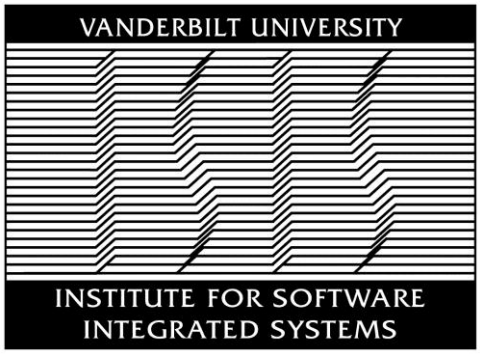

Institute for Software-Integrated Systems

Technical Report

TR#: **ISIS-15-101**

Title: **Component Models for Vehicle Software Platforms: Two Case Studies**

Authors: **Daniel Balasubramanian, Gabor Karsai**

Component Models for Vehicle Software Platforms: Two Case Studies

Daniel Balasubramanian

Abstract: This report (1) presents use cases and requirements for a vehicle information architecture platform (VIAP), (2) reviews and evaluates the Automotive Open System Architecture (AUTOSAR) and the Distributed Real-time Managed System (DREMS) architecture specifications, and (3) presents a preliminary architecture specification VIAP that addresses the needs of the DARPA Adaptive Vehicle Make program.

This work was supported by the DARPA AVM Program under contract HR0011-13-C-0041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA.

Contents

Requirements for a Vehicle IAP

A Vehicle Information Architecture Platform (VIAP) is a software platform for running mission-relevant software applications on the embedded computers of a vehicle. Mission critical applications may include navigation applications, management of digital communications, sensor management and processing, storage management, and many others. The list below summarizes high-level requirements for such a platform. Note that these requirements have been influenced by earlier work on the DARPA System F6 Program where a similar Information Architecture Platform was designed for fractionated satellites.

1. Support a full spectrum of modern software application architecture and design approaches

The VIAP should permit application developers to use a wide-range of design and implementation techniques, particularly with respect to existing and anticipated developments in hardware and software platforms. Techniques such as massive parallel processing, virtual machines, application controlled memory management, garbage collection, modern implementation languages (including but not limited to functional, actor-based and object-oriented languages), component-based software development, model-based code generation, etc. should be usable on the platform.

2. Enable efficient use of system resources

The VIAP should enable the efficient use of all low-level platform resources, including: (1) processing (e.g., CPU time), (2) dynamic memory (e.g., RAM), (3) persistent storage (e.g., file space), (4) communication (e.g., network bandwidth), and (5) system services (provided by the VIAP).
Efficient use means that the resources are well-utilized (nothing is over- or under-utilized), and that effective resource management services are available on the platform to make this feasible.

3. Enable flexible use of system resources

The VIAP should enable the dynamic change and adjustment of resource usage, so that the system can respond effectively and, to the degree possible, autonomously, to changes in application needs, system configuration and/or operating environment.

4. Provide a robust, flexible, and manageable software infrastructure

The software platform has to be robust and dependable, but it also has to permit the configuration and management of the software applications. Mission software can and will change over time, so a robust administration interface is required to facilitate these software configuration changes.

4. Enable controlled sharing of system resources

The VIAP should enable controlled sharing of low-level platform resources among applications potentially running on multiple security levels. Shared resources include (1) processing (e.g., CPU time), (2) dynamic memory (e.g., RAM), (3) persistent storage (e.g., files, flash memory), (4) communication (e.g., network

bandwidth), (5) hardware devices (e.g., special physical interface devices for sensors, etc.), and (6) system services (provided by the VIAP). The purpose of the controlled sharing is to ensure non-interference among applications and compliance with security policies.
The VIAP should provide flexible administrative mechanisms to ensure control of both the static and dynamic allocation of system resources.

5. Provide operational flexibility and maintainability

The VIAP should provide flexibility for the operation and maintenance of itself and of applications that it hosts. This requirement calls for features to support, including, but not limited to: system and application debugging, on-line and post-mortem fault diagnosis, analysis of system and application operating state, analysis of system and/or application state after a failure, tracing and inspection of system and application activities, and replacement and update of system and application software components.

6. Provide a comprehensive framework for fault management

A fault management framework should enable accommodation of arbitrary inputs from within and from outside of the system, diagnostic procedures, and response mechanisms, supporting both manual and autonomous fault responses.
The fault management framework should accommodate fault detection and response for the VIAP itself, for system and vehicle special hardware, and faults and responses within applications. The fault management framework is required to provide resilience; mission-critical system functions are expected to be restored even if hardware and software fail.

7. Provide a comprehensive framework for quality of service and resource management

A QoS framework provides a system-wide view for resource definition, allocation and management. It provides for autonomous responses to changing resource needs or system resource availability. It should provide strong predictability to mission application designers and operators, enabling rigorous service level guarantees.
Resources that can participate in a QoS framework include: (1) network resources (e.g. bandwidth), (2) communication quality (e.g., latency, jitter, and reliability), (3) processing resources (e.g. CPU utilization), (4) dynamic memory, (5) persistent storage, (6) special vehicle hardware, (7) system services.

8. Support real-time processing within a computing node

The VIAP should provide well-defined and predictable real-time properties for all its functions, and should provide scheduling mechanisms (e.g., for processing and communication) that ensure predictable performance in response to explicitly defined real-time requirements within a computing node. Within a computing node, real-time operation is constrained primarily by processor scheduling.

9. Support real-time processing across the distributed computing platform

The VIAP should provide well-defined and predictable real-time properties for all its functions, and should provide scheduling mechanisms (e.g., for processing and

communication) that ensure predictable performance in response to explicitly defined real-time requirements, across the distributed computing platform (that includes vehicle local area networks). Within the network, real-time operation is constrained primarily by network bandwidth and latency.

10. Provide security mechanisms sufficient to satisfy system security requirements

The platform should support applications running side-by-side that are of different security levels. Isolation between levels and the prevention of unauthorized information flows is of utmost relevance. The platform should provide assurances for robustness and resistance to attacks, including ones introduced by untrusted applications.

Overview of AUTOSAR

The Automotive Open System Architecture (AUTOSAR) is an open and standardized automotive software architecture jointly developed by manufacturers, suppliers and tool developers working in the automotive industry. Its primary objective is to create standards for automotive electrical/electronic (E/E) architectures that provide the basic infrastructure to assist with developing vehicle software, user interfaces and management functionality. AUTOSAR can be seen as an infrastructure that supports many standards.

The stated goals of AUTOSAR include the following:

- Standardize basic software functionality of ECUs.
- Reduce complexity of the heterogeneous software landscape within cars.
- Implementation and standardization of basic system functions as an OEM wide "standard core" solutions.
- Scalability to different vehicle and platform variants.
- Transferability of software.
- Definition of an open architecture.

The motivation for AUTOSAR is the following:

- Management of complexity associated with increase in functionality.
- Flexibility for product modification, upgrading and updating.
- Improved quality and reliability of E/E systems.

The main intention of AUTOSAR is a common standard for the layer of software in automobiles that is invisible to end-users, with the objective of creating a basis for industry collaboration on basic functions and competition on innovative functions. One of the basic sayings of AUTOSAR is to "collaborate on standards, compete on implementations."

The Run-Time Environment (RTE)

At system design level, (i.e. when drafting a logical view of the entire system irrespective of hardware) the AUTOSAR Runtime Environment (RTE) acts as a communication center for inter- and intra-ECU information exchange. The RTE provides a communication abstraction to AUTOSAR Software Components attached to it by providing the same interface and services regardless of whether inter-ECU communication channels are used (such as CAN, LIN, FlexRay, MOST, etc.) or communication stays intra-ECU. As the communication requirements of the software components running on top of the RTE are application dependent, the RTE must be tailored, partly by ECU-specific generation and partly by configuration. Thus, the resulting RTE will differ between one ECU and another due to the fact that it is partly generated and also configured for specific ECUs. How the RTE is realized in the run-time system differs between implementations (it could, for instance, be realized as a lightweight middleware).

The RTE provisions inter- and intra-ECU communication across all nodes of a vehicle network and is located between the functional SW-components and basic SW-modules. It also enables integration of customer specific functional SW modules. Figure 1 below shows how the RTE fits in with the other software modules and components in an AUTOSAR system.

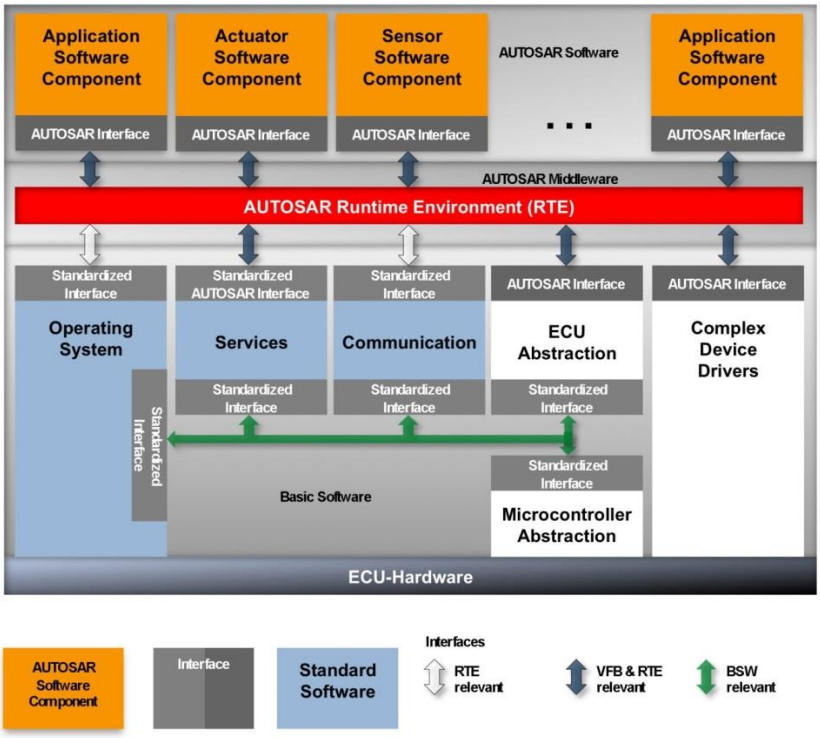

Figure 1 - The AUTOSAR RTE and Overall Architecture.

AUTOSAR Basic Software (BSW)

The Basic Software (BSW) is standardized software layer which provides services to AUTOSAR software components. As Figure 1 above shows, the BSW layer sits below the RTE and contains standardized and ECU specific components. The BSW layer includes modules for primitive operations, such as communication, and is used by higher-level components. The BSW contains components that are standardized across all systems, as well as components that are specific to an ECU. Examples of the standardized components of the BSW include:

- System services (NVRAM, flash, memory management)
- Communication management (FlexRay, CAN, LIN), I/O management, Network management
- OS specifications (see below)
- Microcontroller abstractions (see below)

ECU specific components includes:

- *ECU abstraction* to provide a software interface to electrical values of any ECU.
- *Complex Device Driver* (CDD) allows direct access to hardware for resource critical applications.

AUTOSAR Operating System

AUTOSAR specifies only the requirements for an operating system (OS), and thus any OS that satisfies its OS specification can be used on the ECU, including proprietary OSs. Any OS must be abstracted to an AUTOSAR OS by providing the interfaces and services listed in the AUTOSAR OS Specification. The Standard OSEK OS (ISO 17356-3) is used as the basis for the AUTOSAR OS. The following are example requirements for the OS:

- Must be configured and scaled statically.
- Must be amenable to reasoning about real-time performance.
- Must provide priority-based scheduling.
- Must provide protective functions at run-time.
- Must run on low-end controllers and without external resources.

Even though the AUTOSAR OS does not explicitly state which OS must be used, the majority of the implementations seem to use a derivative/modification of the OSEK OS.

AUTOSAR Microcontroller Abstraction Layer

Access to the underlying hardware is routed through a layer called the Microcontroller Abstraction Layer (MCAL) to avoid direct access to registers from high-level software (see Figure 1). MCAL is a hardware-specific layer that provides a standard interface to the Basic Software (BSW). This interface is then used by the BSW layer to query specific information from the underlying microcontroller. This allows the BSW to provide higher-level components with microcontroller independent values.

The Microcontroller Abstraction Layer can provide several capabilities, including Digital I/O, Analog/Digital Conversion, Pulse Width Modulation, Flash, Watchdog Timers and I^2C Bus interfacing.

AUTOSAR Applications

An application in AUTOSAR consists of interconnected Software Components (SWCs). These can be seen at the top of Figure 1. AUTOSAR makes a clear distinction between application and infrastructure. Each instance of an AUTOSAR software component is assigned to one ECU, and in this sense they are atomic. The communication is described at a very abstract level called the "Virtual Function Bus" (VFB). Components communicate through ports with no knowledge of the communication path. The Run-Time Environment (RTE) implements the VFB on the ECU. The RTE gives an interface that is bus-independent and issues commands to

the basic software of the ECU. The basic software can then access the hardware directly.
To create an executable ECU component, there are series of steps. XML is used as the interchange format (the XML is based on a schema derived from the AUTOSAR UML model). The schema file consists of 4 basic parts:

1. Software component template (defines individual software components).
2. Basis software module description template (describes all information about a Basis Software Component).
3. ECU configuration template (describes architecture and interfaces of ECU).
4. System template (defines overall system). Stores information about bus systems, signals, mapping and topology. Shares commonality with FIBEX standard.

Encryption in AUTOSAR

AUTOSAR supports security by providing the Crypto Security Manager (CSM), which is an integral part of the system. The CSM provides an abstraction layer in the form of a standardized interface that gives access to basic cryptographic functionalities for all software modules. Software modules that need access to cryptographic functionality can configure and initialize the CSM for their specific needs, such as synchronous or asynchronous processing.
The current specification of the CSM does not place a requirement on where the CSM executes in relation to the software components that use it. This permits, for example, the CSM running on one ECU to be used by software components running on different ECUs. This may have security implications because the data a software component wants to encrypt must first travel to the SCM over a communication channel that is not necessarily encrypted.
Currently, encryption is done in software, but there is on-going work to implement the encryption functions in hardware which would improve the security (because the private keys are then stored in a way that prevents external access). A hardware solution would also increase performance. Hardware support for encryption will enable the encryption of all on-board data communications across CAN, FlexRay and Ethernet bus systems. Currently, messages across the communication busses carry a signature but are not encrypted. The encryption will massively restrict the access to internal data buses and ECUs. This is currently one of the big topics for OEMs and tier ones.
Data in the system may be encrypted for a variety of reasons. Manufacturers may wish to encrypt proprietary data to prevent third-party vendors from reverse engineering their protocols. As automobiles integrate more multimedia functionality and connectivity to devices such as cell-phones and tablet computers, data may need to be encrypted to keep it safe from malicious network attacks. This is especially true as the use of Car-to-X is being integrated into cars.
Reference:
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_SWS_CryptoServiceManager.pdf

TCP/IP in AUTOSAR

TCP/IP is specified in version 4.1.1 of the AUTOSAR specification. The entire communications stack is specified according to AUTOSAR, which means it can be configured using the AUTOSAR methodology. Although the specification does not prescribe a certain physical layer or data rate, currently only Ethernet over wired-LAN is being considered (the upcoming BMW 7 Series will be the first vehicle to integrate Ethernet). The benefit of Ethernet is that it is very mature. Even though such a communication stack requires more hardware resources than, for example, CAN communications (for both computing power and in particular for RAM), the benefit is that it provides a high-performance data network technology suitable for future applications, such as multimedia applications and Car-to-X.
Reference: http://www.AUTOSAR.org/download/R4.1/AUTOSAR_SWS_TcpIp.pdf

Memory Protection with AUTOSAR

Memory protection prevents a process from accessing memory that has not been allocated to it. This helps prevent a memory bug in one process from affecting other processes or the operating system. For instance, memory protection can prevent one process from accessing the memory stack of another process.

Section 7.7.1 of the AUTOSAR OS specification states that memory protection will only be possible on processors that provide hardware support for memory protection. The memory protection scheme is based on the data, code and stack sections of the executable program. However, because the AUTOSAR OS Specification is based on OSEK and OSEK systems are expected to run on chips without memory protection, it is not unexpected for AUTOSAR to assume that memory protection will not be available.

Error reporting with AUTOSAR

Error reporting in AUTOSAR is enabled through the Diagnostic Event Manager (Dem) service, which is a component that processes and stores *Diagnostic Events* (errors) and associated data. A Diagnostic Event defines the atomic unit that can be handled by the Dem module. The Dem handles and stores events detected by *diagnostic monitors* in both Software Components (SWCs) and Basic Software (BSW).
A diagnostic monitor is a routine entity that determines whether a component is functioning properly. The diagnostic monitor provides monitoring that identifies a specific fault type (for example, short to ground, open load) for a *monitoring path*. A monitoring path represents the physical system or a circuit that is being monitored. Each monitoring path is associated with exactly one diagnostic event. A diagnostic monitor is implemented as a piece of code in a SWC (or in the BSW) that communicates with the Dem using AUTOSAR standard communication ports.
Each Diagnostic Event has an associated priority that ranks the event based upon its level of importance and determines whether its fault entry may be removed from the event memory of the Dem in case the event memory is full. The Dem also stores

an occurrence counter per event memory entry (with a maximum limit of 255, after which the counter stays at 255). Each Diagnostic Event falls under one of two types of significance levels that is configurable per event:

- Fault: classifies a failure that relates to the component or ECU itself and requires, for example, a repair action.
- Occurrence: Classifies an issue which is not a fault, but which indicates insufficient system behavior. This may relate to an condition out of the ECU's control.

While the Diagnostic Event Manager is the entity that stores diagnostic events, the Diagnostic Communication Manager (Dcm) Software module is what provides a common API for diagnostic services. The DCM module is what is used by external diagnostic tools during development, manufacturing or maintenance and servicing. A *Diagnostic Trouble Code* (DTC) defines a unique identifier that is shown to the diagnostic tester. This unique identifier is mapped to a Diagnostic Event of the Dem module. The Dem then provides the status of the DTC to the Dcm.
The Dem module supports DTC standardized formats, including ISO 14229-1, ISO 15031-6, SAE J1939-73 and ISO 11992-4. DTC groups (as opposed to single DTC values) are also supported. The AUTOSAR provides DTC groups for powertrain, chassis, body and network communication codes.
References:
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_SWS_DiagnosticCommunicationManager.pdf
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_SWS_DiagnosticEventManager.pdf (especially Sections 7.1 and 7.1.6).

Timing Specification with AUTOSAR

Beginning with version 4.0, AUTOSAR provides Timing Extensions that provide the basic means to describe and specify timing information: Timing descriptions, expressed by events and event chains, and timing constraints that are imposed on these events and event chains. Both means, timing descriptions and timing constraints, are organized in timing views for specific purposes. The timing extensions serve two main purposes. The first is to provide timing requirements that guide the construction of systems which eventually shall satisfy those timing requirements. The second purpose is to provide sufficient timing information to analyze and validate the temporal behavior of a system.
Events refer to locations in systems at which the occurrences of events are observed. The AUTOSAR Specification of Timing Extensions defines a set of predefined event types for such observable locations. Those event types are used in different timing views and each of these timing views correspond to one of the AUTOSAR views: VFB Timing and Virtual Function Bus VFB View; SW-C Timing and Software Component View; System Timing and System View; BSW Module Timing and Basic Software Module View; as well as ECU Timing and ECU View.
In particular, one uses these events to specify the reading and writing of data from and to specific ports of software components, calling of services and receiving their

responses (VFB, SW-C, System and ECU Timing); sending and receiving data via networks and through communication stacks (System and ECU Timing); activating, starting and terminating executable entities (SW-C Timing and Basic SW Module Timing); and last but not least calling basic software services and receiving their responses (ECU Timing and Basic SW Module Timing).

An Event Chain describes the temporal correlation between two observable events (referred to as the stimulus and response) that have a functional dependency and contains a timing constraint. Event chains can be built-up in hierarchies. The notion of an event chain enables one to specify the relationship between two events, for example when an event A occurs then the event B occurs, or in other words, the event B occurs if and only if the event A occurred before. In the context of an event chain, the event A plays the role of the *stimulus* and the event B plays the role of the *response*. Event chains can be composed of existing event chains and decomposed into further event chains; in both cases, the event chains play the role of event chain segments.

The notion of an Event is used to describe that specific events occur in a system and at which locations in this system the occurrences are observed. In addition, an Event Triggering Constraint imposes a constraint on the occurrences of an event, which means that the event triggering constraint specifies the way an event occurs in the temporal space. The AUTOSAR Specification of Timing Extensions provides means to specify periodic and sporadic event occurrences, as well as event occurrences that follow a specific pattern (burst, concrete and arbitrary patterns).

Latency and synchronization timing constraints impose constraints on event chains. With timing constraints on Events, the constraint is used to specify a reaction and age, for example if a stimulus event occurs then the corresponding response event shall occur not later than a given amount of time. For timing constraints on Event Chains, the constraints are used to specify that stimuli or response events must occur within a given time interval (tolerance).

In addition to the timing constraints that are imposed on Events and Event Chains, the AUTOSAR Timing Extensions provide timing constraints which are imposed on Executable Entities, namely the Execution Order Constraint and Execution Time Constraint.

There are five distinct timing views, each associated with a particular AUTOSAR view:

- Virtual Function Bus: describes timing information related to the interaction of software components at the VFB level. Typically captures end-to-end timing constraints, including physical sensors and actuators. Does not refer to the internal behavior of a SWC. Can express timing constraints such as, “From the point in time when a value is received on an input port, at most 2ms can elapse before a value is produced on an output port.”
- Software Component (SWC): describes timing information related to the internal behavior of a Software Component. This timing description can refer to the activation, start and termination of the execution of Runnable Entities

within a SWC (the description of the internal behavior of a SWC is broken down into Runnable Entities, which are executed at runtime).

- System: describes timing information at a system level using information about topology, software deployment and signal mapping. This view allows timing to refer to the concrete communication of software components, which is either local communication over the RTE if the components are on the same ECU or remote communication over the RTE if the components are on different ECUs.
- Basic Software Module: describes timing information about a single Basic Software Module. This view is similar to the timing view for Software Components described above, and the timing description also refers to the activation, start and termination of Runnable Entities within the Basic Software Module.
- ECU: describes timing information that can reference all ECU-relevant information, including the deployed software component instances and ECU related interactions (such as bus communication or Basic Software interactions). This timing view has the same expressivity as the System Timing view, but only focuses on one specific ECU.

To summarize: the timing description for the entities in a system is specified by describing Events or Event Chains (which specify some observable behavior) and then describing timing constraints on those Events or Event Chains. This timing description is a requirement on the eventual implementation of the system and should not necessarily be considered as describing characteristics of the actual implementation (unless the actual implementation of the system has somehow been proven or shown to meet the requirements). The specification does acknowledge that the worst-case execution time (WCET) of a SWC, which is specific to a CPU, is needed to perform timing assessments. This is enabled by having the SWC template specify the timing-requirements of each runnable entity of a SWC, including the period (how often it has to be run) and the reaction time (time between stimulus and response). However, the specification does not specify a methodology for checking either timing constraints of individual software components or global timing properties of an integrated system.

Timing Protection with AUTOSAR OS

In regards to the timing protection of the AUTOSAR OS, a timing fault is defined as what occurs when a task or interrupt in a real-time system misses its deadline at runtime. The AUTOSAR OS does not offer deadline monitoring for timing protection due to the fact that it is insufficient to identify the task/ISR causing the timing fault in the system. Instead, whether a task or ISR meets its deadline in a fixed priority preemptive system like AUTOSAR OS is determined by:

1. Execution time of the tasks/ISRs in the system

2. The blocking time that tasks/ISRs suffer from lower priority tasks/ISRs locking shared resources or disabling interrupts.
3. The interarrival rate of tasks/ISRs in the system.

The system must control these three factors at runtime for timing protection. AUTOSAR prevents (1) by using *execution time protection*. Execution time protection here consists of first statically configuring an upper bound, called an execution budget, on the execution time of tasks and category 2 ISRs (a category 2 ISR is supported by the OS and can make OS calls; a category 1 ISR is not supported by the OS and is only allowed to make a very small selection of OS calls to enable and disable all interrupts). At run-time, the OS monitors the time that a task or category 2 ISR executes and preempts the task if its execution time exceeds its statically configured execution budget.
AUTOSAR OS prevents (2) by using *locking time protection* to guarantee a statically configured upper bound (called the Lock Budget) on the time that resources are held by tasks/category 2 ISRs, OS interrupts are suspended, and all interrupts are suspended.
AUTOSAR OS prevents (3) by using inter-arrival time protection to guarantee a statically configured lower bound (called the Time Frame) on the time between (1) a task being permitted to transition into the ready state, and (2) a category 2 ISR arriving.
Section 1.1 of the Time Service specification describes the Time Service module, an AUTOSAR Basic Software module, that is part of the System Services Layer and provides services for time-based functionality. It can be used for time measurement, time based state machines (state changes based on time can be implemented), timeout supervision and busy waiting (can use predefined "Predef" Timers instead of loops or no-op instructions to implement timeout supervision or busy waiting). It can be used to measure the execution time and cycle time of code, including the run time and cycle time of tasks, ISRs, functions and pieces of software. *However*, section 4.2 (Limitations) states that functionality of the time service module is based on hardware timers provided by the GPT (General Purpose Timer) driver. The specification defines no standardized AUTOSAR interfaces, meaning that the services of the Time Service module are not accessible by AUTOSAR SWCs located above the RTE; a standardized interface may be added in the future.
References:
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_TPS_TimingExtensions.pdf
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_RS_TimingExtensions.pdf
http://www.bmw-carit.com/projects/AUTOSAR-timing-specification.php
http://www.AUTOSAR.org/download/R4.1/AUTOSAR_SWS_TimeService.pdf

Tool support for AUTOSAR

The majority of tools supporting the development of AUTOSAR-based systems have a commercial license. Four specific tools are described below; there are several other commercial tools that are available to the public, as well as proprietary tools developed by manufacturers for their own internal use.

Mentor Graphics offers a tool suite called Volcano VSx for top-down vehicle system and ECU design. It covers automotive software and electronic systems design, virtual verification, testing and configuration. It supports standard AUTOSAR import/exports. Timing analysis of communications is supported with the Volcano VSA COM Designer tool. They offer a Basic Software (BSW) stack, as well as a Bootloader, RTE, customized diagnostics and customized software components. They claim their BSW stack offers predictable real-time network behavior, efficiency (low memory use, fast execution time, and small code size), easy portability and high quality.

ARCCORE offers a GPL license for Arctic Core and the base version of Arctic Studio tools. They report that a global Tier1 supplier headquartered in Japan is using the ARCCORE AUTOSAR 4.x solutions (Arctic Core and Arctic Studio) to develop new ECUs. The Arctic Core Standard Package includes features required in an ECU, including communication, diagnostics, safety services and an RTOS. The tools support PowerPC and ARM architectures, and more can be added for additional cost. The communication protocols support include TCP/IP, LIN services and CAN services; FlexRay is not advertised as being available out of the box.

MathWorks is an AUTOSAR premium member and actively participates in development of the standard with a focus on how to use model-based development within an AUTOSAR development process. Simulink and Embedded Coder allow engineers to import and export AUTOSAR software component descriptions and generate AUTOSAR production code in an integrated environment. Simulink provides support through model configuration settings rather than AUTOSAR specific blocks, allowing a single model to be used as a reference for simulation, prototyping and production code generation in both AUTOSAR and non- AUTOSAR environments. Advanced capabilities for AUTOSAR applications are provided through the AUTOSAR Target Production Package (ATPP), which may be requested at the following site: http://www.mathworks.com/matlabcentral/answers/97870-are-AUTOSAR-advanced-production-capabilities-available-for-simulink-and-embedded-coder.

The current workflow for using Simulink for AUTOSAR development includes the following steps.

1. Use an AUTOSAR Authoring Tool to design the software architecture of the vehicle functionality.
2. Export the software component description files using the AUTOSAR .arxml format.
3. Import these software component description files into Simulink, which will automatically generate a skeleton model of the interfaces and internal behavior defined in the software component description.
4. To generate AUTOSAR compliant code, finalize and validate the AUTOSAR Configuration using the Simulink AUTOSAR Mapping Editor.
5. AUTOSAR compliant code and corresponding .arxml files can then be generated directly from Simulink. The generated components are then ready for integration into the ECU.

BMW has a tool called *Artime* that is a textual editor to create text-based timing models that comply with the AUTOSAR Timing Extensions. *Artime* is a domain-specific language built on top of the *AUTOSAR Tool Platform* (Artop), a base platform for developing AUTOSAR tools, as well as *ARText*, a framework for building textual modeling languages for AUTOSAR. The level of support this tool provides for automatically checking whether timing requirements can/are satisfied is unclear.

Vehicles using AUTOSAR

While the adoption of AUTOSAR was primarily found in high-end automobile manufacturers in its early years, several automobile makers now use it in at least some components of their cars. *BMW* originally introduced AUTOSAR ECUs in their 7 Series and now uses it in all of their product lines. The next 7 Series *BMW* will fully use version 4 of the AUTOSAR specification and will be the first car to introduce Ethernet as a data backbone in vehicles.

The latest *Mercedes* S-Class used AUTOSAR in about 70 ECUs, which is considered fairly large scale. Other lines besides the S-Class will use AUTOSAR in the future.

Robert Bosch is now using AUTOSAR in all markets and integrating it into all relevant vehicle domains.

Toyota had their first ECU with AUTOSAR-based software as of 2013 and expects to gradually transition to a heavier use of AUTOSAR over time.

PSA Peugeot Citroen states that all engine management systems and body controllers now run on AUTOSAR ECUs.

Reference:

http://www.AUTOSAR.org/download/media_release/Ten_Years_of_AUTOSAR_EN.pdf

DREMS: A Distributed Real-time Embedded Managed Systems Software Platform Specification

DREMS is a software infrastructure specification for designing, implementing, configuring, deploying and managing distributed real-time embedded systems[1] that describes two major subsystems: (1) a *design-time toolsuite* for modeling, analysis, synthesis, implementation, debugging, testing, and maintenance of application software built from reusable components, and (2) a *run-time software platform* for deploying, managing and operating application software on a network of computing nodes. The DREMS specification is primarily targeted towards platforms that provide a *managed* network of computers and distributed software applications running on that network; in other words, a cluster of networked nodes.

The *design-time toolsuite* specification is naturally supported by a model-based paradigm of software development for distributed, real-time, embedded systems where modeling tools and generators automate the tedious parts of software development and also provide a design-time framework for the analysis of software systems. The *run-time software platform* specification is to reduce the complexity and increase the reliability of software applications by describing reusable technological building blocks in the form of an operating system, middleware, and application management services.

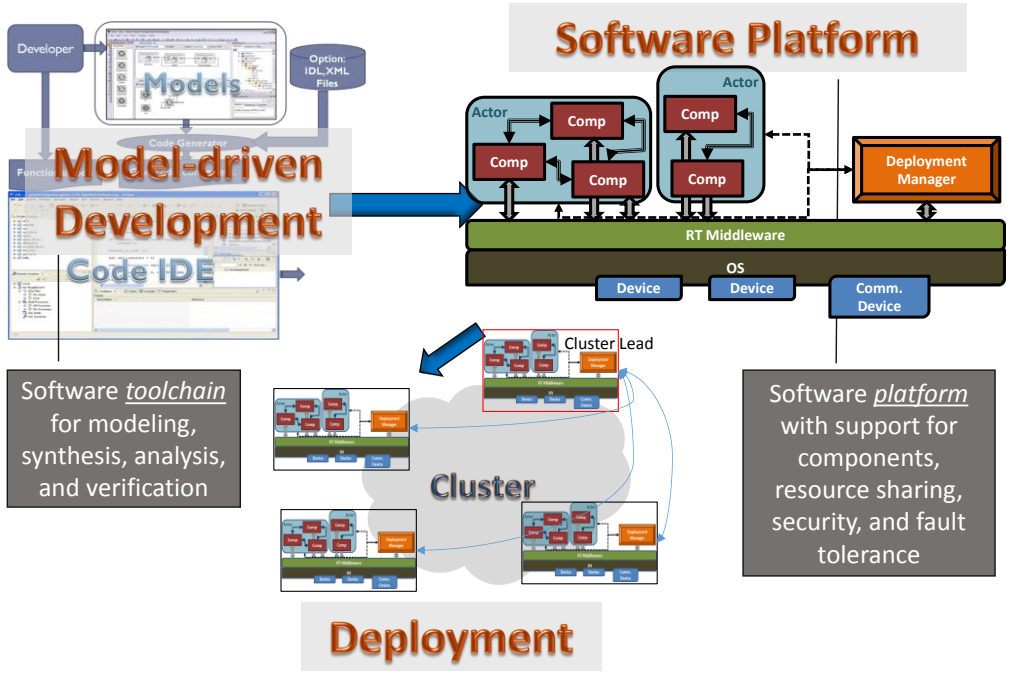

Figure 2 - DREMS Architecture

[1] DREMS was supported by the DARPA System F6 Program under contract NNA11AC08C through NASA ARC.

A system implementing the DREMS specification has the ability to provide a complete, end-to-end solution for software development: from modeling tools to code to deployed applications. DREMS focuses on the architectural issues of the software, and promotes the modeling of application software, where the models are directly used in the construction of the software.

The sections below describe the high-level requirements of the DREMS specification for Information Architecture Platforms.

Applications in DREMS

Software applications running on the DREMS platform shall be distributed: an application can consist of one or more *actors* that run in parallel, typically on different nodes of a network. Actors specialize the concept of processes: they have identity with state, they can be migrated from node to node, and they are managed. Actors are created, deployed, configured, and managed by a special service of the run-time platform: the deployment manager – a privileged, distributed, and fault tolerant actor, present on each node of the system, that performs all management functions on application actors. An actor can also be assigned a limited set of resources of the node it runs on: memory and file space, a share of CPU time and a share of the network bandwidth.

Applications shall be built from *software components* – hosted by actors – that interact via *only* well-defined interaction patterns using security-labeled messages, and are allowed to use a specific set of low-level services provided by the operating system. The low-level services include messaging and thread synchronization primitives, but components use these indirectly through the middleware libraries.

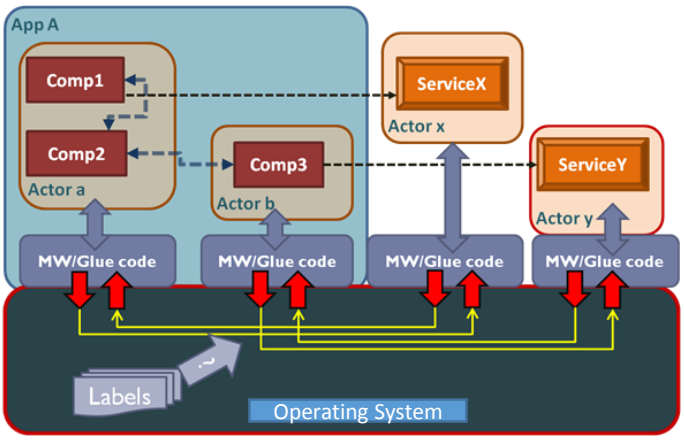

Figure 3 - DREMS applications, actors, components and services

The middleware libraries shall implement the high-level communication abstractions: synchronous and asynchronous interactions, on top of the low-level services provided by the underlying distributed hardware platform. Interaction patterns shall include (1) point-to-point interactions (in the form of synchronous and asynchronous remote method invocations), and (2) group communications (in the form of asynchronous publish-subscribe interactions). Component operations can be event-driven or time-triggered, enabling time-driven applications. Message exchanges via the low-level messaging

services are time-stamped, thus message receivers are aware of when the message was sent. Hence temporal ordering of events can be established (assuming the clocks of the computing nodes are synchronized).

Specialized, verified *platform actors* shall provide system-wide high-level *services:* application deployment, fault management, controlled access to I/O devices, etc. Each application actor exposes the interface(s) of one or more of its components that the components of applications can interact with using the same interaction patterns. Applications can also interact with each other the same way: exposed interfaces and precisely defined interaction patterns.

The DREMS Operating System shall implement all the critical low-level services to support resource sharing (incl. spatial and temporal partitioning), actor management, secure (labeled and managed) information flows, and fault tolerance. A key feature of the OS layer is support for temporal partitions (similarly to the ARINC-653 standard): actors can be assigned to a fixed duration, periodically repeating interval of the CPU's time so that they have a guaranteed access to the processor in that interval. In other words, the actors can have an assured bandwidth to utilize the CPU and actors in separate temporal partitions cannot inadvertently interfere with each other via the CPU. The DREMS Operating System specification provides the possibility for several types of implementations, such as a set of extensions to the Linux kernel or possibly using a microkernel approach.

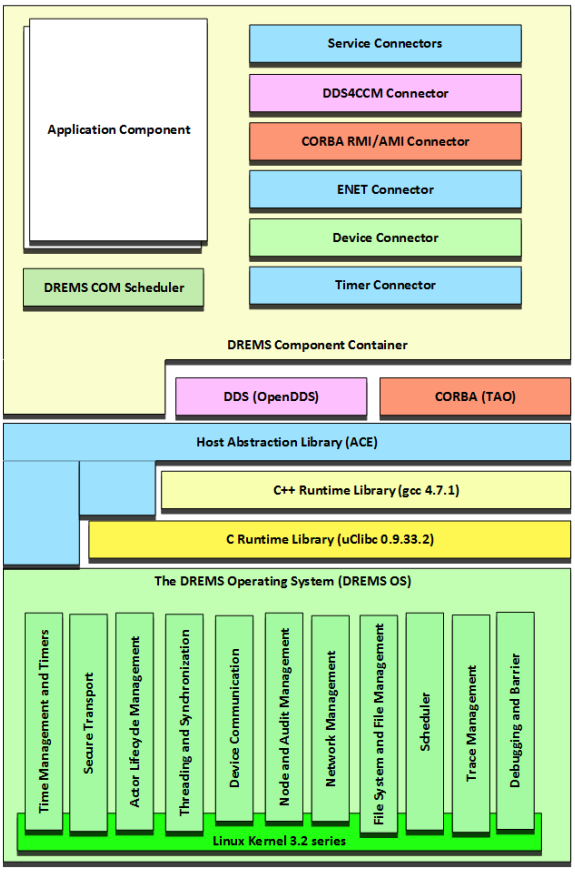

Figure 4 - DREMS run-time software layers

DREMS Run-time Software Platform Specification

The run-time software platform shall consist of several layers, as shown in the Figure. Practically all layers are based on existing and proven open-source technology. The bottom of the Figure shows how the operating system layer could be implemented by extending the Linux kernel with a number of specific services while at the same time keeping the existing Linux system calls. The advantage of this approach is that developers can use existing Linux system calls, side-by-side with the DREMS OS system calls.

The C and C++ run-time support libraries implement the conventional support services needed by the typical C and C++ programs. The C run-time library has entry points to access the DREMS OS system calls. These calls utilize data structures that shall be defined using the standard Interface Definition Language (IDL), which will allow them to be created, and manipulated using generated constructor and manipulation operators. The implementation of the DREMS operating system calls shall check the integrity of all data structures passed on the interface. This enables validation of the data structures on the interface, preventing potential abuse of the system calls.

Layered on top of the C and C++ run-time libraries are the Adaptive Communication Environment (ACE) libraries, which shall provide a low-overhead isolation layer for the higher level middleware elements that support CORBA and DDS. The CORBA implementation can be based on The ACE ORB (TAO, currently: version 6.1.4) that implements a subset of the CORBA standard for facilitating point-to-point interactions between distributed objects. Such interactions are in the form of Remote Method Invocations (RMIs) or Asynchronous Method Invocations (AMIs). RMIs shall follow the call-return semantics, where the caller waits until the server responds, while the AMIs shall follow the call-return-callback semantics, where the caller continues immediately and the response from the server is handled by a registered callback operation of the client. The CORBA subset that shall be implemented by the middleware has been selected to support a minimal set of core functions that are suitable for resource-constrained embedded systems.

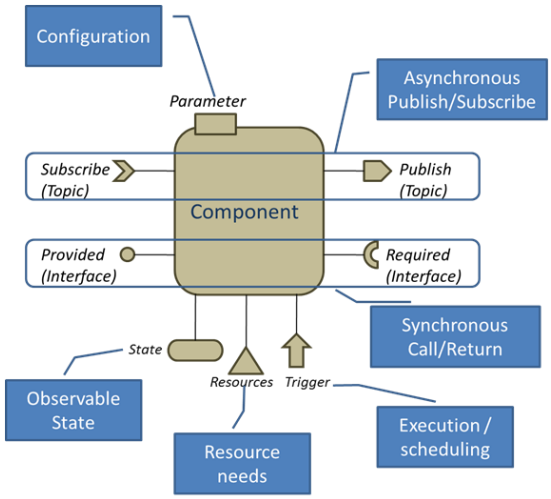

Figure 5 - A DREMS component

The DDS implementation can be based on the OpenDDS (currently: version 3.4) that implements a subset of the DDS standard for facilitating anonymous publish/subscribe interactions among distributed objects. In these interactions, publishers send typed messages of specific *topics* via the middleware which then distributes them to subscribers interested in those topics. Subscribers can be anywhere on the network, they can join and leave the system at any time – the distribution middleware decouples publishers from the subscribers. There shall be several quality-of-service attributes associated with publishers and subscribers that control features like buffering, reliability, delivery rate, etc. DDS is designed to be highly scalable, and its implementations meet the requirements of mission-critical applications.

CORBA and DDS shall provide for data exchange and basic interactions between distributed objects, but in DREMS objects are packaged into higher-level units called

components. A component, shown in the Figure, shall *publish* and *subscribe* to various topics (possibly many), implement (thus *provide*) interface(s), and expect (thus *require*) implementations of interfaces. Note that a component may contain several, tightly coupled objects. Components may expose (part of) their observable state via read-only state variables, accessible through specific methods. Components shall be configured via parameters and have memory resources needs. Component operations shall be scheduled based on events or the elapse of time. An event can be the arrival of a message to which the component has subscribed or an incoming request on a provided interface. Time-triggering is done by associating a timer with the component that invokes a selected operation on the component when a configurable amount of time elapses, possibly periodically repeating the operation. Component operations can perform computations, publish messages and call out to other components via the required interfaces. To avoid having to write complex locking code for components, component operations shall always be single threaded: inside of one component at most one thread shall be active at any time.

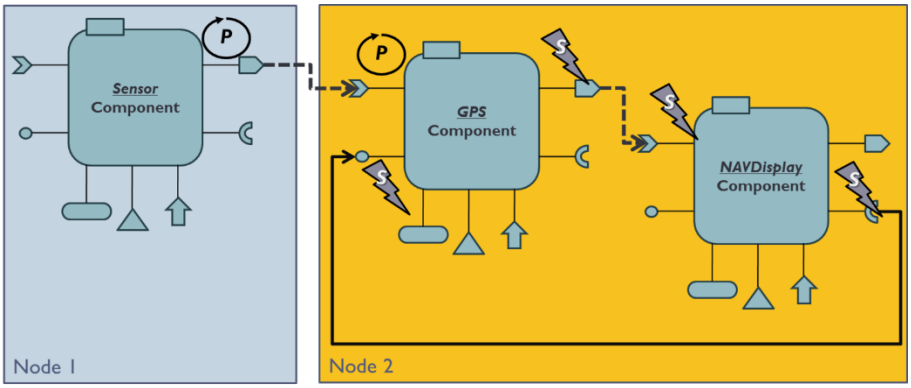

Figure 6 - Interacting DREMS components deployed on two different nodes

Actors shall be formed from interacting components, and applications shall be formed from actors that interact with each other via their interacting components. Actors (together with their components) can be deployed on different nodes of a network, but their composition and interactions shall always be clearly defined: they must happen either via remote method invocations or via publish/subscribe interactions. The Figure above shows an application where a `Sensor` component periodically (P) publishes a message to which a `GPS` component subscribes and which, in turn, sporadically (S) publishes another message that a `NAVDisplay` component consumes. This last component invokes the `GPS` component via a provided interface when it needs to refresh its own state. The messages published can be quite small, while the method invocation (that happens less frequently, and on demand) may transfer larger amounts of data. The number of possible combinations of interactions among components is quite large, but each interaction pattern is precisely defined, which allows application developers to understand all operational scenarios. Note that applications can be multi-threaded, but individual components shall be single threaded.

Interactions shall be realized by *connectors* that support specific interaction patterns. In addition to the two main patterns described above, components may interact using network sockets (for conventional message oriented networking using POSIX standard socket APIs), timers, and I/O devices. For each case, the synchronization between component code execution and the events of the external world is precisely defined and allows the implementation of various interactions to enable a high degree of asynchrony and responsiveness.

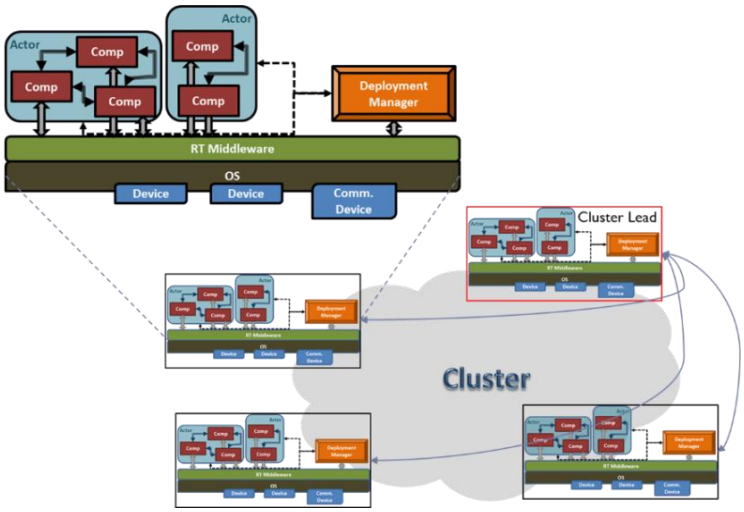

Figure 7 - The DREMS Deployment Manager and Applications

The run-time software platform shall include a key platform actor: the Deployment Manager (DM) that shall instantiate, configure, and dismantle applications. Every node on a network shall have a copy of the DM that acts as a controller for all applications on that node. The DMs shall communicate with each other, with one being the lead 'Cluster' DM. This cluster leader DM shall orchestrate the deployment of applications across the cluster with the help of the node DMs. For deployment, the binaries of application components and a deployment plan (an XML file) shall be placed on each node, then the cluster lead DM shall read and execute the plan: it shall start with the actors, install components, configure the network connections among the components, etc., and finally activate the components. This last step shall release the execution threads of the components. When the applications need to be removed, the DM shall stop the components, remove the network configuration, and stop the actors. A key feature of the deployment process is that the network connections among the parts (i.e., actors and components of the distributed application) shall be managed: the application business logic does not have to deal with this problem; everything is configured based on the deployment plan.

DREMS Design-time Development Platform Specification

Configuring the middleware and writing code that takes advantage of the component framework provided by a DREMS system can be a highly non-trivial and tedious task. To mitigate this problem and to enable programmer productivity, a model-driven development environment shall be available to simplify the tasks of the application developers and system integrators.

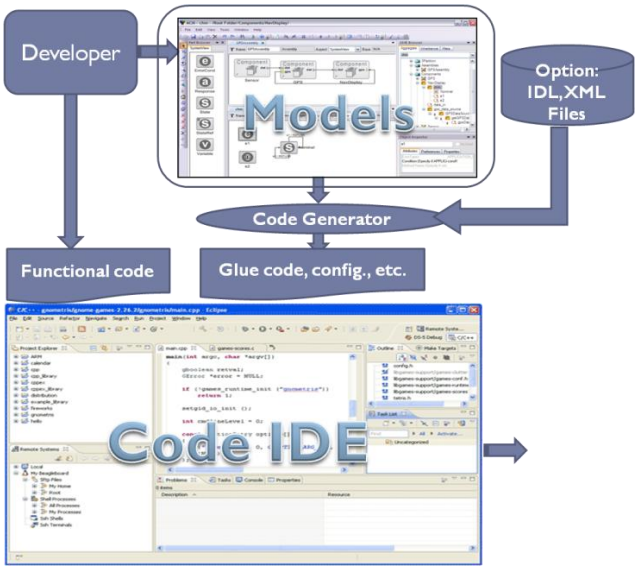

Figure 8 - Model-driven development with DREMS

In this environment, developers shall define with graphical and textual models various properties of the application, including: interface and message types, components types (in terms interfaces and publish/subscribe message types), component implementations, component assemblies, and applications (in terms interacting components and actors containing them). Additionally, the hardware platform for the cluster can be modeled: processors, network and device interfaces, network addresses, etc. Finally, the deployment of the application(s) on the hardware platform can be modeled (in terms mapping actors onto hardware nodes, and information flows onto network links). The framework shall permit both dynamic deployments that change over time as well as static deployments to be modeled. Models shall be processed by code generators that in turn produce several artifacts: source code, configuration files, scripts that facilitate the automated compilation and linking of the components, and other documents. The application developer shall provide the component implementation in the form of C++ code (currently; in the future: any other, supported executable language) and add it to the generated code. The compilation and debugging of the applications shall happen with the help of a conventional Interactive Development Environment (such as Eclipse) that supports editing, compiling and debugging the code. The result of this process shall be a set of component executables and a deployment plan – ready to be deployed on a cluster of nodes.

The model-driven approach has several benefits. (1) The model serves as the single source of all structural and configuration information for the system. (2) The tedious work of crafting middleware 'glue' code and configuration files for deployment is automated: everything is derived programmatically from the models. (3) The models provide an explicit representation of the architecture of all the applications running on the system – this enables architectural and performance analysis on the system before it is executed. (4) Models can be used for rapidly creating 'mockup' components and applications for rapid prototyping and evaluation.

DREMS Summary

The DREMS specification describes a sophisticated, end-to-end solution for building and running distributed real-time embedded applications. It specifies both a run-time framework that includes a state-of-the-art operating system with special features for

resource, application and network management together with a component framework with a precisely defined model of computation, and also a model-driven development toolchain that assists developers and integrators in managing the development process.

Suitability for a Vehicle IAP

This section evaluates the suitability of DREMS and AUTOSAR for a vehicle IAP using the IAP requirements listed at the beginning of the document.

Support a full spectrum of modern application architecture and design approaches

The DREMS specification describes both a *design-time* methodology that uses model-based development as well as a *run-time platform* to support modern architectures and design approaches. The design-time tools provide analysis (the models can verified for conformance to certain properties) and synthesis capabilities (many implementation-level artifacts are generated from the models). The run-time platform supports the needs of modern applications: real-time requirements, quality of service provisions (for network communication, CPU, memory), a security model and memory management.

The AUTOSAR specification places no mandates on a design-time methodology, which forces users to either develop their own tooling or choose a commercial vendor that they believe can meet their needs for application development. The AUTOSAR OS is designed to run on chips without memory protection, which imposes a severe security risk on applications that are increasingly developed by third parties. This puts a heavy burden on the system integrator to ensure that applications do not inadvertently corrupt one another and that no malicious applications that intentionally affect other applications are integrated into the system.

Enable efficient use of system resources

Both DREMS and AUTOSAR describe mechanisms for the efficient use of system resources. A DREMS run-time platform provides system-level resource managers that allocate and manage the use of system resources by applications. The DREMS design-time tools provide the ability to ensure that an integrated system is capable of providing the requested amount of resources to each application.

With the AUTOSAR approach, the Run Time Environment (RTE) is partly configured and partly generated based on the resource needs of the software components (applications) running on top of it. This approach allows the RTE to be optimized for a specific set of applications and ensures that the run-time system is neither under- nor over-provisioned. The allocation of resources to software components in AUTOSAR is very static, with many resources (such as mutexes and memory) being allocated once and never changing. This is in contrast to DREMS, which allows the dynamic creation and allocation of many system resources, including memory and mutexes.

Enable flexible use of system resources

While both DREMS and AUTOSAR enable *efficient* usage of system resources, DREMS permits a more *flexible* use of system resources. With AUTOSAR, the configuration of

a system is very static and is not expected or permitted to change over time: the communication and computational requirements of components are specified at design-time, and the RTE is generated and configured based on these requirements. An AUTOSAR system is provisioned for the optimality of a given set of software components (applications) with the expectation that the applications do not change.
DREMS is designed to allow dynamic software reconfigurations, both in the system software (the platform components) and user level applications. The DREMS operating system scheduler is designed to allow "extra" time to be utilized by applications. DREMS includes run-time support for QoS requirements on network communications which are expected to vary over time, and there is design-time tool support to analyze whether the communication requirements can be met based on the expected network availability.

Provide a robust, flexible, and manageable software infrastructure

As stated above, AUTOSAR relies on static configurations of system software. For instance, the AUTOSAR OS specification requires the OS to support the static configuration of the number of tasks and mutexes that will be created. Once deployed, AUTOSAR applications are not configured or updated dynamically.
DREMS, on the other hand, allows both the system software and user applications to be configured and updated dynamically at run-time. The DREMS operating system is expected to provide an administrative interface that can be used as a "command-line" interface to the system. This interface allows applications to be added, removed, stopped and started at run-time by a system administrator.

Enable controlled sharing of system resources

While both DREMS and AUTOSAR permit system resources, such as the processor and memory, to be shared between applications, only DREMS includes access control mechanisms (in the form of a multi-level security policy) to ensure that access to resources is controlled based upon a well-defined security policy. This allows a DREMS system to more easily include untrusted, third-party applications that run beside trusted applications.
Additionally, AUTOSAR is limited on the amount of access restrictions it can place on memory access due to the fact that it is expected to run on systems without memory protection. Because of this, the AUTOSAR system integrator is responsible for ensuring that applications do not access memory that has not been allocated to them and that memory errors in one application do not cause another application to crash.

Provide operational flexibility and maintainability

Both DREMS and AUTOSAR support debugging of applications, although to different degrees. AUTOSAR includes a Diagnostic Event Module which can store diagnostic system errors based on standardized Diagnostic Trouble Codes (DTCs). These trouble codes can then be used for post-mortem fault-diagnosis. AUTOSAR does not perform or provide facilities for online inspection of system and application activities other than through the use of the Diagnostic Event Module. AUTOSAR provides no provisions for replacing/updating either user or system level software at runtime.

DREMS includes support for limited online debugging, as well as both online and post-mortem fault analysis through a system-level fault manager. DREMS allows and expects that both user level and system level software will be configured/replaced/updated at runtime and provides facilities and interfaces for performing this activity.

Provide a comprehensive framework for fault management

This is an area of large differences between the two specifications. AUTOSAR does not provide any specification for a fault management framework. AUTOSAR relies on the underlying OS to provide timing protection to applications at run-time to ensure that application deadlines are satisfied. AUTOSAR uses the concept of *diagnostic monitors* to monitor specific physical systems or circuits. If an error is detected, the diagnostic monitor logs a Diagnostic Event with the Diagnostic Event Manager. Complex fault detection and isolation strategies could be built-in to either diagnostic monitors or to custom user applications, but these are beyond the scope of the AUTOSAR specification.

DREMS supports an extensive layered fault detection and isolation strategy that detects anomalies in different layers and diagnoses the root cause which should then be treated by user-provided fault mitigation logic. The DREMS fault model considers both physical and software faults that can occur during design-time, deployment-time and run-time, and prescribes measures and methodologies that can help prevent faults at all stages of development.

Provide a comprehensive framework for quality of service and resource management

AUTOSAR lacks a Quality-of-Service (QoS) model. Ensuring that applications meet their expected level of performance requires a-priori knowledge at design-time about their requirements and configuring/provisioning the system so that this level is always statically satisfied. AUTOSAR provides no way to describe levels of service that fluctuate over time (for instance, in response to the physical environment) and thus expects that all resource levels and requirements are constant throughout an application's lifetime. The burden is on the system integrator to know these requirements at design-time and provision the system accordingly.

DREMS includes both design-time and run-time QoS facilities. The design-time modeling tools can capture an application's network QoS requirements, and design-time analytical tools (based on Network Calculus) can then verify whether these requirements can be satisfied based on the expected network profile of the mission. DREMS assumes that both the network bandwidth and network communication requirements of applications can both vary over time, and includes design-time facilities to specify the requirements and run-time support to ensure the requirements are satisfied. DREMS is also capable of enforcing run-time guarantees on an application's usage of other system resources, including the CPU, dynamic memory and persistent storage.

Support real-time processing within a computing node

Both AUTOSAR and DREMS support real-time processing with single computing nodes. Each AUTOSAR ECU is expected to run an operating system that is amenable to reasoning about real-time performance. A derivative of the OSEK operating system is used as the basis for most AUTOSAR compliant operating systems to provide real-time scheduling. DREMS also provides support for real-time scheduling and uses a strict partition scheduler to ensure that, at run-time, tasks are run for the period and duration that they requested at design-time.

Where AUTOSAR and DREMS greatly different in this regard is the methodology each provides for ensuring that a system is capable of meeting its real-time requirements. AUTOSAR prescribes no specific tool or formalism for checking analytically that a set of real-time tasks are capable of meeting their timing requirements. The timing extensions to AUTOSAR are quite recent (introduced in version 4 of the AUTOSAR specification) and provide only statements about the timing requirements that an application should satisfy; they neither provide nor recommend any analytical methods for analyzing these requirements for satisfiability.

DREMS, on the other hand, allows timing requirements of individual processes to be specified at design-time in models. From this timing description, DREMS describes an analytical method that can not only check the satisfiability of the timing requirements of an integrated system, but that can generate a schedule for the operating system that satisfies the timing requirements.

Support real-time processing across the distributed computing platform

Both AUTOSAR and DREMS support real-time processing across the distributed computing platform. However, the assumptions that each place on the platform plays a key role in how this works. AUTOSAR assumes that the network bandwidth and latency of the underlying platform remain constant, and thus providing real-time processing across the platform partly consists of ensuring that the communication delays between communicating software components are satisfactory. Ensuring real-time processing across the whole system also involves finding a suitable mapping of software components to ECUs so that communication times are minimized but CPU requirements are still satisfied; this is a non-trivial constraint problem, and AUTOSAR describes no methodology to assist the user in this complex task.

DREMS does not assume that the network bandwidth and latency of the system remain constant over time. The design-time modeling tools of DREMS allow the expected network availability to be modeled and design-time analytical tools can then be used to check whether the requirements are satisfied. The infrastructure then enforces QoS policies at run-time to ensure that the real-time processing requirements are satisfied across the system.

Provide security mechanisms sufficient to satisfy system security requirements

DREMS and AUTOSAR have very different capabilities in regards to security. AUTOSAR has the capability to provide secure communications through the Crypto Security Manager (CSM), a software component that provides a standardized

interface that gives access to basic cryptographic functionalities. However, messages traveling across the bus are not currently encrypted, and there is nothing in the AUTOSAR specification to prohibit the CSM from being placed on a different ECU than a software component that uses it, which presents potential security vulnerability in simply getting the data to the CSM. The AUTOSAR standard does not describe any provisions to prevent one application from snooping on another, or even to prevent one application from maliciously manipulating the data of another application. In fact, because the underlying OS of an AUTOSAR compliant system is expected to run on a chip that lacks memory protection, the burden of ensuring that applications do not maliciously interfere with one another falls entirely on the system integrator. As more software is purchased from third-parties and then integrated by a central authority, which is one of the stated goals of AUTOSAR, this will become more difficult to ensure.

In addition to running on hardware that provides memory protection, DREMS also includes several other security features. Every process in DREMS runs in its own, private address space, including the operating system. Processes requiring persistent storage are also allocated separate file systems. The temporal scheduling of processes in DREMS prevents covert timing channels across different temporal partitions. To ensure secure communication, DREMS has a functionality called "Secure Transport" that provides cryptographic protection of messages sent between nodes, which ensures the confidentiality (only the destination can read the message), integrity (the message is not modified in transit) and authenticity (the message was sent by the source node from which it claims to originate) of each message.

Further, DREMS uses a multilevel security (MLS) policy to ensure that applications cannot arbitrarily communicate and exchange information. DREMS does this by attaching *labels* (such as "classified" or "top-secret") to messages and communication endpoints. When an application wants to send a message, it attaches a label to the message and specifies a destination communication endpoint. The run-time infrastructure (i.e., the operating system) then ensures that the message obeys the security policy with respect to the message label (i.e., it ensures that the destination endpoint has permission to read messages with that label). Internally, DREMS uses a Bell-LaPadula model to implement its MLS policy.

To summarize the security differences between the two platforms: DREMS has extensive security policies and mechanisms (MLS, IPSec, separate address spaces between applications, temporal isolation), while AUTOSAR provides very limited provisions (a cryptography module for encrypting messages sent between components).

Timing model

One major difference between DREMS and AUTOSAR is the real-time support each requires from the underlying operating system. The AUTOSAR OS specification states that it must be run on a real-time operating system that can be configured statically and that is amenable to reasoning of real-time performance. As described in the section on timing protection with AUTOSAR, the specification expects the underlying operating system to control three factors (the execution time of

tasks/ISRs, blocking time that tasks suffer from lower priority tasks locking shared resources, the interarrival rate of tasks/ISRs) at runtime to ensure timing protection. However, AUTOSAR describes no methodology for analyzing a whether a system design will meet its real-time requirements. The timing extensions to AUTOSAR are relatively recent additions, and thus manufacturers previously had to use proprietary timing specifications and internal tools to perform automated reasoning about the timing of their specifications. Even with the standardized timing extensions, manufacturers are on their own to devise methods and tooling for checking whether an integrated system is schedulable, which is a highly non-trivial task.

DREMS, on the other hand, uses a strict partition scheduler to ensure that at run-time, tasks are run for with the period and duration that they requested at design-time. This "temporal partitioning" concept is borrowed from the ARINC-653 standard. However, DREMS extends the ARINC-653 partitioning concept by allowing partitions to include multiple address spaces and permitting changes to the partition schedule without restarting the system. This temporal partitioning concept lends itself to a straightforward algorithm for (1) verifying at design-time whether an integrated system (consisting of processes from multiple applications) is schedulable and (2) generating such a schedule. The DREMS tool-suite includes built-in support for modeling the temporal partitioning requirements of individual processes, analyzing whether these temporal partitioning requirements can be satisfied and then synthesizing a valid partition schedule for the entire system.

Component model

Both DREMS and AUTOSAR use the software component as a fundamental abstraction. With AUTOSAR, both the basic software (BSW) and higher-level "functional" software (which makes use of the BSW) are built from components. The communication of components in both models is very similar, as described below.

An AUTOSAR component has well-defined ports through which it can interact with other components. AUTOSAR Interfaces define the services or data provided or required on/by a port. The interface can be a client-server interface (defines operations that can be invoked) or sender-receiver interface (allows data-oriented communication mechanisms). There are two types of ports: PPort and RPort. PPort provides an AUTOSAR interface, and an RPort requires one. A PPort either provides interface or sends data, and an RPort either invokes operations on an interface or reads data elements from a sender interface. Client-server communication can be either synchronous or asynchronous. Sender-receiver communication is asynchronous, and the sender neither expects nor gets a response from receivers. The communication infrastructure is responsible for distributing messages: the receiver does not know identity of sender.

Similarly, components in DREMS also expose well-defined ports for interacting with other components. Asynchronous and anonymous publish/subscribe is possible, as well as both synchronous and asynchronous client/server interactions.

AUTOSAR provides a way to describe the internal behavior of a SWC by breaking it down into Runnable Entities, which are executed at runtime. These Runnable Entities are described at design-time in an XML-configuration file. A timing

description can then refer to the activation, start and termination of the execution of Runnable Entities within a SWC, although this timing description is a specification of how the SWC and its Runnable Entities should behave, not necessarily how they behave, nor is a specific tool or methodology described for checking whether the timing description holds for a given SWC. DREMS does not provide a way to describe the internal behavior of a component other than its communication ports, timers and types the component uses.
DREMS does include a well-defined component scheduling model that states how and when component operations are scheduled. The AUTOSAR specification, on the other hand, does not state anything regarding the scheduling of component operations. Thus, the AUTOSAR component model is mainly an abstraction for communication, while the DREMS component model abstracts both communication and the scheduling of operations.

Fault model

The fault models used by AUTOSAR and DREMS are very different. AUTOSAR does not specify a fault model, nor does it describe a specific mechanism or methodology for dealing with faults that may occur in the system. Such fault handling is high-level functionality must be built on-top of the basic functionalities that are specified by AUTOSAR. The cited reference below describes one attempt at extending the AUTOSAR specification with fault tolerance provisions. That paper presents the standard fault-tolerance concept of *replication* to provide duplicate copies of software components that are used when the primary copy of a software component fails.
DREMS supports an extensive layered fault detection and isolation strategy that detects anomalies in different layers and diagnoses the root cause which should then be treated by user-provided fault mitigation logic. The DREMS fault model considers both physical and software faults that can occur during design-time, deployment-time and run-time, and prescribes measures and methodologies that can help prevent faults at all stages of development.
Because the fault management portion of DREMS is so extensive, the reader is referred to the paper, “A software platform for fractionated spacecraft,” cited below.
References:
“An AUTOSAR-Compliant Automotive Platform for Meeting Reliability and Timing Constraints”, J Kim, G Bhatia, R Rajkumar, M Jochim, SAE Technical Paper 01/2011; DOI:10.4271/2011-01-0448.
"A software platform for fractionated spacecraft", Dubey, A.; Emfinger, W.; Gokhale, A.; Karsai, G.; Otte, W.R.; Parsons, J.; Szabo, C.; Coglio, A.; Smith, E.; Bose, P., Vol 1, Number 20, IEEE Aerospace Conference, 2012.

Tool-support

Another big difference between the two specifications is the amount of included tool support for each. AUTOSAR is largely a group of specifications on different pieces of a system (the operating system, software components, communication, etc.) that does not mandate the use of any particular set of tools, nor is there a particular methodology recommended for the many stages of the workflow. Users must either

purchase or implement their own solutions for working with AUTOSAR. The result is that using AUTOSAR necessitates a huge commitment of both time and money. For this reason, many automobile companies, such as Toyota, are gradually integrating AUTOSAR into their workflow and do not expect to be fully AUTOSAR compliant for several years.

DREMS, on the other hand, provides reference implementations and tools for all stages of development. A comprehensive modeling tool provides the basis for design-time activities: defining component interfaces, configuring communication, specifying process scheduling requirements, defining hardware modules, and mapping software onto the hardware. Design-time analysis tools include a Colored Petri Net based-tool for modeling and analyzing the component-based software applications running on the platform. This allows the system to be verified for properties like deadline violations of component operations. Based on the results of this verification, the application model can be refined and restructured as required before code development begins. Another included design-time tool, based on Network Calculus, provides an analytical method of ensuring that applications receive their requested Quality of Service (QoS) requirements for network resources. A corresponding run-time tool ensures that the requirements are also satisfied at run-time when the network performance and availability varies over time.

The run-time software includes an operating system and middleware that provide the services and capabilities described in the specification. The reference implementation currently runs on the x86 architecture.

References:

J.-Y. Le Boudec and P. Thiran, “Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.” Berlin, Heidelberg: Springer-Verlag, 2001.

Computational requirements

The computational requirements of DREMS and AUTOSAR are substantial. Even though AUTOSAR does not specify which specific operating system must be used, it does specify requirements on the underlying operating system, and these requirements are based on the OSEK operating system specification. OSEK is primarily designed for low-end microcontrollers, and OSEK systems are expected to run on chips without memory protection. The AUTOSAR OS specification also states that it must run on low-end microcontrollers and without external resources.

DREMS has higher computational requirements than AUTOSAR. It expects both virtual memory, a memory management unit and memory protection to be present. The reference implementation versions of the DREMS OS and middleware run on the x86 architecture.

However, as automotive applications demand more and more computing power, the differences in the computational requirements needed by the two specifications will very likely start to converge. For instance, the introduction of TCP/IP over Ethernet into the upcoming BMW 7 Series will necessitate more memory and computing power than required in previous generations of the automobile.

Dynamic reconfiguration capabilities

The ability to dynamically reconfigure the system, such as adding a new application or moving an existing application to a different ECU, is a big advantage that DREMS has over AUTOSAR. In AUTOSAR, the system configuration is, by design, very static: the AUTOSAR OS specification states that the underlying operating system must be able to be configured statically, such as the number of processes and the number of resources (like mutexes). The AUTOSAR Run-Time Environment (RTE), which sits on top of the BSW but below the application-level Software Components (SWCs), is not only configured at compile-time for specific ECUs, but is also partly generated based on the requirements of the SWCs that will be running on it. A reconfiguration of the system, such as adding an application or moving an application from one ECU to another, cannot be done dynamically at run-time.

DREMS, on the other hand, is designed with such reconfiguration capabilities in mind. In fact, it is expected that applications may need to migrate, at run-time, between physical hardware nodes in response to both anticipated changes caused by the environment, such as network connectivity, as well as unexpected failures in both software and hardware. The deployment and configuration of an application onto a DREMS system is handled by a trusted piece of software called the *Deployment Manager* that is responsible for starting and stopping applications running on the system.

References:

"AUTOSAR – challenges and solutions from a software vendor's perspective", Th. M. Gall and R. Pallierer, in Elektrotechnik und Informationstechnik, Volume 128, Number 6, 2011.

Glossary

CAN: vehicle bus standard that allows microcontrollers and devices to communicate without a host computer. CAN uses a message-based protocol (rather than directly invoking a function, send a message to a process and rely on the process to select and invoke the actual code to run). Started in 1983, first CAN controller chips in 1987.

Car-to-X: technology enabling the exchange of information between vehicles and between vehicles and the traffic infrastructure. This is currently being integrated into vehicles by Mercedes-Benz.

E/E: electrical/electronic system. The elements of a vehicle's E/E architecture include data networks, diagnostics, fault tolerance, energy management, power and signal networks, and physical and functional partitioning.

Event Chain: describes the temporal correlation between two observable events (referred to as the stimulus and response) that have a functional dependency.

Fibex: Field Bus Exchange format. An XML-based standardized format used to describe complex, message-oriented communications systems. It is aimed at easing data/information exchange. Can be used to export on-board network databases, and for importing into different types of tools during development of vehicle networks. Fibex presently supports FlexRay, CAN, MOST and LIN.

FlexRay: automotive network communications protocol to govern on-board automotive computing. FlexRay is designed to be faster than CAN and TTP, but also more expensive. It supports data rates up to 10 Mbps and can have 2 independent data channels for fault-tolerance (degraded performance). Used by BMW, Audi, Mercedes.

Jitter: For a periodically occurring timing event, the jitter is defined as the maximum variation of its period with respect to a predefined standard period.

Latency: The latency of a timing event chain describes the time duration between the occurrence of the stimulus and the occurrence of the corresponding response.

LIN: local interconnect network is a broadcast serial network protocol used for communication between components in vehicles. Comprises one master and up to 16 slaves, both of which are usually microcontrollers. All messages are initiated by the master with at most one slave replying to a given message identifier. Developed partly because CAN bus is too expensive to implement for every component in car. The LIN Consortium was founded by 5 automakers and the first implemented version of the specification was done in 2002.

OBD: On-Board Diagnostics is a generic term referring to a vehicle's self-diagnostic and reporting capability. OBD systems give the owner access to the state of health information for vehicle subsystems.

OSEK: Open Systems and their Interfaces for the Electronics in Motor Vehicles is a standards body that has produced specifications for an embedded OS, communications stack and a network management protocol for automotive embedded systems. OSEK was designed to provide standard software architecture for the various ECUs throughout a car. It is an open standard published by a consortium. Some parts are standardized in ISO 17356. Specifies standard for

optional time-triggered real-time OSs. AUTOSAR reuses the OSEK specifications, with the OS being a backwards compatible superset of OSEK OS which covers the functionality of OSEKtime, and the communication module is derived from OSEK COM.

OSEK is architecture dependent; OSEK systems are expected to run on chips without memory protection. Features can be configured at compile-time. The number of tasks, stacks, and mutexes is statically configured. There are two types of tasks: basic (never block, but instead run to completion) and enhanced (can sleep and block on event objects). Only static priorities are used for tasks. FIFO is used to schedule equal priority tasks. The priority ceiling protocol ensures there are no deadlocks or priority inversions. Several implementations exist: Arctic Core, FreeOSEK, openOSEK, PICOS18, and others.

Even though the AUTOSAR OS specification does not explicitly state an operating system that must be used, but it is widely accepted that a modified version of OSEK is used as the underlying operating system.

Period: Describes the expected time interval between two consecutive event occurrences, neglecting variation (jitter).

Response: End point of an event chain.

Stimulus: Start point of an event chain.