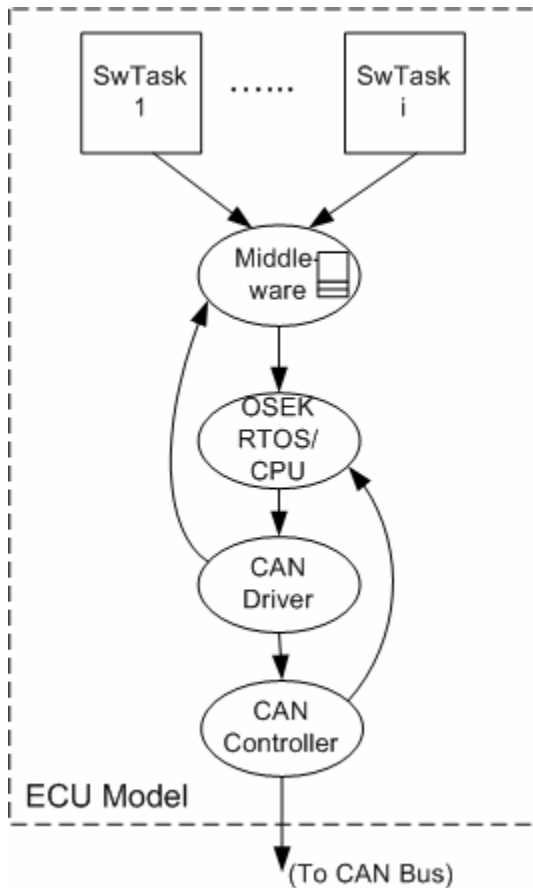


FIGURE 6: ECU MODEL

When the functional model is simulated independently, the sequence numbers for the read operations are generated to correspond with the message loss/duplication parameters for the model. When the functional model is mapped to the architectural model, these sequence numbers follow the loss/duplication characteristics of the architectural simulation faithfully.

The inclusion of sequence numbers is a modeling artifact for both the functional and architectural models. It is required to keep the functional model and architectural model in lock-step during simulation. However, the use of sequence numbers does not affect the performance estimation, namely the computation of latency or power.

ARCHITECTURAL MODEL

The architectural model consists of 4 electronic control units (ECUs) and 2 sensors connected to the CAN bus (**Figure 5**). Each ECU consists of software tasks, middleware, a CPU scheduler, an interrupt handler, a CAN Driver, and a CAN Controller, as shown in **Figure 6**. The sensors have a CAN interface to inject sensor data onto the bus. The following sections describe each of these sub ECU components.

SOFTWARE TASK

The software task provides a set of services such as get(), put() and compute() to the functional model. It uses the services of the middleware for this purpose. The software task is given execution time by the real-time operating system.

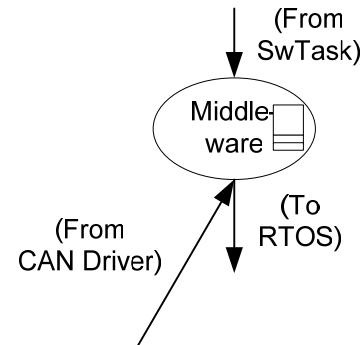


FIGURE 7: MIDDLEWARE MODEL

MIDDLEWARE

The middleware layer in Figure 7 handles the transfer of data between tasks. It provides the following concepts of transparency:

1. Location transparency. The source and destination of data are hidden from the tasks. Tasks simply read and write to the middleware buffers. The middleware handles the transfer of data using either local memory or remotely over a communication bus.
2. Access transparency. The resource's internal representation and its access mechanism are hidden from the producers and consumers of the resource. For example, the middleware packs the signal data from processes into CAN messages before transmission. Access to critical resources by multiple features is hidden by having a scheduler that provides mutual exclusion.

The middleware achieves this by maintaining a list of all the signals in the ECU and implements a mechanism for packing and retrieving signals from the CAN messages.

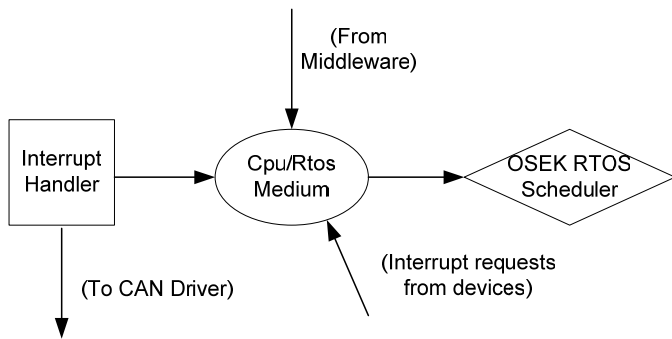


FIGURE 8: CPU AND INTERRUPT HANDLER MODEL

CPU SCHEDULER AND INTERRUPT HANDLER

The CPU scheduler implements a priority-based preemptive scheduling algorithm as specified in the OSEK specification for BCC1 type tasks. The interrupt handler is used to service interrupts generated by the CAN Controller.

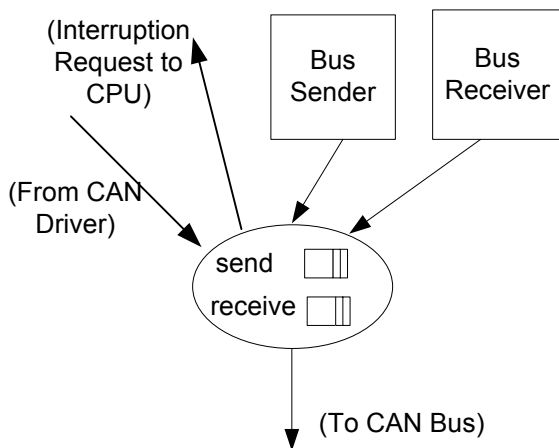


FIGURE 9: CAN CONTROLLER MODEL

CAN CONTROLLER

The model of the CAN Controller is based on the specifications in [3]. It consists of the BusSender and BusReceiver processes, and the transmit and receive buffers.

The BusSender and BusReceiver processes run concurrently with the software tasks. This models the CAN Controller hardware that works concurrently with the CPU in a real hardware implementation. In the CAN protocol all the BusReceiver processes listen to the bus. When the bus is idle (detected by receiverState==IDLE), each station transmits the highest priority message in its queue. The station that transmits the highest priority

message wins the arbitration. All other stations stop transmitting and wait for the bus to go idle again. The sender finishes transmission and monitors the checksum (shown by ACK), sends end of transmission bits and returns to the idle state.

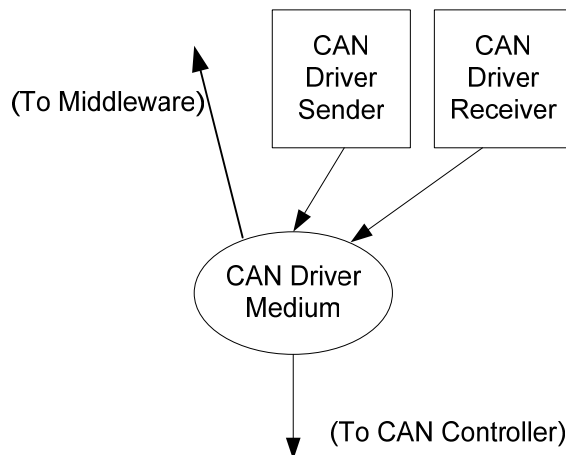


Figure 10: CAN Driver Model

CAN DRIVER

The CAN Driver transfers the messages between the Middleware and the CAN Controller. It can be operated in either polling or interrupt-driven mode. In polling mode, the CAN Driver polls the CAN Controller to check for an empty transmit buffer or the presence of a new message in the receive buffer. In interrupt mode, the CAN Controller interrupts the CPU when a message has been transmitted successfully or when a new message is in the receive buffer. The interrupt handler then invokes the CAN Driver through an interrupt service routine.

MAPPING

For every process in the functional model, a task is defined in the architectural model. The task is a placeholder where functions can be mapped so that they can be scheduled for execution. The read() and write() methods of the process are mapped to the get() and put() methods of the task using synchronization constraints. The processes execute at a rate that is enforced by the architecture using a quantity manager and a trigger.

DESIGN SPACE EXPLORATION

One of the main goals of this infrastructure is to enable

design space exploration. To achieve this goal, we need to be able to rapidly and effectively:

1. Model alternative design configurations.
2. Evaluate a given design configuration.

Let us first identify the details that we want to capture when modeling a design configuration. Of these details, there will be some that we are interested in varying to explore the design space: the **degrees of freedom** of the design.

DEGREES OF FREEDOM

The modeling infrastructure that we have developed so far allows a quick and effective exploration of the following important degrees of freedom:

- Functions to ECU allocation;
- Task priorities;
- Signals to messages packing;
- Message priorities;
- CAN Controller and CAN Driver specifics.

The flexibility in exploring these degrees of freedom derives primarily from separation of concerns, between function and architecture and the use of parameterized models. Moreover, the separation between computation and communication enables the exploration of different communication/interaction models between the application processes. In other words, we could explore different models of computation such as lossy vs. lossless communication channels, data-driven vs. periodic execution, etc.. However, in this paper, we did not explore this degree of freedom.

PERFORMANCE METRICS

As for the evaluation method, we are currently focusing on simulation as a way to assess the quality and properties of a given design configuration.

The metrics that we compute during simulation are related to timing and include latencies, rates, and number of message overwrites. In addition to these performance-related metrics, we also qualitatively assess the cost of the evaluated solution.

In the next section we illustrate some of the scenarios that we explored using this infrastructure.

PRIORITY INVERSION

Priority inversion is a well known but often overlooked problem in embedded systems and computer science [7]. The basic scenario involves three or more “tasks”

and a shared resource.

When the shared resource requires mutually exclusive access, e.g. a peripheral or a hardware register, it is recognized and generally accepted that a higher priority task cannot execute or complete execution until the resource is “released”, even if the task “occupying” the resource is a lower priority task. This phenomenon is not referred to as priority inversion, even though the higher priority task may need to wait for the completion of the lower priority task. The actual priority inversion, and the associated “pain”, comes from the presence of one or more intermediate priority tasks that do not need the shared resource and that may preempt or otherwise delay the completion of the lower priority task, which in turn delays the completion of the higher priority task. This latter phenomenon occurs between lower priority tasks and the higher priority task(s) without any mutual exclusion constraints that would otherwise justify it.

In our models this inversion may occur between messages of different priority originating from the same ECU, when transmit buffers are shared among them. In an extreme and not uncommon case, a single transmit buffer is shared in mutual exclusion by all transmitted messages.

Clearly, if a high priority message is queued in the middleware while a low priority message is sitting in the transmit buffer of the CAN Controller, the high priority message will be blocked until the low priority message is successfully transmitted. Again, this conflict in accessing the transmit buffer is not the inversion per se, but when intermediate priority messages are transmitted on the CAN bus by other nodes, they effectively delay the transmission of the low priority message and, consequently, of the high priority message.

MITIGATION STRATEGIES

An obvious mitigation strategy consists of having dedicated transmit buffers, so that there is no shared resource and all messages can compete for the bus arbitration solely based on their priorities, as they are supposed to do in the ideal protocol.

This solution may prove overly expensive, especially because configuring many CAN buffers for transmission reduces the number of buffers available for receiving messages, which may result in high message loss rates due to shared receive buffers.

Another conceptual strategy consists of “aborting” the current message when a higher priority message becomes available for transmission. The practical

viability of this strategy is highly questionable: there is a high risk that the abortion may be forced when the message has already won the bus arbitration and is being transmitted without further blocking. In this scenario, the effort to abort a transmission, signal the error, and to invalidate all recipients' copies would typically be much larger than if we simply waited for the completion of transmission.

A mix of the two strategies above may prove very effective. If we use two transmit buffers, we can easily determine which message should be aborted to make room for a higher priority message: the lowest priority message cannot possibly¹ have won the arbitration against the other message. We are assuming here that the minimum interval between two consecutive queue re-adjustments is longer than the transmission time of a message. If that were not the case, we could run again into pathological scenarios where we:

- First queue in the controller a medium priority message right when the low priority message starts arbitration;
- Then look again at the middleware queue, and find a high priority message,
- Finally, decide to abort the low priority message and overwrite the buffer with the high priority message.

Clearly, in this scenario we would be aborting a message that is already transmitting!

Ideally, the hardware in the CAN Controller could provide some support for this automatic re-ordering of messages using a pair of buffers and the knowledge of whether the message is being transmitted/arbitrated or is simply waiting for the next arbitration cycle.

Finally, we discuss a simpler strategy which is not based on message transmission abortion [9]. Each available transmit buffer is assigned a set of contiguous priorities. To focus ideas, let us consider the case of two transmit buffers. One of the two is dedicated to messages with priority higher than a given threshold; the other buffer can be used to transmit messages of any priority. In the case of three or more transmit buffers we need to define two or more threshold priorities. The idea is that we "reserve" at least one buffer to transmit messages of relatively higher priority, so they cannot experience priority inversion due to messages of relatively lower priority.

We may still observe priority inversion, but only among messages of the same "priority class". These inversions

¹ We are assuming that the controller transmits messages based on their priority.

are usually less severe, because the priorities are very similar, and fewer intermediate messages exist, if any.

TRADE-OFF ANALYSIS

We performed tradeoffs for the CAN Controller buffer configurations and gathered the performance numbers for message loss and message latency. The design choices for the CAN Controller configurations are summarized below:

Receive buffer configuration

1. Number of buffers:
 - a. Full CAN dedicated
 - b. Full CAN shared
 - c. Basic CAN⁺
2. Buffer masks:
 - a. Masks are optimized to minimize the number of spurious CAN messages entering the receive buffer
 - b. Masks are generated to minimize the message loss for high priority messages
3. Buffer overrun policy:
 - a. Discard new message
 - b. Overwrite old message
 - c. Priority based

Transmit buffer configuration:

1. Number of buffers
2. Buffer access policy
 - a. Message priority-based
 - b. Buffer priority-based⁺
 - c. FIFO⁺

CAN Driver configuration

1. Activation policy
 - a. Interrupt
 - b. Periodic⁺
2. Priority inversion mitigation strategy
 - a. none (shared transmit buffers)
 - b. partitioned priorities
 - c. message abortion⁺

The architectural model was annotated with the following numbers

- i. The ECU system clock operates at 40 MHz. The CAN Controller operates at 20 MHz.
- ii. THE WCET of a functional process is estimated to be 200 microseconds.
- iii. The CAN bus transmits data at the rate of 500

⁺ Presently these design choices have not been exercised in the tradeoff analysis

kbits/sec

SIMULATION RESULTS

We analyzed two different mappings and three different configurations of the CAN Controller and CAN Driver in the Supervisor module.

In the first mapping we use a smart sensor for the inertial measurements (as in **Figure 5**), while in the second mapping that sensor is hard-wired directly to the Supervisor module.

In the three CAN configurations of the Supervisor node we have

- one transmit buffer, shared
- two transmit buffers, shared
- two transmit buffers, one of which is shared and the other is dedicated to high priority messages (message ID \leq 0x400).

In the third configuration, the first buffer can transmit any message, including high-priority ones, but the second buffer cannot transmit the 6 lowest priority messages from the Supervisor.

RESULTS ANALYSIS

We will focus our attention on the following Supervisor signals and on the messages that contain them.

- "e_r_StrgRatOverrrdF_orw", in message 0x0D0
- "e_w_VehYawRateCntrd_orw", in message 0x700
- "e_transmit_id_isg", in message 0x7C0.

It is important to notice that in the second mapping some high priority messages from the Inertial Sensor are not transmitted over the bus and their signals become local signals.

In both mapping configurations, due to the specific offsets, messages 0x700 and 0x7C0 are enabled before message 0x0D0, respectively at times 1ms, 1ms, and 2ms.

Figure 11 and **Figure 12** depict the GANTT charts for the three messages above (columns) for each of the three CAN configurations (rows) and for the two mapping configurations, respectively.

Each chart illustrates the times when the signal is posted to the middleware buffer, to the CAN Controller transmit buffer, to the CAN Controller receive buffer (on the receiver node), and to the middleware buffer (on the receiver node).

Since we modeled the CAN Driver to operate in interrupt, signals stay in the CAN Controller receive buffer for a very short time.

In this exploration we are interested in highlighting priority inversion, so the response time of the third signal (within message 0x0D0) is the most interesting metric. In other explorations the designer may be interested in other metrics, like the timing of other events or the loss of signals due to overwrites.

In the first and second CAN configurations, the low priority messages "lock" the transmit buffer(s) for a long stretch of time, and, although enabled at time 2ms, the high-priority message 0x0D0 needs to wait in the middleware due to priority inversion.

It is interesting to note that in the second CAN configuration the additional transmit buffer does not mitigate the priority inversion problem. In fact, in the second rows of **Figure 11** and **Figure 12** message 0x0D0 is transmitted after message 0x7C0. More precisely, the start-of-transmission times in the second row of **Figure 11** are 4.2801ms for message 0x7C0 and 4.5321ms for message 0x0D0, and similarly in **Figure 12** they are 4.0581ms for message 0x7C0 and 4.1841ms for message 0x0D0.

In principle, after message 0x700 is transmitted, there is a transmit buffer available for message 0x0D0, but in practice it takes time for the CAN Driver to copy the message into the transmit buffer, and message 0x7C0 starts arbitrating/transmitting before then.

In the third CAN configuration only message 0x700 is allowed to occupy a transmit buffer, because the other buffer is dedicated to messages with ID \leq 0x400.

Clearly, when at time 2ms message 0x0D0 is enabled, it will be queued quickly in the vacant transmit buffer, thus avoiding priority inversion.

Finally, looking at **Figure 12**, we notice that in the second mapping configuration, due to the reduced traffic load, the latency of the other messages is generally improved. In particular, in the case of priority inversion message 0x0D0 puts up with a shorter waiting time because fewer messages from other nodes block message 0x700, hence reducing the penalty due to the priority inversion.

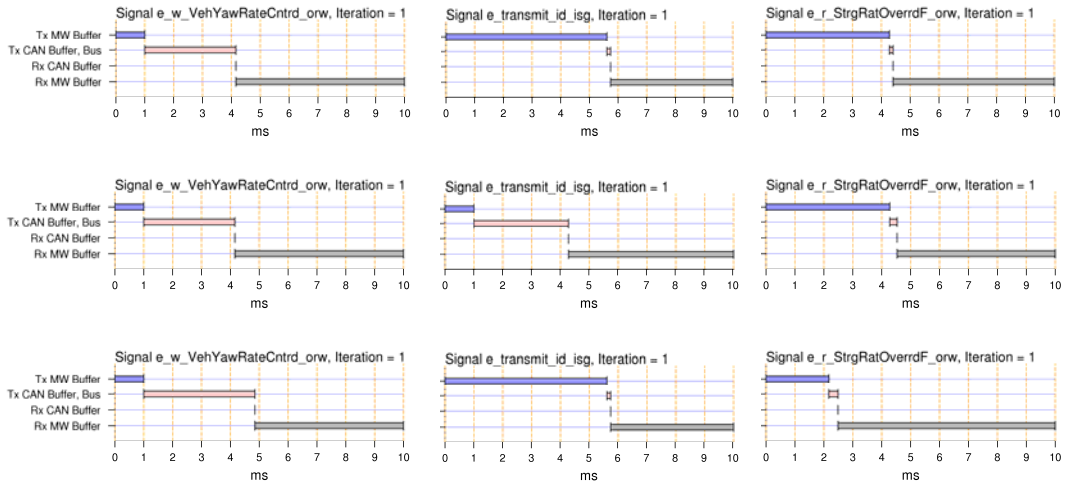


Figure 11 First mapping configuration (smart inertial sensor). The rows correspond to the three CAN configurations: one transmit buffer, two transmit buffers, and two transmit buffers with priority threshold, respectively.

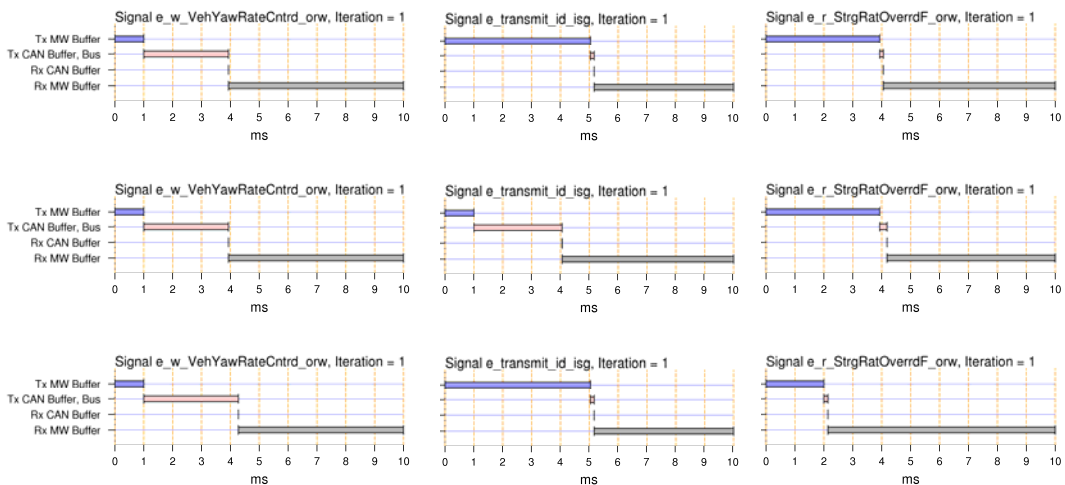


Figure 12 Second mapping configuration (hard-wired inertial sensor). The rows correspond to the three CAN configurations: one transmit buffer, two transmit buffers, and two transmit buffers with priority threshold, respectively.

CONCLUSIONS AND FUTURE WORK

In this paper we demonstrated with an industrial case study the value of having a virtual integration and exploration environment that is based upon formal semantics. We also showed that a consistent functional and architectural model of computation is required for an effective architectural exploration. Using the Metropolis metamodel simulation environment, we quickly uncovered a priority inversion issue in the present implementation of the software and hardware stack for the case study. This issue was mitigated by exploring different CAN Controller hardware setups that

had more than 1 transmit buffer configured. The simulation results reflected the improvement achieved and enabled us to trade off between buffer sizing and the amount of latency incurred by the different signals. Other exploration scenarios are possible and will be the object of further experiments.

Future work involves extending the design exploration to include different modeling styles. In particular, in this paper, we proposed a model of computation that reflects some of the non-idealities that are typical of the architectures being explored to make the convergence to a solid design faster. Alternatively, we plan to explore a different strategy: keep the functional model at the

ideal level, but implement appropriate communication adapters (protocols) to make sure that the assumptions made at the functional level are guaranteed to hold at the implementation level.

An interesting approach along these lines is represented by [8], where a synchronous reactive model of computation is mapped into a Globally Asynchronous Locally Synchronous (GALS) model that achieves better performance by relaxing the synchronous assumption for the global communication patterns. Adapters can be synthesized so that the behavior of the GALS system is equivalent to the original, ideal, synchronous reactive model.

ACKNOWLEDGMENTS

We would like to acknowledge the advice and guidance from GM ECI lab researchers, specifically Thomas Fuhrman, Arnold Millsap, Tom Forest and Prof. Ramesh from the GM India Science Lab.

REFERENCES

1. Sangiovanni-Vincentelli, A.L.: "Defining platform-based design," EEDesign, February, 2002
2. Lee, E.A., Sangiovanni-Vincentelli, A.L.: "A framework for comparing models of computations," IEEE Transactions on Computer Aided Design Integrated Circuits, 17(12):1217-1229, Dec. 1998.
3. Road vehicles - interchange of digital information – controller - area network (CAN) for high-speed communication – ISO 11898
4. Systems Architecture: The Empirical Way — Abstract, Graham R. Hellestrand, EMSOFT 2005
5. SystemC Reference Manual, Synopsys Inc, www.systemc.org
6. Klaus Havelund, John Penix, Willem Visser (Eds.): SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings. Lecture Notes in Computer Science 1885 Springer 2000, ISBN 3-540-41030-9
7. Mars Pathfinder Priority Inversion, <http://www-2.cs.cmu.edu/afs/cs/user/raj/www/mars.html>
8. A. Benveniste, B. Caillaud, L.P. Carloni, P. Caspi, A.L. Sangiovanni-Vincentelli, Heterogeneous Reactive Systems Modeling: Capturing Causality and the Correctness of Loosely Time-Triggered Architectures (LTTA), Proceedings of the Fourth International Conference on Embedded Software (EMSOFT), 2004
9. A. Millsap, T. Forest, GM internal report and personal communication.