KTH Industriell teknik
och management

Scoop Technical Report
Year 2011

Sagar Behere

Technical report
Department of Machine Design
Royal Institute of Technology
SE-100 44 Stockholm

TRITA - MMK 2012:12
ISSN 1400-1179
ISRN/KTH/MMK/R-12/12-SE

SCOOP Technical report

Sagar Behere

Contents

1 Terminology 2

2 Introduction 3
- 2.1 What the project is about 3
- 2.2 What this report is about 4

3 Design considerations 4
- 3.1 Technical characteristics of platooning applications 4
- 3.2 Engineering requirements for component based architectures 6

4 Our specific solution 10
- 4.1 Preliminary decisions 10
- 4.2 System architecture 12
  - 4.2.1 Top level functions 13
  - 4.2.2 System overview 14
  - 4.2.3 Implementation 15
  - 4.2.4 Concept of a software component 16
  - 4.2.5 Emergence of system behavior 19

5 Datalogging in component based architectures 20
- 5.1 What data should be logged 21
- 5.2 Principles of datalogging 21
- 5.3 Datalogger implementation 22

6 Testing the system 24

7 Conclusions and reflections 26

8 Appendix A: Dataflows 27

9 Appendix B: Additional questions 30

References 31

1 Terminology

The following terms occur throughout the document

Architecture The definition of an architecture used in ANSI/IEEE Std 1471-2000 is: "The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."
This document, however, uses the TOGAF[1] sense of the term, which embraces but does not strictly adhere to ANSI/IEEE Std 1471-2000 terminology. In TOGAF, "architecture" has two meanings depending upon its contextual usage:
1. A formal description of a system, or a detailed plan of the system at component level to guide its implementation
2. The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time.

Software component A component is a basic unit of functionality which executes one or more (real-time) programs in a single thread.[2]

Software framework is an abstraction in which software providing generic functionality can be selectively changed by user code, thus providing application specific software. It is a collection of software libraries providing a defined application programming interface and is used to build the standard structure of an application.[3, 4]

Middleware is computer software that connects software components or applications. It is used most often to support complex, distributed applications.

Realtime toolkit is a part of the Orocos[5] software framework. It provides a C++ framework, or "runtime", targeting the implementation of (realtime and non-realtime) control systems.

Ego vehicle is a fancy way of saying "our" vehicle

Datalogging storing of data flowing within the system for purposes of future retrieval, visualisation or processing

HMI stands for Human Machine Interface, and refers to the technologies that permit interaction between a human being and a machine. This generally is in the form of graphical displays and control panels.

2 Introduction

2.1 What the project is about

The Scoop project was initiated in order to create a participating entry in the Grand Cooperative Driving Challenge(GCDC)[6]. The GCDC is a situation where multiple vehicles drive cooperatively in a platooning scenario. The terms 'cooperative driving' and 'platooning scenario' will now be explained, before proceeding further.

Cooperative Driving is a driving condition where vehicles can communicate with other vehicles (V2V) and also with the surrounding road infrastructure (V2I). This communication can take place over a variety of wireless media and can be used for vehicle motion control, among other things.

Platooning scenario is one possible scenario in cooperative driving. In this scenario, vehicles drive one after the other, where the first vehicle is denoted as the 'lead vehicle'. The lead vehicle is generally driven manually. The rest of the vehicles follow the lead vehicle autonomously. The lead vehicle and the following vehicles together form a 'platoon'. It is useful in a rough way to think of a platoon of vehicles as a 'road train'. A platoon on a road can split into multiple platoons, merge with other platoons etc. The platoon must obey the local road laws, respect speed limits, react appropriately to traffic lights and so on.

Cooperative driving and platooning scenarios have a variety of benefits for road traffic and the environment. The benefits include reduced traffic congestion, lower emissions and fuel consumption as well as safer driving.

The GCDC is organized by TNO of the Netherlands[7]. In the GCDC, teams from all over Europe participate with their vehicles. TNO provides the lead vehicle and the vehicles of the participating teams will be the following vehicles in a platooning scenario. The lead vehicle may try to induce oscillations in the platoon by means of erratic driving. The resulting effect on the platoon will be a function of behaviors of each following team vehicle. TNO then uses a number of criteria to determine which vehicle performs the best with respect to automated driving in a platoon. Detailed rules of the competition can be found in the GCDC rules and technology document[16].

The GCDC is challenging because the platoon as a whole must function, despite the fact that no vehicle has precise control over the motion of other vehicles in the platoon. The vehicles exchange data continuously using standardized specifications, but the control strategies and implementations of each participating team/vehicle are different.

The first GCDC was held in May 2011 at Helmond, the Netherlands. One of the participating teams was Scoop. Scoop was comprised of researchers and engineers from KTH and Scania CV AB. The team vehicle was a Scania R730 series truck. This is a state of the art commercial product from Scania, which was further modified with functions to enable autonomous driving in a platoon.

2.2 What this report is about

This report deals with the technical solution that was implemented for the GCDC 2011. Some reflections on the design process are also included. The goal of the report is to make the user understand the technical solution and the motivations behind the design choices made. Individual technologies are not descibed in depth, but adequate references are provided where necessary.

Section 3 presents some technical characteristics of platooning applications, followed by engineering design considerations for the component based architectures.

Section 4 describes the specific solution we implemented. It begins by listing and motivating some preliminary decisions followed by a description of the implementation. It concludes by showing how the solution leads to a realization of the desired functionality.

Section 5 elaborates on the specific topic of datalogging in component based architectures. It describes what kind of data should be logged and some principles of datalogging relevant to component based architectures.

Section 6 describes how the system was tested. Section 7 mentions some reflections on the design process as well as the solution and concludes the report.

3 Design considerations

This chapter presents some design considerations that affect the specific solutions presented in later chapters. The intention is to set a context for describing the solutions that follow.

3.1 Technical characteristics of platooning applications

Platooning applications inherit characteristics from the automotive domain. Some characteristics of the automotive domain are discussed in [17, 20, 18, 19]. Additional characteristics especially important for platooning applications are mentioned below

1. There is little, no or mostly unpredictable control over the behavior of other vehicles in the platoon. The only certainty is the ability to control the ego vehicle, and broadcast information about it. Despite this, it is the responsibility of all platooning systems to maintain the integrity of the platoon.

2. The development of platooning systems is practically always a mixture of visual programming methods (Simulink etc.) and hand coding. The architecture should enable smooth integration of the artefacts produced by both methods.

3. A platooning system is made up of a mixture of tasks. Some tasks have hard real-time constraints, some are soft realtime, while others are not generally time critical.

4. A platooning application is developed as an optional feature to an existing vehicle platform. Therefore, the onus on the architect is two-fold
   (a) The platooning system should conform to the existing architecture, its possibilities and limitations
   (b) The platooning system should not dictate changes or otherwise impact the existing system to the maximum extent possible
5. The design of a platooning system is significantly affected by the sensors used to obtain a "world-view". For example, a system using a camera would differ from a system using a radar, which in turn would be different from a system using a bank of lasers. It is therefore important to have a flexible data fusion component, yet keep the rest of the system invariant to its changes.
6. A platooning application mixes safety critical and non-safety related functions. A strict awareness needs to be maintained of this fact when developing the architecture, because the impact goes beyond obvious technical matters and affects processes like certifications.
7. Platooning applications integrate domain expertise. Therefore, the architect needs to have sufficient knowledge of all domain functions; mere knowledge of functional and/or behavioral requirements is not sufficient to create the architecture.
8. Platooning applications involve an inherent safety paradox. Ideally, a system should not depend on external input to assure its own safety. However, a platooning application must do so. It is a design challenge to make the system as safe as possible, without making assumptions about the quality and trustworthiness of incoming data.
9. A platooning application is closely related in concept to an autonomous system. Therefore, the presence of a system ego is needed in some form or the other. In other words, there must be a system component that knows what functions the system-as-a-whole is supposed to perform, and how those functions are done by the system.
10. A platooning application is characterized by distributed, hierarchical control. The platooning controller takes decisions and creates setpoints for high-level motion variables, like speed, acceleration etc. However, the actual regulation of these variables occurs in controllers that are completely distinct from the platooning controller
11. A key differentiator for a platooning system is the need to percieve the environment and form a world-view from fairly limited sensor input[1]. Therefore, it is

[1] as compared to the perception abilities of a human driver.

necessary to have a lot of trust in sensor data (e.g.: There really is no car in front of us.) However, the reality is that the sensors in use provide reliable data only under a subset of circumstances that occur while driving in a platoon. Therefore, redundancy, multiple sensors, fusion and above all, reasoning on received data is an unavoidable part of a platooning system

12. In addition to the accuracy of input data, it is also necessary at all times to maintain the accuracy of the data being broadcast from the platooning system, because this is used by other vehicles in the platoon

13. The development, testing and verification activities for a platooning system are entirely governed by the datalogging and visualization infrastructure. Although not a part of the final product, good datalogging tools and frameworks are crucial during the development phase. As such, the architecture must have native support for datalogging.

14. Data, control and computation need to be handled separately within the architecture. A platooning system is a mix of time and event triggered control. The specification and design of inter-component interfaces is driven primarily by data, while the scheduling of the components and their priorities are driven primarily by control and computation requirements

15. A platooning application involves heavy parameterization and the architecture should support live calibration methods so that parameter values can be correctly decided during live tests of the system

16. A platooning system needs a human machine interface (HMI) that can be used to make the human driver aware of what the system is doing. Since the vehicle already has an HMI, which is part of its existing architecture, decisions need to be made on the HMI specification of the platooning application and whether it can fit into the existing HMI scheme

17. The platooning system should ideally be able to determine which of its constituent components are working properly and use that as a basis for system behavior. Thus, it should be possible to gracefully degrade functionality based on the health of monitored components

3.2 Engineering requirements for component based architectures

When talking about embedded systems and software development, engineers often use phrases like, "This system is a joy to work with." or "Working with this code is like kicking a dead whale down the beach." Such statements indicate that there are certain systems and implementations that are 'better' for engineering work. Words like 'elegant

design', 'feels good' creep in, without there being an objective set of metrics for what they mean.

This section stems from the experiences of the authors in designing embedded systems. The authors feel that if the requirements listed below are fulfilled, the system will be more 'pleasurable' to work with, than if these requirements are not fulfilled. These requirements may or may not have a direct relation to customer requirements or other requirements of the product, but they enable smoother and more satisfying work during system development, testing and debugging. Some reasoning is given for stating each requirement. This reasoning is not comprehensive. There may be many more reasons for a requirement, than stated here.

1. The system should be implementable with existing tools.
   Most experienced developers already have a 'bag of favorite tools and tricks' which they try to apply to the new systems they are developing. It is often far easier to design an architecture keeping in mind the capabilities and limitations of the tools that may potentially be used for the project. Contrast this approach where the architecture is designed in 'free space', whereupon it is often realized during the implementation that no single (or multiple but compatible) tools can get the job done. This results in quickbrewed solutions in place of tested and well-maintained ones for problems that could have been solved using the latter. Good developers like to reuse trusted, 'has worked before' solutions and having to use something else just because the architect 'didn't think of it' is a cause of frustration.

2. There should be flexibility throughout the Design and Development process.
   Once the rough skeleton of the architecture has been decided, it is time to start with the implementation without waiting (or even attempting) to refining it down to the finest detail. During the D&D of most new projects, all information about all project aspects is not (and can not) be known in advance. Therefore, attempting to refine the architecture prior to learning the lessons that are learned during development is a recipe for rework. Flexibility implies the ability to refine the architecture as it is developed, using the lessons learned during development.

3. Evolvability must be a deliberate design decision.
   Often, the customer does know tacitly what is needed, but the specification may not be comprehensive enough. If the system is closely engineered to the specification, then addition of new features that the customer has 'just realized that he wants' can be a problem. If the architecture is not designed for evolvability, this either results in 'quick hacks' to 'get this specific task done' or a costly and time consuming redesign. The latter will then be susceptible to the same problem and the former method of using quick and dirty hacks often gets repeated till the system becomes an unmanageable mess that has to be redesigned. Therefore, the architect can save trouble all around, if evolvability has been built into the system, for some defined boundaries of 'evolvability'. This is especially true for

products characterized by the use of common components and platforms, which are reused in different product variants.

4. Dynamic architecture changes.
   This refers to the possibility of changing the inter-component connections to give rise to new system behaviors. Such possibilites are enabled by making sure that the components in the architecture are self-contained. There should be minimum dependence on other components and it should never be necessary for a component to know the inner working of another component. In the extreme case, a component need not know where its input data comes from and who consumes its output data. When components are decoupled to a great extent, the system developers are free to throw a bunch of components together and decide how they should be strung together to generate different system behaviors. The use of supervisory components then enables dynamic changes to component assemblies leading to adaptable, robust systems.

5. It should be easy to swap algorithms.
   The most optimal algorithm for a given task may be a topic of research while the architecture is being designed. Alternatively as the system is deployed, newer algorithms may come to the fore. Sometimes, it is necessary to evaluate differing algorithms keeping the rest of the system constant. It is therefore a good idea to separate the algorithm from the architecture. Providing well defined 'slots' to run algorithms provides a way to execute/test various algorithms. This is consistent with the principles of flexibility and evolvability.

6. Ability to be implemented in existing frameworks like AUTOSAR
   This isn't really a strict requirement, but is a 'nice-to-have' property of component based architectures for automotive solutions. AUTOSAR is rapidly becoming the upcoming standard for implementing automotive software and it is a good thing if the architecture can be implemented in AUTOSAR without extensive reworking.

7. Ability of real-time interaction with the architecture
   Real-time interaction is the ability to start, stop, examine and modify components while the system is executing. This not only saves time during system tuning (by avoiding the stop system, change values, recompile, re-execute cycle), but also provides an insight into the system operation as it executes. In a sense, such a real-time interaction is 'debugging on steroids' and can be used to quickly isolate problems and causes of erratic system behavior.

8. Ability to include autogenerated components
   There are many parts of a modern embedded system that may not be hand-coded. For example, control components are often designed as block diagrams in Simulink. However, if these parts can be included in the framework with further manual intervention, the system development efforts are reduced.

9. Implementation documentation autogeneration
   The actual source code of an executing system is its ultimate reference and documentation. Architectural description documents are very useful to understand system design concepts, however, these documents often get out-of-sync with the current implementation for a variety of reasons. Additionally, certain finer implementation level detail may not be documented in any architecture description. Therefore, it is advantageous to be able to generate some form of documentation from the actual source code of the implementation. The tools and frameworks used for the implementation should support the creation of such automated documentation.

10. Diagnostic/Monitoring services
    The architecture should permit the examination of the health and status of each component. In case of reported or suspected problems with a component, there should be means to run diagnostics to assess the extent of the problem and reason about its consequences. Having this facility enables the design of robust architectures, with the possibility of graceful degradation in performance, when errors occur.

11. Ability to satisfy timing constraints
    In case any of the components need to run time critical tasks, the design should not be limited because the component framework does not support realtime operation. Therefore, it should be possible for the architectural framework to mix together realtime and non-realtime components, satisfying the timing constraints wherever applicable.

12. Platform services-scheduling, memory management etc.
    Either the component framework or the underlying operating system should take care of basic services like scheduling, memory management etc. This leaves the architect free to focus purely on the component based design within the limits of the framework.

13. Ability to reuse existing code
    Some frameworks are so specialized that they no longer support the possibility of using third party or legacy code. When implementing a new system, it beneficial to reuse as much existing code as possible. Doing so not only conserves time and effort, but also provides some defence against bugs, since the reviews and maintainence of third party code, like software libraries, is often superior to what can be managed by the project team.

14. Minimum impact on existing infrastructure
    The introduction of a new framework should be as minimally intrusive into the existing infrastructure as possible. This makes it possible to gradually evolve the architecture of the entire system. Designers of established products often frown

upon large changes and having a minimally invasive method to add functionality makes for a safer upgrade path for existing products.

4 Our specific solution

This section describes the specific solution we implemented. The description begins with some preliminary decisions based on constraints, developer preferences and long term interests.

The reader will notice how the theoretically large set of choices is actually quite narrow in practice. This is because one choice often constrains another. For example, when you choose to use a certain readily available framework, you must often use the programming language the framework was written in.

4.1 Preliminary decisions

1. The software platform of choice should be capable of supporting real-time operation, should that prove necessary. It should be possible to use existing tools and software libraries to prevent re-inventing the wheel unless absolutely necessary. Finally, the entire software ecosystem should preferable be open-source and freely modifiable, because this leads to a great many benefits while preventing vendor lock-in[8]. Based on these considerations, GNU/Linux is the only mature candidate for consideration. It has the widest choice of pre-existing software needed to support the task at hand. The 802.11p protocol driver[15] is currently only available for the Linux kernel. GNU/Linux was also the preferred choice of the lead developers, for reasons of familiarity and ideology.

2. The architecture should be component based, with good separation of concerns between the components. (The concept of a software component is extensively described in subsection 4.2.4.) The system behavior would result from interaction among the components. To support this design, there was a need for a mature component based framework that enables easy creation of components and provides a way for the components to interact with each other. Furthemore, the component based framework should be able to take advantage of the underlying operating system's realtime capabilities and provide realtime execution of the components and inter-component communication. The framework should be cross-platform capable, to facilitate easy migration of the solution to other operating systems in future, should it prove necessary. Based on these considerations, the OROCOS framework[5, 21] was selected, since it is the only open-source framework that fulfils the requirements. Orocos has been successfully used in similar projects in the past and has an active user community and support base.

3. Most realtime code for Linux is written either in C or C++. The orocos framework is coded in C++ and orocos based applications must be written in C++.

Therefore, C++ naturally falls out as the programming language of choice.

4. The Boost programming libraries[9] are cross-platform, peer reviewed and highly regarded among C++ programmers. Boost provides many (if not most) commonly used facilities required by a complex C++ program and does so with exceptionally high quality. Therefore, Boost must be used wherever it can. Given the extraordinary support provided by Boost, 'C++ with Boost' turns into a self-reinforcing and justifying combination.

5. Data exchanged among components should be in the form of opaque structures with accessors. An opaque structure is a structure whose internal reprsentation is not known to the user, and the data in the structure is extracted by means of dedicated functions, called accessors. This enables changing the way data is stored without changing all the code that accesses the data. It also enables actions like serializing the transferred data (more on this in section 5) without being aware of the actual data content and organization. Google protobufs[10] or boost serialization library[11] provide convenient classes for creating seriazable objects. Furthermore, the classes are extensible from a base class, enabling creation of multiple class types that share common access methods.

6. It should be possible to log all data exchanged between components. Additional (perhaps temporary) data generated by a component during operation, including status and error messages should also be logged. The data logging mechanism should not significantly affect the system performance, and the system should be capable of running without the logging mechanism. Furthermore, it should be possible to transmit all data to be logged to another system, where it can be saved, replayed and analyzed. It should be easy to add/remove data types and structures to the logs. Based on these requirements, dedicated datalogging channels are created from each component to a datalogging component. The datalogging component then uses the google protobuf object methods to serialize the data over a TCP/IP socket. A number of small 'consumer' programs would be written for accessing the log data and storing/visualizing/processing it as necessary.

7. The developed architecture should be easily transferrable to an AUTOSAR implementation. This is a requirement because once the system goes beyond the prototyping phase, it tends to be implemented in AUTOSAR, which is the upcoming implementation standard in the automotive industry. It is very inefficient if the prototyped system needs to undergo an architectural change before it can be implemented 'for real'.

8. The Human Machine Interface (HMI) to the system should be completely independent in design and execution from the core system. It should not affect the execution of the core system in any way. As long as the core system provides the information sufficient for an HMI, the actual HMI implementation can be done independently on a distinct hardware/software setup. Based on this requirement,

an HMI program would be created that scans the log data from the datalogger component's TCP/IP stream and displays it to the driver. The program would be run on a different computer.

9. The system should be designed and coded in a 'realtime friendly' manner. Realtime friendly means not having to change the design or the implemented code, should the system be moved to a real-time platform. This means that the developed solution can initially be tested on a non-realtime kernel and then on a realtime kernel if necessary. Thus, merely booting with a different kernel and/or recompiling should be all it takes for transforming between realtime and non-realtime operation. Examples of realtime friendly coding are: no memory allocation in dynamic code, usage of non-blocking functions only and so on.

10. The developed software system should use the services of existing system daemons as much as possible, unless overriding technical reasons against their usage are present. There is a gps daemon[12] that can already connect to different types of gps units and serve their data. Therefore, the gps daemon must be used instead of writing code to communicate to the device directly. The Network Time Protocol (NTP) daemon[13] exists to synchronize the system clock with an external source, therefore it must be used to synchronize the system clock with the gps, instead of writing code to do that. The same argument goes for usage of the calm daemon for wireless communication.

11. The control part of the system must be executed in an ECU. This is because the control system will be designed in Simulink and established methods exist to translate the Simulink design into a real-world executing system. Re-implementing the Simulink design manually in C++ simply because the rest of the system is in C++ doesn't make sense when ECUs and associated tool-chains are readily available. This decision splits the system hardware into two: A part that executes Linux, and another that executes the Simulink control models.

12. Since CAN is the established communication protocol for an ECU, the communication between the Linux computer and ECU will be over CAN.

13. Finally, the cooperative driving solution should make minimum modifications to the existing truck system. Preferably, the solution should be 'plug-and-play', taking over certain system functions when it is plugged in.

These preliminary decisions impose several tight constraints on the system design. This is good because they narrow down the possible design space, enabling us to move on to the next iteration of the design.

4.2 System architecture

In this description of the system architecture, we first look at the top level functions that need to be realized. This is followed by a system overview for realizing these

functions. Finally, we describe the specific implementation of the architecture.

4.2.1 Top level functions

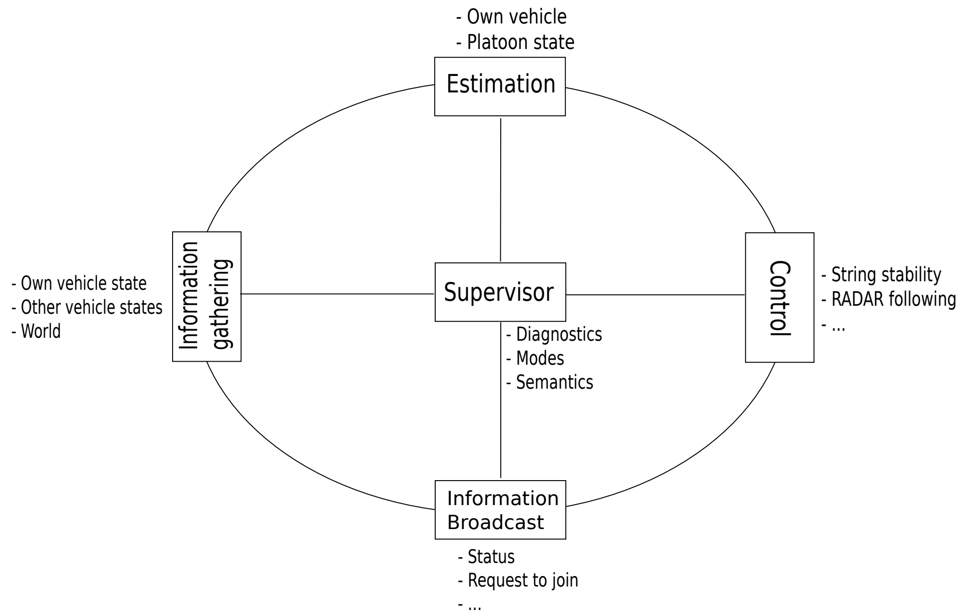

Figure 1: Function Structuring

The five main functions needed for a platooning application[2], as shown in figure 1 are

1. Information gathering
2. Estimation
3. Information broadcast
4. Control
5. Supervision

The information gathering function has the responsibility of gathering data about the ego vehicle and the environment. The environment consists at least of other vehicles in the vicinity and road objects like speed signs, traffic lights, lane markings etc. The information can be gathered by different or multiple means. For example, information about the ego vehicle can be gathered by reading the vehicle's CAN bus and via a GPS device. Information about other vehicles can be gathered from wireless broadcasts or

[2]Just to be clear, these are the additional functions needed for platooning, and assume the existence of a vehicle platform providing basic sensing, control, HMI etc.

local sensors like radar. Information about road objects can be gathered from wireless broadcasts or local sensors like cameras. The information gathering component is thus a conceptual function that provides all the raw data necessary for operation, regardless of the nature and source of the data.

The raw data provided by the information gathering function has to be filtered through the estimation function. The estimation function is needed for multiple reasons. The raw data provided by a sensor cannot be trusted blindly and used as a control input. The data must be 'sanitized' before use. Sanitaization means that some commonsense checks have to be made to ensure that values received are not improbable. This includes elimination of outliers from the signals. For example if the position of the vehicle ahead suddenly shifts by a few kilometers in a matter of milliseconds, then there is an obvious flaw in the readings. Some pieces of information can come from multiple sources. An example is the vehicle speed signal, which can come from the vehicle's internal CAN bus and also from a GPS receiver. The values from multiple sources must be fused together to provide a single value stream with a high degree of confidence. Sometimes, the signal from a data source may be briefly lost. For example, a GPS device may not provide position information when the vehicle is in a tunnel or below a bridge, and there is no line of sight to overhead satellites. Under such situations, the estimation function must provide extrapolated information as long as possible. The estimation function can also calculate values of variables which are not directly observeable. The output of the estimation function would be continuously updated state vectors representing the state of the ego vehicle, the vehicles in the environment and the road objects.

A vehicle participating in a cooperative driving scenario must broadcast certain information about itself. This information is collected from various parts within the system and broadcast (generally over wireless media) at different frequencies[3]. The broadcasts may also contain control requests, for example, requests to join existing platoons. The information broadcast function is the single place from which information goes out of the vehicle.

The control function is directly responsible for vehicle motion. Gathered and processed data is fed to this function which decides and regulates vehicle speed, acceleration and other motion parameters.

The supervisor function is responsible for the overall working of the system. It performs mode management, diagnostic monitoring and coordinates the information flow within the system. Conceptually, it is above all other functions in the hierarchy. The supervisor is also responsible for graceful system degradation in case of problems.

4.2.2 System overview

This section presents a very high-level logical view of the architecture. Refer figure 2.

The ECU represents the control function. The reason to show it separately is to emphasize the fact that in function, implementation and operation, it is separate from

[3]Frequencies here means at different intervals, not the radio frequency of transmission.

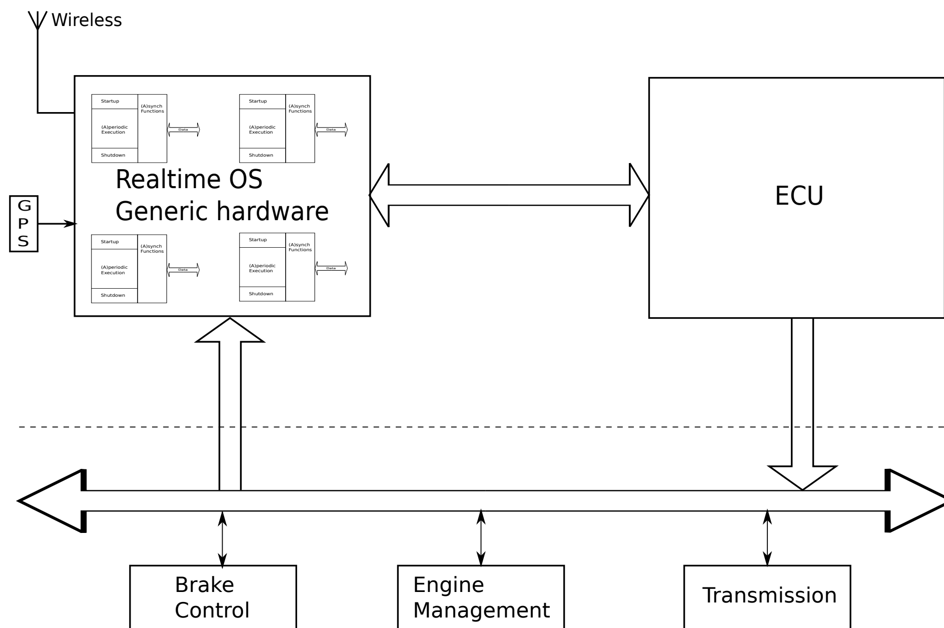

Figure 2: System overview

the rest of the system. It reads a certain set of data as input and makes certain actuation requests as output.

The Brake Control, Engine Management and Cruise Control are the 'actuators' influenced by the ECU. These are standard vehicle functions provided by the manufacturer for the vehicle's motion control. The ECU uses them to realize desired vehicle trajectories.

Finally, the rest of the functions are grouped together for execution on a generic computer. This is because their implementation is foreseen as individual components in a component based software framework.

The generic computer and ECU are connected to the vehicle's CAN bus, on which the 'actuators' are also present. There is a dedicated CAN bus connecting the generic computer and the ECU to handle the high bandwith communication between them.

The GPS device as well as wireless routers are connected to the generic computer.

4.2.3 Implementation

Figure 3 shows an implementation view of the architecture. The topmost layer shows the top level functions that the architecture should realize. These functions are described in section 4.2.1.

The next layer is the software layer, which shows the developed software components and their contribution to the top level functions. A top level function may be realized using more than one software component. For example, the Information Gathering function is realized using a combination of the GPS Manager, CAN Manager and Wireless components. A software component may contribute to the realization of more than one

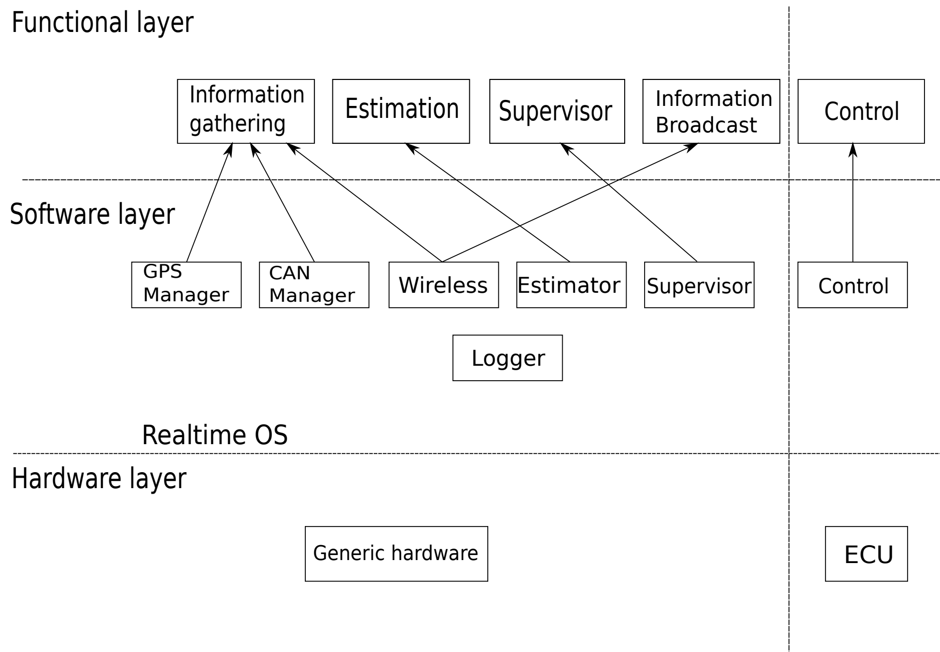

Figure 3: Implementation

top level function. For example, the wireless component contributes to the realization of the Information Gathering as well as the Information Broadcast functions. There also exist software components that do not directly contribute to the realization of top level functions. An example is the Logger component. The logger is useful for debugging the system and exists purely at the software level, in this architecture. The software components are executed by a realtime operating system. The operating system in this case is GNU/Linux, running the Xenomai realtime framework[14] for the Linux kernel. A detailed description of the software component concept is given in section 4.2.4. The software implementation of the Control function is similar in principle to other components in the Software layer. However, it differs in implementation. The control component is designed and implemented in Simulink and the software code it executes is autogenerated from within Simulink. Thus, there is no traditional hand-coding of the control function.

The lowermost layer is the hardware layer, which executes the contents of the software layer. This hardware layer is partitioned into two. The ECU is a physically distinct piece of hardware which executes the control software. The other piece of hardware is a generic computer running the GNU/Linux operating system.

4.2.4 Concept of a software component

This subsection explains the concept of a software component, as it applies to this architecture. It begins by describing the software environment which the component is a part of, and then elaborates on the structure of the component itself.

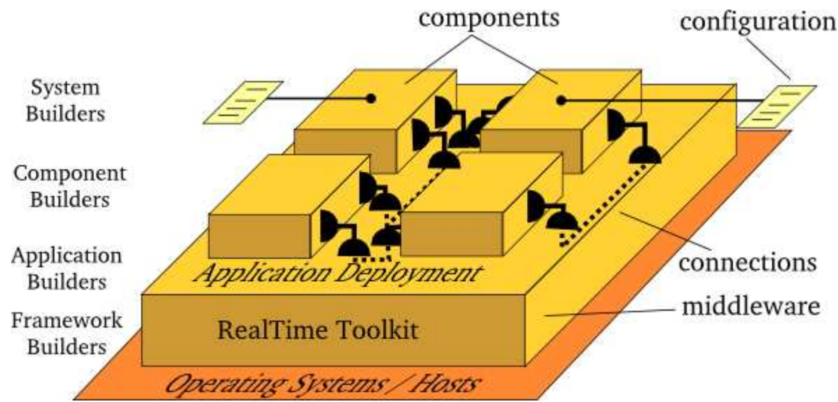

Figure 4: Environment of a software component(source: Orocos component builders manual[2])

Refer to figure 4. At the bottom layer, you have the operating system. On top of the operating system exists an operating system abstraction layer. This abstraction layer presents a generic interface to the services of the operating system to the layers above it. In essence, it makes the implementation of the upper layers independent of the actual operating system being used. In order to use a different operating system, only the operating system abstraction layer needs to be changed; not the layers above it. Such abstraction layers are a common and desirable feature of software architectures. The abstraction layer in this architecture is termed the RealTime Toolkit. This nomenclature originates from Orocos[5], the software framework chosen for the architecture's implementation.

On top of the realtime toolkit are placed the software components. A software component may be thought of as a 'unit of functionality'. The component executes periodically or aperiodically and does some specified tasks every time it executes. For example, a component for acquiring GPS data may periodically execute code that reads data from a GPS device, and send this data to another component that needs it. A component can execute complex state machines and have mutable properties[4], attributes and other meta information. Components can also communicate with each other using a variety of mechanisms.

A component based software framework allows you to create components and specify how to execute them. It also allows you to specify the connections between the components you have created and how they communicate with each other. Sophisticated component based frameworks also provide the operating system abstraction layer suitable for running their components. The framework chosen for implementing our architecture, Orocos, provides such an abstraction layer. In Orocos, one component usually maps to a single (RT)OS thread. However, it is also possible to create so called Orocos 'Activities' which can be implemented in their own threads. Detailed description of the mappings between components and operating system primitives are given in the Orocos component builder's manual[2].

[4]A mutable property is a property that can be changed during runtime.

The desired behavior of a component based architecture emerges from the concurrent execution of multiple components and their interactions.

Now let us look at the structure of a software component (figure 5) used in our architecture. The structure is dictated by the OROCOS, the framework chosen for implementing the components.

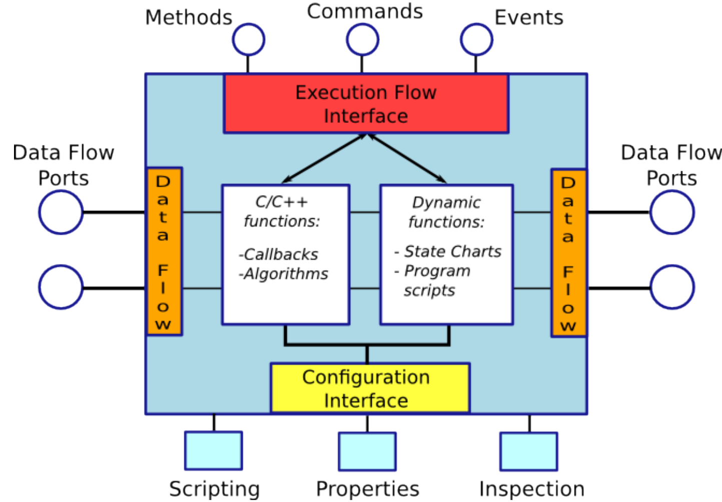

Figure 5: Structure of a component(source: Orocos component builders manual[2])

A component can execute state machines and/or code that is executed everytime the component is invoked. The invocation occurs periodically for a periodic component, or after some event based trigger for an aperiodic component. The nature of the component i.e. whether it is periodic or event triggered, is specified either in an Orocos configuration file or in the source code itself. Each component has functions that are executed on component startup, shutdown and in case of errors. The contents of these functions are user defined, as is the code that runs on each invocation. Every component has a 'configuration interface' which can be used to edit the metadata of the component. Metadata includes component properties and attributes. Properties have the advantage of being writable to an XML format, hence can store 'persistent' state. For example, a control parameter. Attributes are lightweight values which can be read and written during run-time. In addition to the configuration interface, two more interfaces are present: the execution interface and the dataflow ports. The execution interface provides means to control the component's execution and invoke functionality/services offered by a component from another component. Through this interface, the component may be started, stopped or its services may be invoked either synchronously, asynchronously or both ways from within another component. The dataflow ports allow transfer of data from one component to another. A port may be a buffered port or an unbuffered port and is meant either for reading data from, or writing data to. The read port of a component is generally connected to the write port of another component and the data types flowing through such connected ports must be the same. A detailed

description of the component interfaces is beyond the scope of this document. The interested reader is referred to [2].

4.2.5 Emergence of system behavior

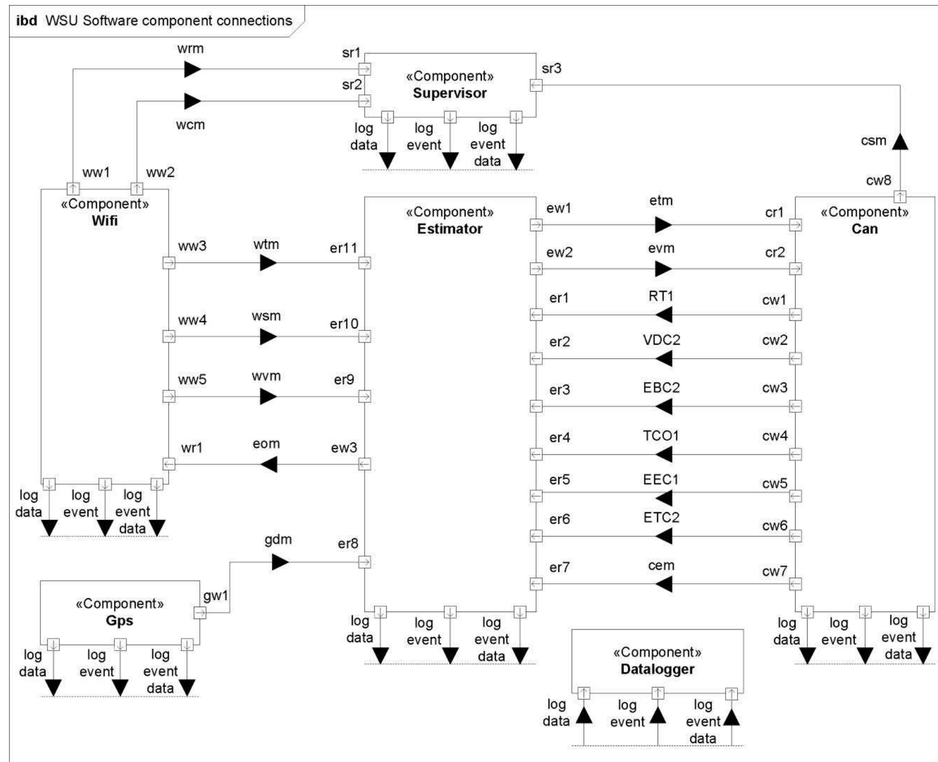

Figure 6: Dataflow depicted using SysML internal block diagram

A big portion of our architecture consists of a set of connected components within the generic computer, which exchange data. Let us use on this set to commence the understanding of system behavior. Refer to figure 6.

All the components in the generic computer are periodic. In every execution period, a component wakes up, does a specific task and goes back to sleep. However, before going back to sleep, a component services requests that may have been made by other components while it was asleep. The components behave as follows, in every execution cycle:

1. The GPS, CAN and Wireless components read data from their respective information sources and send it to the estimator. The CAN component reads data from the ECU as well as the vehicle CAN bus.

2. The Estimator uses its input data to update state vectors for the ego vehicle, surrounding vehicles and road objects. This information is then sent to the CAN component.

3. The CAN component forwards the estimator information to the Control component in the ECU.

4. The Control uses the information to enter/stay in an appropriate control strategy and influences the vehicle actuators. Information about the ECU actions is sent back to the CAN component.

5. The Supervisor component is responsible for system initialization, monitoring the status of other components, rerouting data-flows in case of component malfunctions and system level error management. It maintains the operational state of the overall system.

6. The Wireless component also periodically reads data from the estimator and broadcasts it over the wireless interface.

7. All components send logging data to the Logger component which periodically serializes it through a TCP socket to a separate computer. This separate computer logs the data to disk and also extracts relevant data and presents on a graphical interface.

Now let us see how this process works in a typical scenario. Assume that the vehicle is standing at a red light, and should start moving when it turns green. The following sequence of events takes place

1. The GPS reports current vehicle coordinates to the estimator

2. Information about the position and state of the traffic light is obtained by the wireless component and sent to the Estimator

3. The Estimator calculates the distance to the traffic light, the color and the vehicle position and sends this information to the Control via the CAN component

4. The Control component continuously monitors this information and keeps the brakes engaged while the traffic light directly ahead is red and the vehicle is within a threshold distance to the light

5. When the Control component receives information that the traffic light has turned green, it disengages the brakes and starts accelerating the vehicle forward

5 Datalogging in component based architectures

Data generated by or flowing through a computer based system often needs to be logged. The logged data is an 'audit trail' for determining what is/was going on in the system at any given time. Additionally, logging data provides an insight into the system's operation which is useful for debugging the system during its development. A part of the data being logged may also be useful for the purposes of Human Machine Interaction

(HMI) if it can be displayed to the operator in real-time. For these and other reasons, datalogging is an interesting problem within systems development in general. In this section we look at some principles of datalogging in component based, real-time systems and describe a datalogging solution that was employed within the Scoop project.

5.1 What data should be logged

This section describes the classification of data to be logged, as applicable to the Scoop architecture.

Data in a component based framework may be divided into data that is transferred from one component to another and data that is internal to a component and need not be transferred to other components. The bulk of the data is often numbers. Sensor measurements, calculated values, state information etc. These numbers can be plotted as time series data during analysis, or in a live display. Additionally, there may be text data that needs to be logged. This is the case of error messages, warnings, status reports, HMI messages etc. Often, text messages include a large amount of numeric data, like debug information strings. Text messages though, are not generally plotted continuously as time series data and need different handling mechanisms than purely numeric data. Furthermore, it is a common practice to assign 'log levels' to messages. A log level may be used to indicate levels of verbosity, severity etc. Log levels can be enabled/disabled, such that only messages belonging to the enabled log levels are logged.

5.2 Principles of datalogging

This section describes the principles of datalogging followed by the Scoop framework.

1. All the data to be logged can not be decided during the first iteration of the design. The logging infrastructure must make it trivial to add new data to be logged.

2. The data logging component(s) are non-realtime, but they must exist in harmony with other real-time components. Harmony means that the dataloggers should not adversely affect the operation of realtime components. Data loggers are considered non-realtime because they write the data to physical storage media and ensuring realtime performance for the associated input/output processes is technically difficult, and not worth achieving if the architecture can accommodate non-realtime datalogging.

3. All components should be capable of operation regardless of the state of the logging infrastructure. This implies that a complete breakdown of the logging mechanisms should not affect the functioning of components.

4. The components should be able to send the log data in 'fire and forget' mode. The sending component should not care about or be affected by a failure to send

data to the logging mechanism, but failure to obtain log data must be detectable at the logger end.

5. All log data should be timestamped to identify the time of its origin within the framework. The sender of the data must be clearly identifiable.

6. The logging system should not have to be aware of the data it is logging, its content, structure, or origin, except for a minimum interface for operations on the data (serialize/deserialize).

7. If a component is sending data to another component, it should not have to repackage the data in a different manner merely in order to log it. Thus, the data objects sent to other components must be accepted by the logger without modification

8. The data being logged must be opaque and extensible. Opaque means that it should not be necessary for the framework to have a knowledge of the data structures involved. Extensible means that the data structures used by a component for log data must be easily extensible to accommodate more (or less) data.

9. It should be possible to serialize the log data out of the system via a network socket, so that the task on analysis and display of the data can be carried out on a separate computer, if necessary.

5.3 Datalogger implementation

To understand the implementation of the data logging mechanism, first take a look at figure 7.

The datalogger component is a non-realtime component, which is connected to all other components by means of buffered 'many-to-one' data ports. There are three data ports in this design. One for purely numeric data and the other two for text data. The ports for the text data are respectively for HMI related messages (information that needs to reach the driver as soon as possible) and status information, including errors and warnings. The usage of two distinct ports for text messages is purely for convenience in implementation, and the two ports may be combined into one from a theoretical viewpoint.

Each component pushes its log data into the buffered port connected to the datalogger and this data is popped by the datalogger component. If the buffer is full, the pushing of data to the port fails silently. The datalogger can check the size of the port during each pop operation and raise an alarm if the buffer is full. Thus the design satisfies the principle that the writes fail silently, while the failure can be detected at the read end by the datalogger.

The datalogger is a periodic component that pops data from the buffer ports during each execution cycle. Ideally, the entire data is popped, but it is possible to preempt the pop operation at any time and resume it during the next execution cycle. The popped

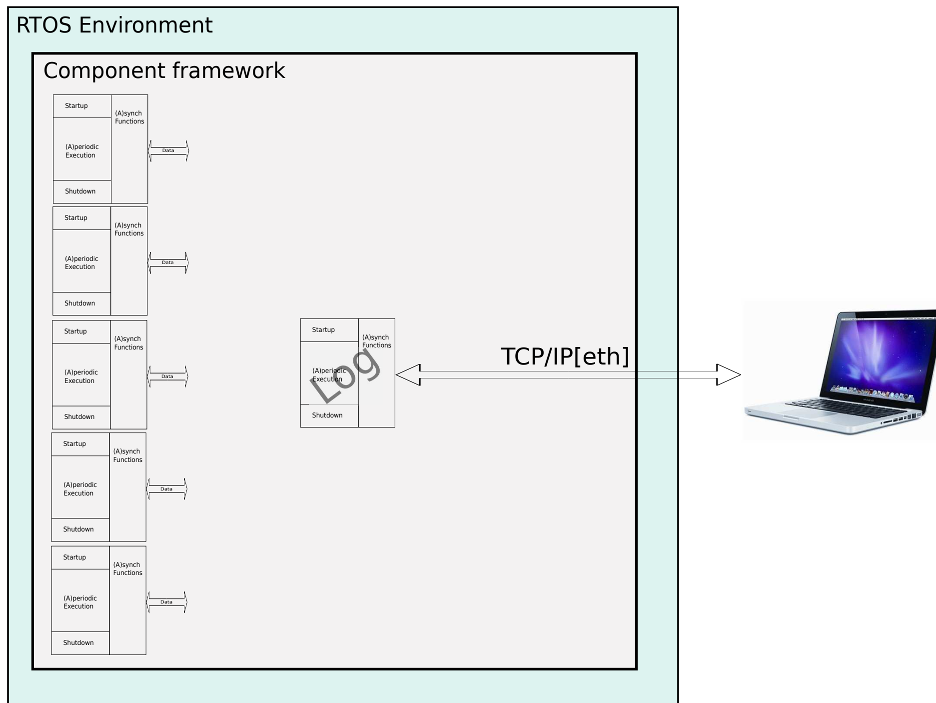

Figure 7: Datalogging

data is then serialized out of a TCP socket by the logger in a low priority, asynchronous thread.

The actual code for transferring data to the datalogger (via the buffer ports) is implemented in a single header file, with templated C++ logging functions. The implementation of each component includes this header file, thus preventing reimplementation of the same code in all components, as well as guaranteeing a uniform interface to the datalogger. The logging functions also insert metadata like timestamps and sending component into the log data.

The data to be logged is transferred through the dataports in the form of google protobuf objects[5][10]. Using google protobufs gives the following benefits

1. The data object has the same 'signature' (data type), regardless of actual content, since all google protobufs are derived from the same base class, which can be used as the object type. Thus, as far as the datalogger is concerned, it is receiving protobuf objects, while the sending components may send any derived object.

2. A google protobuf object is arbitrarily extensible with new data

3. The protobuf object contains methods that are suitable for (de)serializing the data it contains. The datalogger component can call these methods of the received

[5]A google protobuf is an implementation of a data object that contains methods to (de)serialize its datacontent.

object and thus serialize the data, without knowing any further details about the data. At the receiving end, the serialized data can be deserialized to recreate the protobuf objects using provided library functions. This solves the problem of serialization, transmission and deserialization across different computers.

4. The framework already uses protobuf objects to exchange data between components. These same objects can be sent to the datalogger without any modification.

During each execution cycle, the datalogger empties the buffer ports and calls the serialize methods of all objects popped from the dataports. The serialize method transforms the object to a stream of binary data. This datastream is then sent out of a TCP socket through an asynchronous thread. The datastreams are buffered in the datalogger to ensure that their transmission is immune to preemption of the transmitting thread.

The stream of log data objects is received on another computer at the other end of the TCP socket. There, the data may be filtered, logged and/or sent to a HMI. This processing of the log data is on a separate computing node and does not affect the realtime component based architecture in any way.

Some more text about the following may be added if a publication is to be made out of datalogging alone

- realtime components send to non-realtime logger. An option is to use distributed middleware like CORBA and directly call logging functions with datastructures to be logged as argument. But even in this case, the packaging of data is vital to avoid excess overloading of logging function definitions. Ideal is to have a single opaque data object that can be used for all types of data. If the framework allows it, one can embed executables inside the object which can manipulate the objects data. For example, have a serialize function.

- How to log the data? what formats? it depends on how you plan to analyze it. Generally used to plot 2D graphs. However, scientific analysis possible. There are tools like ROOT (from CERN) and qtiplot

6 Testing the system

Testing the system is generally done by following a prepared set of test cases. However, in the Scoop project, an applicable set of test cases was not available at the time testing was initiated. Therefore, the testing methodology of Scoop followed a logical approach from a technical perspective. The approach consisted of the following steps:

1. First the framework facilities were validated. The framework claimed to offer a set of facilities like component creation, execution of state machines within the components, the ability of synchronous and asynchronous communiciation among the components and the ability to transfer data among the components using buffered and unbuffered ports. All these facilities were tested with the help of

dummy code. Doing so gave the high level of confidence needed in the framework and also led to a better understanding of some of the framework behavior that later proved vital. For example, some initial assumptions regarding the Orocos asynchronous communication proved invalid, leading to a breakdown of the datalogging system under heavy system load. Once the communication system was better understood by testing it out using small, dummy examples, the datalogging system was redesigned and performed satisfactorily.

2. The daemons and external libraries that would be used in the system were then tested in scenarios similar to those that would occur during normal and abnormal conditions within the system. The *gps daemon* was made to communicate with the gps device at 10Hz and 100Hz. It was verified that the daemon as well as libraries that would bring the daemon data into the system could handle the resulting data rates. Then the *ntp daemon* was then tested to verify that it could synchronize the system clock with the gps signal. The *calm daemon* that handled the wireless communciation was then tested under various data rates and loads to determine the delays and packet losses that would occur. The CAN libraries were tested to make sure that they could handle the comunication between the vehicle CAN bus and the dedicated CAN bus to the ECU. Careful notes were made of the results of these tests and it was ensured that the concerned developers had a good understanding of how the parts under their responsibility worked. Doing these tests proved beneficial because several unexpected behaviors of third party code came to light. For example, it was found that the send() function of the CAN driver took a non-deterministic amount of time. This disrupted CAN communications completely until a clever workaround was created. (The vendor of the library, when contacted, was as confused as we were).

3. Once the framework and supporting daemons and libraries were verified working, it was time to start testing the system implementation. To do this, the first step was the testing of the datalogging framework. Log messages were sent from all components to the datalogger component and it was verified that that the logger properly sent the messages out of the system. The messages were reconstructed on another computer and the tools for visualizing and analyzing the data were tested. Only after it was verified that the logged data could be properly replayed and analyzed, did the rest of the system tests start.

4. Once the datalogger was verified working, the input/output (i/o) components were tested. I/O components are the components responsible for getting data into or out of the system. The GPS, CAN and Wireless are the i/o components in our design. It was verified that these components could reliably get and send data as per the specification. During this phase, the i/o blocks of the Simulink models running inside the ECU were also verified. Once this is done, then there is a degree of confidence that data reception and transmission is happening properly and it is time to focus on the components that consume the data.

5. The 'consuming' component on the general purpose computer is the Estimator. This was the next component to be tested. Initially it was verified that the state estimation of the Ego vehicle was happening properly. Then the same system was deployed on multiple real vehicles (so that multiple vehicles were transmitting reliable ego data). The next step was to test the estimation of state vectors of surrounding vehicles, based on the data they were transmitting. The final step in testing the Estimator was the verification of the world state vector. This includes variables like the current speed limit and the state and location of traffic lights.

6. After the Estimator is verified, it was time to focus on the ECU and control algorithms. At this point, it was verified that the rest of the system was functioning as per design and the ECU need be the only object of focus. Testing of the ECU involves testing the ability to form a platoon and control the longitudinal vehicle motion based on the data received from other vehicles in the platoon. It also includes testing the functions of stopping at red lights, taking off at green lights and respecting the speed limits.

7. At this stage, all individual components of the system, as well as their integration was verfied working. It was time to test the Supervisor component to make sure that it could start and stop the other components, monitor their health and shutdown the system cleanly when necessary.

8. Once all the tests described above are done, then a system FMEA took place. During this FMEA, possible failure modes, their effects and probability of occurrences were discussed by the project team. The results of the FMEA were then gradually incorporated by making incremental changes to the design and followed up by relevant tests.

7 Conclusions and reflections

The Scoop project had fairly narrow goals, within the broader context of cooperative driving (which in itself is a narrow area within the broader context of Intelligent Transport Systems). As such, the architecture designed to fulfil its goals was not as broad and all-enveloping as generic ITS architectures. However, it did its job extremely well. During the week long GCDC challenge, the system did not hit a single architectural limitation, despite needing several modifications due to last minute rule changes. Neither did the architecture ever fail to deliver during the challenge. In fact, once the architecture was in place, it could safely be ignored and the development focused on the core algorithms.

The architecture of a system reflects not only the technical constraints it is designed under, but also the skills, capabilities and choices of its designers and implementers. The architecture of the Scoop project was initiated with a heavy emphasis on operating systems and programming. While this is a valid approach, it will quickly disintegrate if

the development team lacks a solid, practical understanding of operating systems and programming. An alternative approach is to emphasize on model driven development and code generation tools. For example, it is possible to develop a lot of functionality is Simulink and simply use industrial tools to execute the Simulink models on micro-controllers. This approach does not require detailed operating system or low level programming knowledge, but the gain comes at the cost of limited flexibility both in the development process, as well as in the solutions achieved. The ideal lies in combining/balancing the two approaches, utilizing the strengths of each approach where appropriate. Ultimately however, the achieved balance will depend on human factors, rather than strictly technical ones.

Finally, it should be noted that a good architecture is a necessary but not sufficient condition for the success of a project. Almost equally important is the ecosystem surrounding the architecture. The ecosystem comprises of development and analysis tools, version control methods, documentation and best practices. As an example, almost 2 weeks of work on the Scoop project was nearly lost because some team members misunderstood the git version control system. The algorithms being used are often just as, if not more, important in the performace of the final system. [6] Something as non-technical as the license policy of a certain Simulink tool has a huge impact on development. We had to adopt a workflow where making the slightest change to the contents of the ECU implied uploading the entire model to on offsite server, compiling it and downloading the compiled result. At times, due to poor internet connectivity, this simply wasn't an option and it very adversely affected the development, testing and debugging workflows. Knowing such type of constraints in advance may help the architect design an architecture that relies less heavily on such 'problem areas'. Technology is rarely the main deciding factor.

8 Appendix A: Dataflows

This appendix provides a high level overview of the dataflows between the components of the architecture. A detailed, implementation level description of the dataflows between various components of the architecture is beyond the scope of this document.

When reading the dataflows presented in this appendix, it would be useful to refer to Figure 6. For convenience, that figure is shown again here, as Figure 8.

[6] However we don't consider that in this report because that is out of its scope.

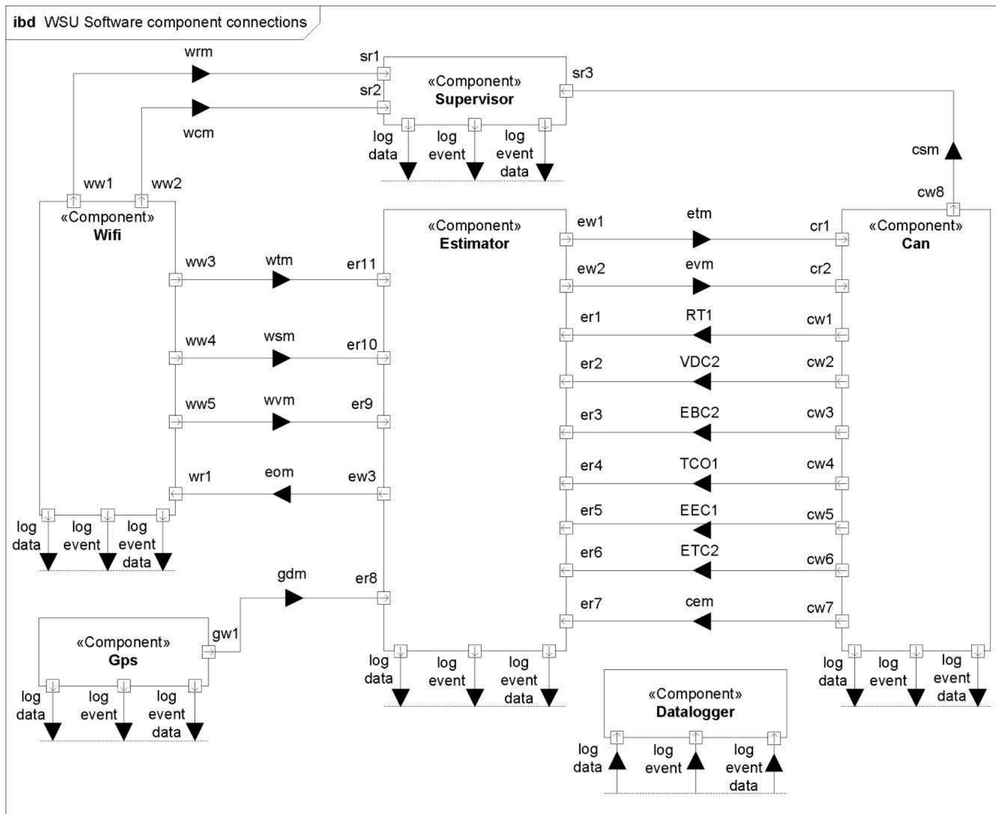

Figure 8: Dataflow depicted using a SysML internal block diagram

Port	Message	Write to	Content
ww1	wrm	Supervisor	Response received to ego vehicle's join platoon request
ww2	wcm	Supervisor	Information about the current challenge state
ww3	wtm	Estimator	Traffic light information
ww4	wsm	Estimator	Speed limit information
ww5	wvm	Estimator	Information about other vehicles in the vicinity

Table 1: Wireless write ports

Port	Message	Read from	Content
wr1	eom	Estimator	Estimated information of ego vehicle

Table 2: Wireless read ports

Port	Message	Write to	Content
ew1	etm	CAN	Estimated information about traffic lights
ew2	evm	CAN	Estimated information of all platoon vehicles
ew3	eom	Wireless	Estimated information of ego vehicle

Table 3: Estimator write ports

Port	Message	Read from	Content
er1	RT1	CAN	Vehicle radar sensor information
er2	VDC2	CAN	Vehicle dynamic stability control message
er3	EBC2	CAN	Vehicle wheel speed message
er4	TCO1	CAN	Vehicle tacograph message
er5	EEC1	CAN	Vehicle engine controller message
er6	ETC2	CAN	Vehicle transmission controller message
er7	cem	CAN	Information of ECU platooning algorithm
er8	gdm	GPS	GPS data
er9	wvm	Wireless	Information about other vehicles in the vicinity
er10	wsm	Wireless	Speed limit information
er11	wtm	Wireless	Traffic light information

Table 4: Estimator read ports

Port	Message	Write to	Content
cw1	RT1	Estimator	Vehicle radar sensor information
cw2	VDC2	Estimator	Vehicle dynamic stability control message
cw3	EBC2	Estimator	Vehicle wheel speed message
cw4	TCO1	Estimator	Vehicle tacograph message
cw5	EEC1	Estimator	Vehicle engine controller message
cw6	ETC2	Estimator	Vehicle transmission controller message
cw7	cem	Estimator	Information of ECU platooning algorithm
cw8	csm	Supervisor	ECU status, platoon role and state, speed intent

Table 5: CAN write ports

Port	Message	Read from	Content
cr1	etm	Estimator	Estimated information about traffic lights
cr2	evm	Estimator	Estimated information of all platoon vehicles

Table 6: CAN read ports

Port	Message	Write to	Content
gw1	gdm	Estimator	GPS data

Table 7: GPS write ports

Port	Message	Read from	Content
sr1	wrm	Wireless	Response received to ego vehicle's join platoon request
sr2	wcm	Wireless	Information about the current challenge state
sr3	csm	CAN	ECU status, platoon role and state, speed intent

Table 8: Supervisor read ports

9 Appendix B: Additional questions

The following questions could have been discussed in the report, but were ultimately not discussed. This is because it was felt that although the topics are interesting and relevant, answering them requires research of a different nature than that conducted within the confines of the Scoop project. It is very good to have interesting questions but the unless concrete work has been towards answering them specifically, bringing them up in the report adds no value. Nevertheless, they are listed here for reference in the hope that it may be possible to conduct further research in an attempt to answer them.

1. Comparison with CVIS/SAFESPOT/HAVE-it architectures
2. How is an architecture design started?
    - Traditional approach is to define requirements, use cases and test cases – Top down
    - Also common to jump on the keyboard and start doing something – bottom up?
        - Where do top down and bottom up approaches meet? Can something be done to ensure they meet up at desired place?
3. Differing types of requirements. Technical, functional, behavioural, non-functional. How and who should manage them such that they actually contribute to the project constructively, instead of being dead documents?
4. How will the architecture be documented? What is the minimum set of docs, or types of documentation needed to completely describe the architecture?
5. How to handle cross-cutting non-functional aspects like safety, timing requirements etc.? Where do they factor in into the design?

References

[1] URL http://pubs.opengroup.org/architecture/togaf8-doc/arch/toc.html.
(Cited on page 2.)

[2] URL http://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/v1.10.x/doc-xml/orocos-components-manual.html.
(Cited on pages 2, 17, 18, and 19.)

[3] URL http://en.wikipedia.org/wiki/Framework.
(Cited on page 2.)

[4] URL http://en.wikipedia.org/wiki/Software_framework.
(Cited on page 2.)

[5] URL http://www.orocos.org.
(Cited on pages 2, 10, and 17.)

[6] URL http://www.gcdc.net.
(Cited on page 3.)

[7] URL http://www.tno.nl.
(Cited on page 3.)

[8] URL http://www.dwheeler.com/oss_fs_why.html.
(Cited on page 10.)

[9] URL http://www.boost.org.
(Cited on page 11.)

[10] URL http://code.google.com/p/protobuf/.
(Cited on pages 11 and 23.)

[11] URL http://www.boost.org/doc/libs/release/libs/serialization.
(Cited on page 11.)

[12] URL http://gpsd.berlios.de/.
(Cited on page 12.)

[13] URL http://en.wikipedia.org/wiki/Ntpd.
(Cited on page 12.)

[14] URL http://www.xenomai.org.
(Cited on page 16.)

[15] The gcdc communication stack documentation. URL http://www.gcdc.net/mainmenu/Home/technology/Communication_Stack.
(Cited on page 10.)

[16] Gcdc rules and technology document. URL http://www.gcdc.net/mainmenu/Home/technology/Rules_and_Technology.
(Cited on page 3.)

[17] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. URL http://dx.doi.org/10.1145/1134285.1134292.
(Cited on page 4.)

[18] H. Heinecke. Automotive System Design - Challenges and Potential. In *Design, Automation and Test in Europe*, pages 656–657. IEEE, 2005. URL http://dx.doi.org/10.1109/DATE.2005.79.
(Cited on page 4.)

[19] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering, 2007. FOSE '07*, pages 55–71, May 2007. URL http://dx.doi.org/10.1109/FOSE.2007.22.
(Cited on page 4.)

[20] V. Schulte-Coerne, A. Thums, and J. Quante. Challenges in Reengineering Automotive Software. pages 315–316, March 2009. ISSN 1534-5351. URL http://dx.doi.org/10.1109/CSMR.2009.27.
(Cited on page 4.)

[21] Peter Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.
(Cited on page 10.)