Safe Automotive Software Development

Ken Tindell
LiveDevices
York, U.K.
ken.tindell@livedevices.com

Hermann Kopetz
Technische Universität
Wien, Austria
hk@vmars.tuwien.ac.at

Fabian Wolf
Volkswagen AG,
Wolfsburg, Germany
fabian.wolf@volkswagen.de

Rolf Ernst
Technische Universität Braunschweig,
Germany (organizer)
r.ernst@tu-bs.de

Abstract

Automotive systems engineering has made significant progress in using formal methods to design safe hardware-software systems. The architectures and design methods could become a model for safe and cost-efficient embedded software development as a whole. This paper gives several examples from the leading edge of industrial automotive applications.

1 Introduction

R. Ernst, Technische Universität Braunschweig

Automotive software is for a large part safety critical requiring safe software development. Complex distributed software functions and software function integration challenge traditional simulation based verification approaches. On the other hand, software functions have to match the high fault tolerance and fail safe requirements of automotive designs. To cope with this challenge, advanced research in automotive software has developed formal approaches to safe automotive design which could become a model for safe and cost-efficient embedded software development as a whole. The special session which is summarized in this paper includes contributions from most renowned experts in the field. The following section will outline the problems when integrating software IP from several sources on one electronic control unit and gives an example of a hardware-software solution to observe and enforce real-time behavior. Section 3 looks at the next level, distributed systems. Only at this level, the high safety requirements of autonomous vehicle functions such as X-by-wire can be met. The section introduces the concepts of fault-containment regions and error containment and introduces architectures and corresponding hardware-software control techniques to isolate defective parts. The presented time-triggered architecture (TTA) and the time-triggered communication protocol are used throughout the automotive and aircraft industries. The last section gives an example of a practical application of formal methods to software integration for combustion engine control.

2 The need for a protected OS in high-integrity automotive systems

K. Tindell, LiveDevices, York

Motivation

As ECU (Electronic Control Unit) software becomes more complex there is a need to run more than one major subsystem inside a single ECU (such as a mixture of high- and low-integrity software in an X-by-Wire system). This need gives rise to the demand for an ECU operating system that can provide protection between software components. The OS can isolate a fault in a software component and prevent that fault from propagating to another subsystem in the same ECU. The need for such a protected OS has been recognized by the automotive industry. The vehicle manufacture initiative of the HIS group (Herstellerinitiative Software) of Audi, BMW, DaimlerChrysler, Porsche and VW, lays out requirements for a protected OS, the two most interesting of which are:

- Protection must apply between applications (i.e. groups of tasks) as well as tasks. This is because a single application, developed by a single organization, will typically be composed of a set of tasks. Tasks need protection from each other, but they must also share data (typically via shared memory for performance reasons).
- The OS must be designed to tolerate malicious software. This means that a programmer with the deliberate intent of doing damage cannot cause another application in the ECU to fail. Although there are not major concerns about deliberate sabotage by a programmer, this requirement is here because software failures can be as intricate and subtle as if they were deliberately programmed. Thus the requirement to protect

1530-1591/03 $17.00 © 2003 IEEE

against all possible failures results in a more secure system.

Not only does protection need to be applied in the functional domain (e.g. controlled access to memory, I/O) but it also needs to be applied in the timing domain (e.g. tasks must not run for too long or too frequently). This timing requirement is necessary because the new approaches to ECU development are based on schedulability analysis – engineering mathematics that determine the worst-case response times of activities (tasks, interrupt handlers) in a real-time system. Such analysis is necessary for a complex ECU with an OS because the timing determinism of a rigid statically scheduled system is no longer present.

Protection in the timing domain entails that the data used by the schedulability analysis (e.g. task execution times) is enforced at run-time so that the timing analysis always remains valid. For example, if a task runs for too long then the OS needs to step in and stop it in order that lower priority (and potentially more critical) tasks that are in another application are not held out for longer than calculated in the schedulability analysis.

A final and very important set of requirements are imposed by the automotive industry on any use of electronics: component costs must be as low as possible. This means that the choice of hardware is very limited. Today only Texas Instruments with the TMS470 and Infineon with the Tricore TC17xx provide silicon to automotive requirements that also contains the necessary memory protection hardware.

The APOS research project

The Advanced Technology Group (ATG) of LiveDevices has been conducting a research project to look at these issues. It has developed a prototype OS called 'APOS' (ATG Protected OS) to meet the HIS requirements and to investigate the issues surrounding a protected OS (e.g. examining the additional overheads due to protection, the issues of OSEK OS compliance).

The following sections discuss the results of this project in more detail.

Architecture

The APOS kernel runs on the Tricore TC1775 from Infineon. It supports supervisor- and user-mode execution, which prevents normal tasks from executing privileged instructions and restricts these tasks to accessing memory to four pre-defined regions (private RAM, private ROM, application RAM, application ROM). The private RAM is used to store the task stack.

Access to the kernel is via API calls that are implemented with TRAP instructions (as for a conventional OS).

API calls check for permissions before completing. There are permissions assigned off-line to each task in the system that indicates which OSEK OS objects (tasks, resources, alarms, counters, etc.) that the task may access. Some critical API calls (such as the call to shutdown the OS and re-start the ECU) have special permissions checking so that only certain trusted tasks can make these calls.

The complete ECU is constructed from a set of applications (in turn composed of a set of tasks) by the ECU integrator. This is a critical and trusted role since one of the jobs of the system integrator is to assign permissions and levels of trust to different tasks. Another job of the system integrator is to perform the timing analysis of the complete system to ensure that all the tasks in all the applications meet their defined timing requirements (i.e. deadlines). Only the system integrator can do this since the schedulability analysis requires complete information about the timing behaviors of the application tasks.

Communication between tasks within the same application is typically via shared memory. But communication between applications is via OSEK COM intra-ECU messaging.

Execution time monitoring

Every task in APOS is assigned an execution time budget. A task begins running with a virtual stopwatch set to this time. The stopwatch then counts down while the task is running. If the task is pre-empted by another task (or interrupt handler) then the stopwatch is stopped, to be resumed when the task continues running. If the stopwatch reaches zero then the APOS kernel terminates the task. Thus a task that runs for too long is killed before it can disrupt the execution of other lower priority tasks (most especially those that might be in another application).

Execution times are measured in hardware clock ticks and stored as 32-bit integers. The TC1775 unit used in development is clocked at 20MHz; thus no task execution time budget in the system can be longer than 3.6 minutes. Since a task rarely runs for longer than a few milliseconds, this is adequate. One exception is the idle task: this task never terminates, and so will run for longer than 3.6 minutes. The APOS implementation addresses this problem by providing a special "refresh budget" API call. Only the idle task is permitted to make this call.

APOS also keeps track of the longest execution time observed for each task. This aids the developer in setting appropriate execution time limits in the case where the developer does not have access to tools that can calculate statically a worst-case execution time figure. The measurement feature can also guide the testing strategy to ensure that the software component tests do exercise the worst-case paths in the task code.

In addition to patrolling the total execution time of a task, the kernel also patrols execution times of the

task while holding a resource. In OSEK OS a resource is actually a semaphore locked and unlocked according to the rules of the priority ceiling protocol. This is done because the maximum blocking time of a higher priority task can be calculated. However, the time for which a task holds a given resource needs to be bounded (this time is used in the schedulability analysis) and hence enforced at run-time. The APOS kernel does this: when a task obtains an OSEK OS resource another virtual stopwatch is started and counts down. If the task fails to release the resource within the defined time then it is killed and the resource forcibly released. Unfortunately, this may lead to application data corruption (a typical use for a resource is to guard access to application shared data). However, since a failed task can in any case corrupt application shared data, it is important that the application be written to tolerate this failure (perhaps by having a soft restart option).

Execution pattern monitoring

The schedulability analysis performed by the system integrator not only uses the worst-case execution time figures for each task (and resource access patterns), but also the pattern in which the task (or interrupt handler) is invoked. The OS must enforce the execution patterns since any task or interrupt handler exceeding the bounds implied by the pattern may cause another task in another application to fail (and hence violating the strong protection requirements). There is therefore a strong coupling between enforcement and analysis: there is no point analyzing something that cannot be enforced, and vice versa.

Simple schedulability analysis requires a minimum time between any two invocations of a task or interrupt handler (i.e. the period in the case of periodic invocations). But in real systems there are often tasks that run with more complex patterns, particularly when responding to I/O devices. For example, an interrupt handler servicing a CAN network controller may be invoked in a 'bursty' fashion, with the short-term periodicity dictated by the CAN baud rate and the long-term periodicity a complex composite of CAN frame periodicities. Making the assumption that the interrupt handler simply runs at the maximum rate is highly pessimistic. Although the analysis can be extended to account for the more complex behavior, enforcing such arbitrary patterns efficiently in the OS is impossible. An alternative approach is taken by using two deferred servers for each sporadic task or interrupt handler. This approach provides two invocation budgets for each sporadic task and interrupt handler. If either budget is exhausted then no further invocations are permitted (for an interrupt handler the interrupts are disabled at source). The budgets are replenished periodically (typically with short- and long-term rates).

Performance and OSEK

Early figures for performance of the OS compared to a conventional OSEK OS indicate that the CPU overheads due to the OS are about 30-50% higher and the RAM overheads are about 100% higher. Given the very low overheads of an OSEK OS and the benefits of sharing the hardware across several applications, this is quite acceptable. Furthermore, no significant OSEK OS compatibility issues have been discovered.

Summary

In the near future automotive systems will require the ability to put several applications on one ECU. There are many technical demands for this but all are soluble within the general requirements of the automotive industry for low component cost.

3 Architecture of Safety-Critical Distributed Real-Time Systems

H. Kopetz, Technische Universität Wien

Computer technology is increasingly applied to assist or replace humans in the control of safety-critical processes, i.e., processes where some failures can lead to significant financial or human loss. Examples of such processes are *by-wire-systems* in the aerospace or automotive field or *process-control systems* in industry. In such a computer application, the computer system must support the safety, i.e., the probability of loss caused by a failure of the computer system must be very much lower than the benefits gained by computer control.

Safety is a system issue and as such must consider the system as a whole. It is an emergent property of systems, not a component property [1], p.151. The safety case is an accumulation of evidence about the quality of components and their interaction patterns in order to convince an expert (a certification authority) that the probability of an accident is below an acceptable level. The safety case determines the criticality of the different components for achieving the system function. For example, if in a drive-by-wire application the computer system provides only assistance to the driver by advising the driver to take specific control actions, the criticality of the computer system is much lower than in a case where the computer system performs the control actions (e.g., braking) autonomously without a possible intervention by the driver. In the latter case, the safety of the car as a whole depends on the proper operation of the computer system. This contribution is concerned with the architecture of safety-critical distributed real-time systems, where the proper operation of the computer system is critical for the safety of the system as a whole.

A computer architecture establishes a framework and a blueprint for the design of a class of computing systems that share a common set of characteristics. It sets up the computing infrastructure for the implementation of applications and provides mechanisms and guidelines to partition a large application into nearly autonomous subsystems along small and well-defined interfaces in order to control the complexity of the evolving artifact [2]. In the literature, a failure rate of better than 10^{-9} critical failures per hour is demanded in ultra-dependable computer applications [3]. Today (and in the foreseeable future) such a high level of dependability cannot be achieved at the component level. If it is assumed that a component—a single-chip computer—can fail in an arbitrary failure mode with a probability of 10^{-6} failures per hour then it follows that the required safety at the system level can only be achieved by redundancy at the architecture level.

In order to be able to estimate the reliability at the system level, the experimentally observed reliability of the components must provide the input to a reliability model that captures the interactions among the components and calculates the system reliability. In order to make the reliability calculation tractable, the architecture must ensure the independent failure of the components. This most important independence assumption requires fault containment and error containment at the architecture level. Fault containment is concerned with limiting the immediate impact of a fault to well-defined region of the system, the fault containment region (FCR). In a distributed computer system a node as a whole can be considered to form an FCR. Error containment is concerned with assuring that the consequences of the faults, the errors, cannot propagate to other components and mutilate their internal state. In a safety-critical computer system an error containment region requires at least two fault containment regions.

Any design of a safety-critical computer system architecture must start with a precise specification of the fault hypothesis. The fault hypothesis partitions the system into fault-containment regions, states their assumed failure modes and the associated probabilities, and establishes the error-propagation boundaries. The fault hypothesis provides the input for the reliability model in order to calculate the reliability at the system level. Later, after the system has been built, it must be validated that the assumptions which are contained in the fault hypothesis are realistic.

In the second part of the presentation it will be demonstrated how these general principles of architecture design are realized in a specific example, the Time-Triggered Architecture (TTA) [4]. The TTA provides a computing infrastructure for the design and implementation of dependable distributed embedded systems. A large real-time application is decomposed into nearly autonomous clusters and nodes and a fault-tolerant global time base of known precision is generated at every node. In the TTA this global time is used to precisely specify the interfaces among the nodes, to simplify the communication and agreement protocols, to perform prompt error detection, and to guarantee the timeliness of real-time applications. The TTA supports a two-phased design methodology, architecture design and component design. During the architecture design phase the interactions among the distributed components and the interfaces of the components are fully specified in the value domain and in the temporal domain. In the succeeding component implementation phase the components are built, taking these interface specifications as constraints. This two-phased design methodology is a prerequisite for the composability of applications implemented in the TTA and for the reuse of pre-validated components within the TTA. In this second part we present the architecture model of the TTA, explain the design rational, discuss the time-triggered communication protocols TTP/C and TTP/A, and illustrate how component independence is achieved such that transparent fault-tolerance can be implemented in the TTA.

4 Certifiable Software Integration for Power Train Control

F. Wolf, Volkswagen AG, Wolfsburg

Motivation

Sophisticated electronic control is the key to increased efficiency of today's automotive system functions, to the development of novel services integrating different automotive subsystems through networked control units, and to a high level of configurability. The main goals are to optimize system performance and reliability, and to lower cost. A modern automotive control unit is thus a specialized programmable platform and system functionality is implemented mostly in software.

The software of an automotive control unit is typically separated into three layers. The lowest layer are system functions, in particular the real-time operating system, and basic I/O. Here, OSEK is an established automotive operating system standard. The next higher level is the so-called ‚basic software'. It consists of functions that are already specific to the role of the control unit, such as fuel injection in case of an engine control unit. The highest level are vehicle functions, e.g. adaptive cruise control, implemented on several control units. Vehicle functions are an opportunity for automotive product differentiation, while control units, operating systems and basic functions differentiate the suppliers. Automotive manufacturers thus invest in vehicle functions to create added value.

The automotive software design process is separated into the design of vehicle software functions (e.g. control algorithms), and integration of those

functions on the automotive platform. Functional software correctness can be largely mastered through a well-defined development process, including sophisticated test strategies. However, operating system configuration and non-functional system properties, in particular timing and memory consumption are the dominant issues during software integration.

Automotive Software Development

While automotive engineers are experts on vehicle function design, test and calibration (using graphical tools such as ASCET-SD or Matlab/Simulink, hardware-in-the-loop simulation etc.), they have traditionally not been concerned with software implementation and integration. Software implementation and integration is usually left to the control unit supplier who is given the full specification of a vehicle function to implement the function from scratch. Some automotive manufacturers are more protective and implement part of the functions in-house but this does not solve the software integration problem.

This approach has obvious disadvantages. The automotive manufacturer has to expose his vehicle function knowledge to the control unit supplier who also supplies the manufacturer's competitors. It is hard to protect intellectual property in such an environment. Re-implementation of vehicle functions results in design cycles of several weeks. This inhibits design-space exploration and optimized software integration. Often the function returned by the system integrator does not fully match the required behavior resulting in additional iterations. From the automotive manufacturer's perspective, a software integration flow is preferable where the vehicle function does not have to be exposed to the supplier and where integration for rapid design-space exploration is possible. This can only be supported in a scenario where software functions are exchanged and integrated using object codes.

The crucial requirement here is that the integrated software must meet the stringent safety requirements for an automotive system in a certifiable way. These requirements generally state that it must be guaranteed that a system function (apart from functional correctness) satisfies real-time constraints and does not consume more memory than its budget. This is very different from the telecom domain where quality of service measures have been established. However, timeliness of system functions is difficult to prove with current techniques, and the problem is aggravated if software parts are provided by different suppliers. An important aspect is that even little additional memory or a faster system hardware that can guarantee the non-functional system correctness may simply be too costly for a high-volume product like an automotive control unit.

Certification of Multi-Source Systems

The focus is on a methodology and the resulting flow of information that should be established between car manufacturer, system supplier, OSEK supplier and system integrator to avoid the mentioned disadvantages and enable certifiable software integration. The information flow that is needed by the certification authority should be defined via formal agreements. Part of the agreements provides a standardized description of the OSEK configuration, process communication variables and scheduling parameters, such that intellectual property is protected and any system integrator can build an executable engine control. However, the novel key agreements should guarantee real-time performance and memory budgets of the integrated software functions. This requires suitable models for timing and resource usage of all functions involved: vehicle functions, basic software functions and system functions.

Commercial tool suites can determine process-level as well as system-level timing by simulation with selected test patterns. This approach lacks the possibility to explore corner-case situations that are not covered in the tests. It is known from real-time systems design that reliable system timing can only be achieved if properties of each function are described using conservative min-max intervals. Therefore, the enabling requirements for certifiable software integration are to obtain such conservative process-level intervals for all functions involved, including operating system primitives, and to apply suitable system-level analysis techniques to determine all relevant system timing.

Different methodologies for the determination of conservative process-level timing intervals exist. The main problem is the absence of mature, industry-strength tool suites that support certifiable timing analysis for complex automotive systems. The same applies to system-level timing analysis tool suites. So for today's automotive systems, a combination of simulation-based timing analysis with careful test pattern selections and formal approaches where applicable is feasible.

Conclusion

The need for distributed development of automotive software requires a certifiable integration process between car manufacturer and suppliers. Key aspects, i.e., software timing and memory consumption of the operating system, of the software functions provided by the control unit supplier as well as of the software functions provided by the car manufacturer have been identified. These need to be guaranteed for certifiable software integration.

5 References

1. Leveson, N.G., *Safeware, System Safety and Computers*. 1995, Reading, Mass.: Addison Wesley Company.

2. Simon, H.A., *Science of the Artificial*. 1981, MIT Press, Cambridge.

3. Suri, N., Walter, C.J. and Hugue, M.M., eds., *Advances in Ultra-Dependable Systems*. 1995, IEEE Press.

4. Kopetz, H. and Bauer, G., *The Time-Triggered Architecture.* Proceedings of the IEEE, 2003. 91 (January 2003).