Case Study: VCE Electronic Architecture for Real-Time Automotive Applications

Joakim Fröberg
Volvo Construction Equipment Components AB
Eskilstuna, Sweden
joakim.froberg@mdh.se

Abstract

The development of electronic systems within VCE faces challenges of shorter development time and keeping the electronics part of the product cost low. At the same time, electronic content and complexity is rapidly increasing. VCE has a wide range of products and also a wide range of technical solutions, processes, tools, and so on. Improved architecture design and a product line approach are envisioned to improve system properties and decrease cost by coordinating development.

This report describes the current architecture requirements, challenges, component model, technology, and development process adopted in the Sweden branch of VCE electronic development. This is a first step towards identifying challenges and solutions in a company wide survey whose aim is to produce solutions for the cost, coordination, and improved system properties effort.

1 Introduction

In this report, we present a case study of an architecture for the development of software product lines of heavy vehicle on-board electronic systems. Moreover, we present challenges that are faced in this effort.

The contribution of this paper consists of a current industrial method for platform development in product lines of automotive electronic systems. Platform development includes electronic systems architecture, component model, process, and methods for assuring quality properties of the system.

The contributions in this paper are:

- A case study of an automotive system that has been designed by using a component based, product line approach. The presentation includes a functional overview, architecture requirements, component model, design method, process, and challenges that are faced in this effort.

1.1 VCE business context

Volvo Construction Equipment, VCE, develops a variety of construction equipment vehicles. VCE is divided into a number of product companies and other supportive companies. The product companies focus on their specific products and are responsible for cost, deliveries, service etc to the end-customer. End-customers range from single vehicle owners to rental companies with hundreds of vehicles. Typically, a product company manufactures a product line of similar, but differently sized vehicles.

VCE has two departments for development of on-board electronic and software systems. One department is located in Sweden and one in Korea. These departments hold expertise on and develop on-vehicle electronic systems for various products to the product companies. The electronic development includes control systems, other on-vehicle software, and hardware. In the following this will be referred to as electronic development.

The product companies can also decide to buy electronics development from vendors external to VCE. Hence, the final products can include hardware and software components from many vendors both from VCE and external to VCE. From the product companies' perspective, VCE electronic development is essentially equal to any subsystem vendor and must develop control systems and software at competitive prices.

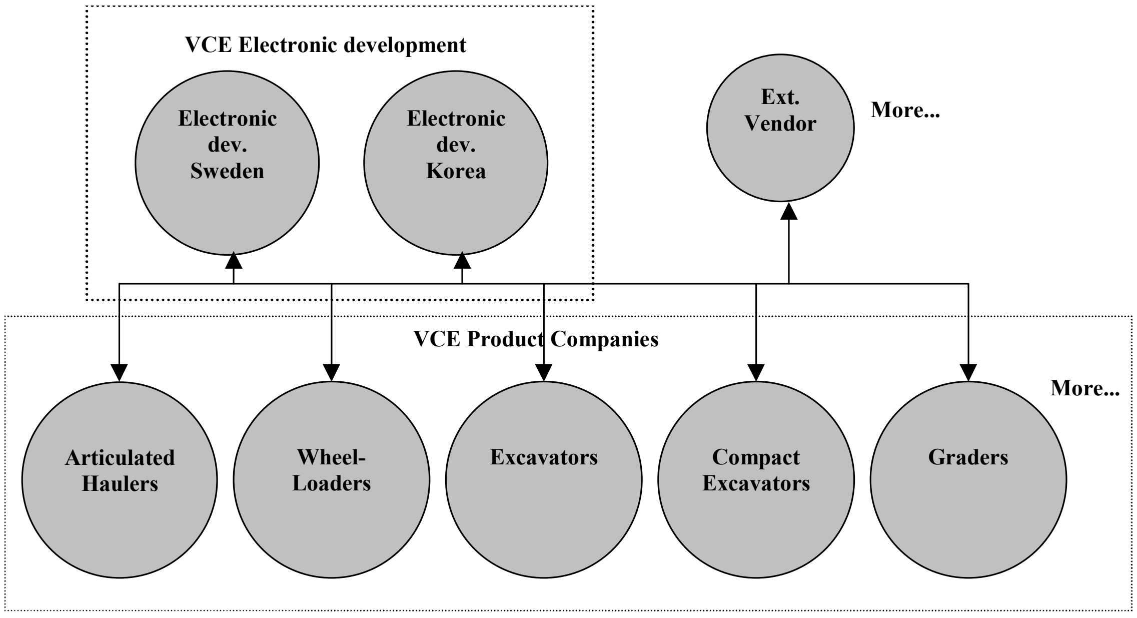

Figure 1. Electronic development in VCE

This report describes the VCE Electronic development and aims at identifying today's challenges and possible improvements.

The VCE Electronic development vision is to coordinate development in terms of processes, methods and technology, and thereby develop cheap, high quality electronic systems to the VCE product companies.

1.2 Challenges in electronic development

The VCE electronic development faces challenges related to a reduction in time and budget for the development of electronic systems for the various product lines. At the same time, the content and complexity of on-vehicle software is rapidly increasing. System properties like safety, reliability and real-timeliness must not suffer when quickening the development. There is also a need for scalable and configurable systems where the customer selects, and pays for, only the wanted features. The main focus when addressing these challenges is to find processes, methods and technology that will shorten development time and lower development cost, but also to satisfy an increasing demand for new functions and flexible configurations in the vehicle's electronic systems.

The main challenges in VCE electronic development relates to:

1. Cost reduction
2. Shorter and more predictable development time
3. Flexibility

In this section we describe these challenges in more depth, and elaborate on various aspects of the main challenges.

1.2.1 Functions and system properties

The vehicle's main functionality in terms of loading capacity and robustness is the key for many of VCE's end-customers. Therefore, it is often difficult to motivate even a small increase in cost for functions that does not obviously increase the vehicle's production. Examples include functions to improve ergonomics, safety and logistics e.g. various x-by-wire type systems, fleet management, and infotainment systems. This is a different setting, compared to passenger cars, where the vehicle is not only intended to make profit.

System properties that do not provide a certain function, such as maintainability, reliability, portability and configurability, are indirectly important to achieve a successful product. The system properties can be divided into operational properties and development properties. The development properties are usually not interesting to end-customers but must be successfully designed in order to keep development time and lead-time low. An electronic system that has a high ranking on development properties meets the developer requirements and is easier to reuse, further develop, and maintain. Thereby, VCE can decrease development time; achieve better profitability, and free developer resources for innovative progress.

Also operational properties such as reliability and safety must be incorporated in the system design to achieve a successful system. Operational properties, as opposed to development properties, are interesting to end customers but typically they are hard to assess. For example, a system could prove more reliable if we design it with redundant hardware components or fault tolerant software design, but still it would be hard to assess the achieved increase in reliability. Traditionally, VCE electronic development "delivers" functions to product companies. As the complexity of electronic systems grow, the need for well defined and measurable system properties grow, and thus also the time and cost for achieving the wanted properties of the system. This means that an increasing part of the total development time is used to design of the system itself and not to design user functions. Finding a structured approach to defining, measuring, and designing system properties is one challenge. Another is to communicate requirements and fulfillment with product companies that traditionally order functions.

1.2.2 Varying requests

Still, end-customers (and thereby the product companies) request a variety of new functions and system properties, but the requests are often different between customers. Therefore, a company that successfully introduces a configurable product and optional functions could support a wider range of customers. However, optionability is a system property and will require a design effort and logistic process without providing any end-user functions. The development process and the technical solutions must accommodate the options at a reasonable cost.

For example, a system with a large number of optional functions must be tested for each feasible combination. As the number of optional functions grows large, the total number of combinations to test could yield an extensive overhead cost. Also, all other parts of the development process must be adapted to handle the variety of configurations. Especially, an optional function that is rarely bought by customers could prove unprofitable although the function itself may be trivial to implement.

On the technical side, including optionability in an on-vehicle system might lead to increased product cost. Including infrastructure to support the optionability could mean overcapacity in CPU, memory and network bandwidth, and thereby increased product cost.

1.2.3 Known problems

Here we present some assorted problems known in the VCE electronic development domain.

- **Assessing benefits of advanced control systems.** Example – engine power. Construction equipment end-customers are not traditionally experienced in judging the benefits of a complex electronic system. Traditionally an important parameter for assessing a vehicle is the engine power. Using advanced electronic control systems can increase the efficiency of the vehicle and therefore lessen the importance of a high engine power. Examples of possible efficiency increasers are; control system for lifting the bucket (lift-by-wire type system), and control system for the gearbox.

- **Communication system benefits.** Communication systems require that the vehicle will be used in an environment where the owner can benefit from them. Some user might want GSM links and some satellite link depending on the intended location for the vehicle. A fleet management system, for instance, may increase production for a rental company with many vehicles, but may be considered worthless for a single-vehicle owner. Examples of communication functions are; anti-theft function, fleet management, and remote diagnostics for service shops.

- **Methods for estimating life cycle cost.** Hardware product cost is critical throughout all VCE product lines. This includes sensors, wiring, actuators, CPU, memory, I/O, casing, displays, and input devices. This would imply that a specialized and minimal hardware set would be fitted in each separate product. However, the total product cost depends also on development cost, maintenance cost, and more. Really the life cycle cost. A custom set of hardware components for each product would bring on a high development cost, a high maintenance cost and thereby a high product cost. In order to find an optimal solution, all costs in the product life cycle, production volumes, and time-to-market issues must be considered.

- **Scalability.** Typically CPU, I/O, and memory are designed as a box to host a number of software functions. These computational resources must be kept scarce to keep product cost low. However, reusing hardware is wanted to keep development cost and maintenance cost low. Reusing hardware for a new product may mean to include an unnecessary amount of resources. Software functions are often relatively feasible to exchange or remove, but changing hardware resources often requires design of a new box. To accommodate a common set of technology, methods and tools in this context is a challenge.

- **Openess and integration.** A number of issues have arisen from integrating external vendor nodes. How to handle reliability, safety, and liability?

  1. External vendor nodes could short circuit or overload the bus and VCE is responsible for product safety. This yields questions of redundant and/or secure (e.g. only control system nodes) busses.
  2. An external node could malfunction due to wrongly interpreted messages. This presents a difficulty in defining responsibility. Vice versa if a VCE node is dependent on an external node by reading its messages.
  3. How to assure network bandwidth? Considering all VCE and external nodes (and different versions and configurations) and their worst-case network usage presents a challenge. Tests cannot fully assure network bandwidth.

- **Maintenance and configuration management.** The on-vehicle system is a distributed system with versions of node hardware and software. If a bug is reported, upgrading all hardware and software in a faulty vehicle is expensive. One question that arises is “Which versions are compatible among the large number of possible configurations?” Also, if a certain configuration is found to host a bug, all products with this configuration should be upgraded. This leads to logistical challenges.

1.3 Approach to address challenges

One way of reducing time spent on development is to reuse software components and architectural solutions between products. The ability to reuse components would put a high demand on a component model and a coherent approach to architecture design between products. A common electronic product line approach, including process, methods, and tools could address several objectives in system development, like time-to-market, development cost, and maintenance cost [1]. An improved architecture design method would aid in the effort of improving the quality attributes of the system e.g. configurability and reliability. Reusing components can also improve the component reliability since code is executed for longer time and tested in other contexts [15].

Applying a product line approach when developing a product family can greatly decrease the time spent in a development project once a common architecture and asset base has been developed [5].

1.4 Terminology in this report

Developing a system from scratch is not the most common type of development today. Companies often develop new systems in an evolutionary way i.e. based upon previously developed systems. Typically a company also develops a *product line* i.e. variety of related systems. The objectives of using a structured method for developing product lines is to shorten time-to-market, reduce development cost, and to reduce maintenance cost. To address the issues of developing product lines, many companies make use of application *platforms*. A platform is a base of core assets that can be used for several products. This includes architecture, software components, specifications, test plans, development process, and more. A *product line approach* to development is an effort to create an overall development process taking into account a whole product line. The aim is to avoid sub optimization and lift the focus from single products. By focusing on the *software architecture*, developers want to get a high-level view of the system's properties. The architecture should allow for reasoning and evaluation of system properties. The architecture is an important core asset in the platform. There exists several definitions of software architecture and one of the most reoccurring is;

"The software architecture of a program or a computing system is the structure or structures of the system, which compromise software components, the externally visible properties of those components, and the relationships between them." [7]

Software architecture design is a method intended to assess and achieve a number of *quality attributes* like maintainability, reusability, scalability. The requirements for quality attributes are called *quality requirements*. Thus, the quality requirements do not state functional requirements on the system, but instead states what qualities the system shall exhibit. A software architecture design method aim at accommodating specification, measurement, and design of quality attributes in the system. The method should provide early assessment of quality requirements fulfillment.

A *Product line architecture* method is an overall strategy to address both the objectives of a product line approach and software architecture design i.e. achieving system quality attributes, reduce development cost, shorten time-to-market, and reduce maintenance cost. A product line architecture method includes several aspects of development.

- Technology
- Methods and process
- Tools
- Organization

2 Case-study

The VCE Electronic development in Sweden has developed a platform for electronic systems and it is used in a number of high-end vehicles. The platform includes aspects of architecture, component model, development methods, tools, technology, and process. Since the VCE challenges relate to coordination and cost reduction, these aspects of the platform is especially focused on. Nonetheless, this section gives a thorough explanation of the background, quality requirements, component model, architectural views, tools, development process, and especially important methods.

2.1 Functionality overview

A useful platform for product lines must support the variety of functional requirements that the different products impose. Ideally, the platform should, to some extent, also support future functions. Here we outline some of the most central functions that are supported by the platform.

The electronic functions in the vehicle can be categorized as the following.

- Monitoring functions
- Controlling functions
- Logging functions
- Communication functions

Monitoring functions measure a quantity via a sensor and provide feedback in the GUI by indicators or display. Controlling functions control physical devices via actuators and often make decisions by analyzing monitored values. Logging functions save data in permanent memory to be retrieved by analyzing or service related functions. Communication functions provide communication with systems external to the vehicle.

Monitoring functions are numerous in a construction equipment vehicle today. Oil levels, pressures, and temperatures are measured in various mechanical components throughout the vehicle. Examples are hydraulic fluid temperature, brake pressure, and transmission oil level.

The larger part of the controlling functionality, today, lies in the engine and the gearbox control systems. The engine functionality is not developed within VCE and is not described further in this report. The gearbox and the automatic shifting of gears are controlled by the electronic system. A gearbox contains a large number of mechanical components that are controlled by actuators and thereby the electronic system. These mechanical components can include some or all of the following, converter, lockup, drop box, retarder, clutches, and brakes. The functionality for an automatic gearbox includes logics for when to shift gears, minimization of slip, avoidance of hunting, various efficiency optimizations, and self-adapting solutions to accommodate a variance of mechanical properties. The gearbox control is one of the most complex devices to control in today's systems.

Other control functions include control of speed (speed restriction), windshield wiper, load body, parking brake, and cooling fan. Control functions also include system functions like cooperation of brakes to achieve increased efficiency. (Many vehicles have several brakes and the control system can decide which to use to minimize heat losses or wear.)

All functions that communicate with external systems are called communication functions. This includes all functionality that is dependent on communication technologies such as GPS, satellite communication, mobile phone technology, and also communication interfaces provided by connectors on the vehicle. Examples are, anti theft function, service tools, production tools, and fleet management.

Trying to guess the functions or types of functions that can be required in the near future is difficult, but here we present some ideas on what could come. On the control side, the future could very well include several x-by wire type control systems. Communication systems could include advanced fleet management like optimizing a fleet's movement with respect to fuel consumption or time. There can be anti-theft functions with vehicle immobilizer. Communication functions could also include various infotainment systems like streaming video or Internet access.

2.2 Quality attributes - quality requirements

The design of VCE's current architecture was done with the intent of using it for a relatively long time. The architecture was to be a base for development of several products over time. In order to be successful in the development effort, the wanted properties of the system were considered. The system must support the implementation of functional requirements for the various products, but it must also exhibit system properties such as scalability and reliability. These are the quality attributes and do not apply to a certain function in the system but are properties of the system as a whole [5]. The requirements for quality attributes are divided into development and operational quality requirements. Although, quality requirements are sometimes difficult to assess, they are often very important to achieve a useful architecture. For instance, a system must often be easy to change and maintain in order to be successful out of a business perspective. In VCE, the architecture was designed with the intent of meeting the identified quality requirements.

Deciding which quality attributes are important to an architecture is dependent on many aspects of the product life cycle, production volumes and business issues. For instance, the quality attributes of mobile phone-, construction equipment-, and salary system -software architectures probably differ substantially. In VCE, various experienced specialists in the product companies identified the important quality attributes. VCE develops systems in an evolutionary way, and the electronic development also has substantial knowledge on what quality attributes are important to end-customers.

The customers of construction equipment buy VCE's products with the intent of making business out of them. Therefore reliability, a low rate of service standstills and system malfunctions are often the main concerns when purchasing a construction equipment vehicle with all its embedded systems. Nonetheless, there is often a need for user specific configurations and flexibility in terms of extra functionality and features.

Out of the developer perspective, the system and its subsystems must be reusable in order to meet the demands on shorter development cycles and a larger variety of products. Moreover, scalability is a wanted attribute of the platform since VCE's products functionally range from quite low complexity to very complex. A scalable platform would allow for commonality in terms of technology, processes, methods and tools that are used in the development. There is also a need for maintaining the system, both with respect to service/bug-fixes and to further enhance and develop the system.

In VCE, the most important quality requirements on the platform were identified as,

Operational quality requirements:

- Reliability – End-customers must perceive the products as robust and reliable. This calls for design that can be assured to perform its function via prediction and testing.

- Safety – The architecture must allow for developing safety critical functions. Again accurate predictions on system behavior must be supported. Also, interrelations between safety-critical and other functions should be kept at a minimum. This is to support analyzing safety-critical design separately from other design.

- Real-timeliness - The response times of functions must be within requirements. The vehicle's control functions may be safety related or control expensive hardware. The architecture must allow for developing functionality with hard real-time requirements.

Development quality requirements:

- Reusability - The architecture must be usable in several products to make the development cycle short. Moreover, using a unified architecture would allow for easier reuse of components and therefore also help in the effort of reducing the time spent on development.

- Scalability - The architecture should be usable for several products where electronic system content varies. There are products that cannot possibly include the expensive hardware that is used in the most advanced vehicles because of the cost. The architecture must also be designed with a focus on scarce resources. Hardware cost is always kept as low as possible in on-board vehicle systems in order to keep product cost low.

- Configurability - The architecture should allow for user-adapted instantiations. A customer with specific demands should be satisfied feasibly.

- Maintainability – The architecture must allow for service download of software and feasible diagnostics.

Methods for identifying the most important quality attributes are based upon experience. Also, assessing the fulfillment of the quality requirements are based on estimates.

2.3 Component model

To accommodate reuse of software components and methodology between products, VCE has incorporated a component model for the real time application domain. The component model is an important part of the VCE electronic platform since it enables some reuse and commonality in terms of tools and methods.

2.3.1 Background

The types of embedded systems that VCE are developing have high demands on timeliness and can be characterized as real-time systems. Moreover, the systems must perform in an environment of limited hardware capacity. Hence, the demands on efficient resource usage are high. Common component technologies, such as COM, JavaBeans and CORBA, are considered unfit for use in the on-vehicle systems because of their excessive resource usage and unpredictable timing behavior. A large resource usage would lead to a high hardware cost for each system produced. The VCE electronic hardware is very sparse on resources to keep costs low.

Designing reusable real-time components is more complex and more expensive than designing non-reusable non-real-time components [16]. This complexity arises from several aspects of real-time systems not relevant in non-real-time systems. In order to reuse software components in this environment, not only structural and functional attributes must be considered, but also temporal properties must be intact when the component is reused. The development of real-time components that can be run on different hardware platforms is complicated by the fact that components will have different temporal behavior on different hardware. Thus, the goal is a reuse method that accommodates the use of components on different hardware platforms. For the future, VCE wants to expand the reuse of components to include new types of hardware.

2.3.2 VCE Current component model

The system functionality is decomposed into different modes of operation for different types of functionality. These modes include drive-, startup-, shutdown-, and reduced-mode. Specification of modes and valid mode transitions constitute a high level of system description.

The functions in each mode are decomposed into tasks. Tasks are defined by their low-level functions together with the data-flow between them. Each task has a number of typed in- and out-ports. A task executes by doing the following: read its in-ports, perform its function, and before termination, write the result to its out-port. A task is not permitted to communicate directly with another task. Each task also has a configuration containing its temporal properties, e.g., period time, deadline, release time and WCET. This construction is similar to the Pipes and Filters model, and implies that the tasks can be executed without knowledge of where the input data was produced or where the output data will be used.

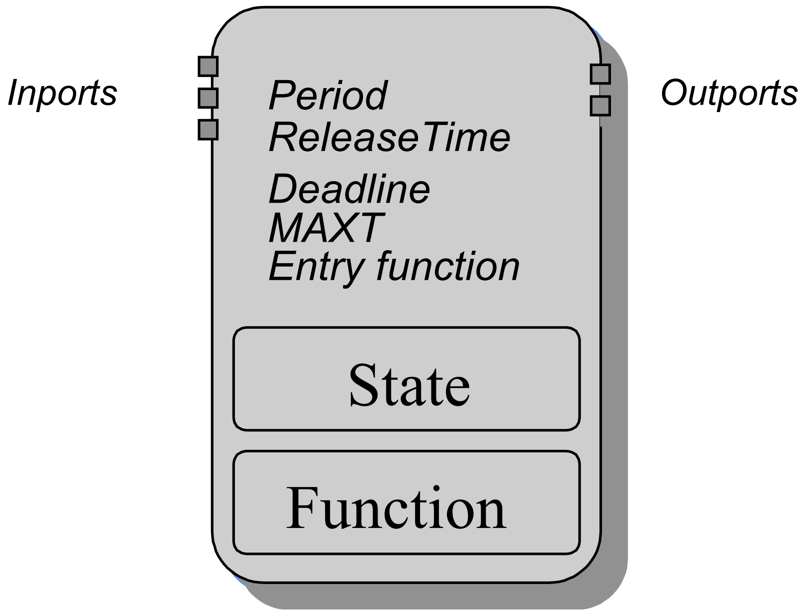

Figure 2. Task model

One or more tasks can be encapsulated in a component. The component is a logical bundling of the task, its properties and its interface. The component provides a way to logically tie together, task(s) source code, task(s) configuration and task ports. Together, these include the task functionality and its temporal properties. When we want to reuse a component in a different application, the task and its configuration can be kept intact and only its thin component wrapper needs to be modified. The loose coupling between tasks and the independent execution of tasks provides a means of easier reuse of components.

In order to be meaningful, no task set can be totally independent, but the VCE effort has been focused on minimization of communication and synchronization among components.

A composite is a logical composition of one or more components. The same thin configuration wrapper that constitutes the component wrapper achieves a composite. Composites and components are technically the same, but logically it has been deemed convenient to name an encapsulation of components a composite.

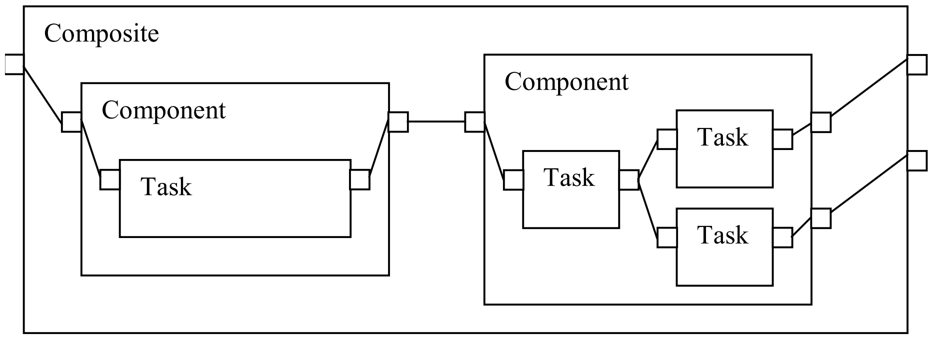

Figure 3. Component decomposition

So, by using the component model described above, a component or composite can be reused with both functionality and temporal properties intact. The communication part of the configuration may need to be altered when a component is reused in a new application depending on the new task set. However, this is only a matter of connecting the ports to their new producers/consumers. As an example, we can compare this task model with a traditional priority based task model. Reusing a task in the latter would force us to alter code for communication and synchronization. Thereby, the timing behavior might change and need further verification. So, by keeping the communication, synchronization, and temporal properties apart from the implementation VCE gains:

- Better and earlier prediction of timing behavior
- Easier reuse

2.3.3 Example - A Communication component

There is a communication component in the VCE architecture that could be reused in many applications. The communication component handles two communication protocols and has a large number of ports. One port for each message ID on the bus. The component is intended for use on all nodes in the system, and each node has an application with different communication needs. In order to accommodate many applications, the component must support communication via all the ports and at the same time perform within its time budget. The time budget is allocated by estimating a worst-case execution time, WCET, which is used in the component's configuration. Hence, given our model, the component must be allocated enough resources in terms of execution time to handle a worst-case situation with all the ports enabled. When the component is used in an application where only a few of its ports are connected, the execution time of the component is never close to the WCET and the processor utilization gets unnecessarily low. In this context, the resources are not abundant and low processor utilization cannot be accepted. To change the configuration of the component every time it is reused could be done, but is not considered feasible or safe. The WCET and the temporal configuration would have to be estimated depending on the number of ports that are required by a certain node. There is no tool support to do this today and it would have to be done manually.

In the VCE case, this communication component was intended for reuse on a computer node that has a task set with an especially heavy load of calculations, but not much communication. In this case the communication component could not be allotted its worst-case execution time and could not be reused without some alterations.

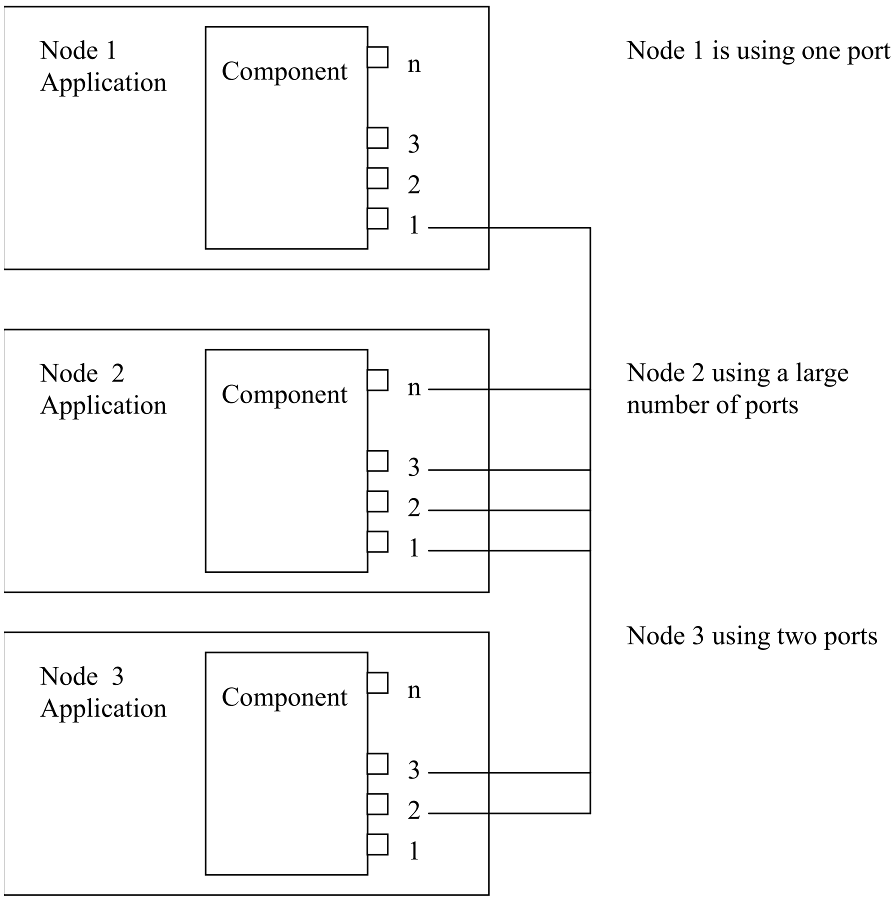

Figure 4. Communication component example

This exemplifies a general problem in reusing components in the VCE approach. Given the component model, the number of ports in a component cannot change from one application to another easily. By altering the number of ports, either the real-time properties of the component or the processor utilization will suffer. So, the current component model does provide means for reuse but also has limitations.

2.4 Architectural views

Wall [17], proposes a number of architectural views important to a system in the real-time system domain. In order to describe the VCE electronic architecture we use some of these views to illustrate the current architecture and its development. The concept of views is used to separate various aspects of the system apart from only its internal structure.

2.4.1 Hardware view

A number of nodes, electronic control units (ECUs), are connected to two busses. One bus is the SAE J1939, CAN, bus and the other is a SAE J1587 bus. The number of nodes differs in different products but is the same within product lines. Today a unified hardware is used for all nodes developed by the VCE Sweden branch except the display ECU. A unified hardware means, in this case, a common design built for relatively easy change of CPU, I/O, and memory, but primarily ease of change in I/O configuration.

The complete system includes nodes that are developed externally and do not use the same hardware. The ECU hardware is developed within VCE and includes processor, RAM, EEPROM, flash, CAN controller, analog and digital I/O, and drive circuitry. All encapsulated in a box designed to fulfill the environmental requirements i.e. electro-magnetic fields, vibration, moisture etc. Each node includes around 100 I/O channels and the software functions are distributed over the nodes.

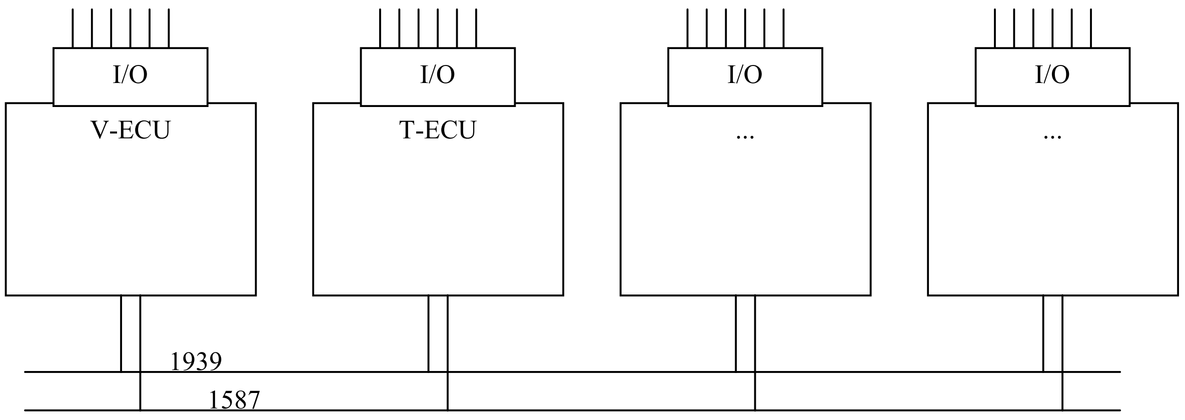

Figure 5. Hardware setting

When designing a new system, we can choose the number of ECUs depending on resource demand. This solution is not optimal with respect to achieving maximum resource usage, but many advantages are had with respect to reusing software, methods and tools. Software that is reused include drivers, communication software, service software, error handling. Also infrastructure software like layering functionality and watchdog software is reused. Tools like compiler, code generators, and scheduler can be used more easily due to the fixed hardware platform. Methods for parameterization of software and makefile are also reused.

2.4.2 Temporal & Synchronization view

The temporal view is the description of the system that deals with analyzing timing behavior. Therefore, this view partly overlaps with the synchronization view that deals with the concurrency control of tasks. Control of task synchronization is necessary to avoid simultaneous access to shared resources and to guarantee task precedence relations.

The system functionality is decomposed into tasks. Each task has a function that implements its behavior and also a state i.e. it can store data that will not be lost between task invocations. Apart from having a function and a state, each task has temporal attributes, precedence relations, and communication settings. The VCE development model supports specification of these properties and they all affect timing behavior of the task (and the system.)

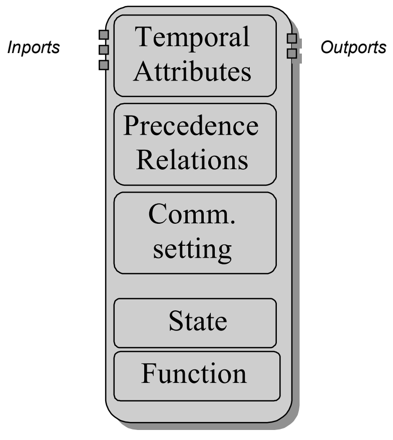

Figure 6. Task configuration

Each task has a set of temporal attributes. The temporal attributes are *period*, the *Worst Case Execution Time* (WCET), the *release time*, and the *deadline* of a task. The WCET is the execution time for the task. The release time is the earliest time at which the task can be activated, relative to its period start. The deadline is the latest time at which a task is permitted to terminate, relative to its period start.

Tasks can have precedence relations to other tasks. Any task can have a precedence relation to any other, but only when tasks are dispatched at the same time will the precedence relation be meaningful. Tasks with the same period time will always be dispatched at the same time. A precedence relation put a requirement on the concurrency of tasks during run-time. The preceding task will be executed before the preceded task.

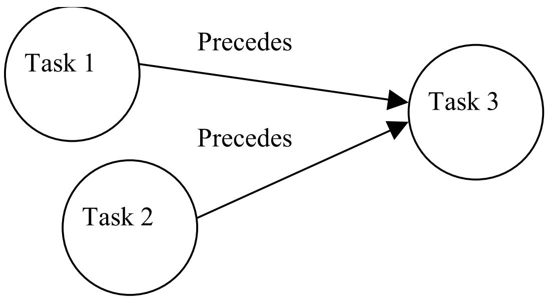

Figure 7. Precedence relations

In the VCE case, tasks are not allowed to preempt each other and thereby mutually exclude each other by a separation in time. Tasks start and finish execution before another task is started. Shared resources can, in this way, only be accessed by one task at a time.

Based on experience, the WCET for each task is estimated before the actual implementation and this allows early temporal verification. The task-set can be scheduled before the tasks are implemented and, assuming that the estimated time budget is not violated during implementation, the system is schedulable. Schedulability and timing analysis can, in this way, be performed early in the development. This leads to that the time budget for each task is also a functional requirement for the implementation phase.

Currently, the VCE system is quite interrupt intensive. Interrupts are used to manage bus messages and to read some sensor data. The interrupts have an impact on system timing properties. The temporal behavior for each interrupt is modeled with minimum inter-arrival time and execution time. VCE uses a method for scheduling interrupts with predictable real-timeliness kept intact [3].

2.4.3 Communication view

In the VCE model, a task has a set of typed in-ports and out-ports. All communication of the task goes through these ports. On activation, a task reads its in-port, perform its function, and write the result to its out-port. This construction facilitates a loose coupling between tasks and implies that each task can be designed independently of other producing or consuming task's design.

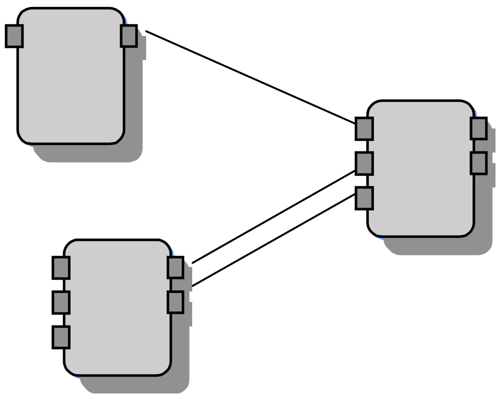
Figure 8. Communication flow

Exchange of data between tasks is performed by unbuffered communication (shared memory) and a port is an unbuffered communication interface. The port can always be read and written to, and this implies that the data can be overwritten at any time. Unbuffered communication is generally preferred in controller applications [10]. The other choice would be to have buffered interfaces for the tasks (message queues). Buffered interfaces can be more complicated in a real-time system, as upper bounds on produced/consumed data must be determined to predict timing behavior of a task.

Note the separation of communication relations and precedence relations. Tasks can be configured to communicate regardless of period times as opposed to precedence relations that can only be specified between tasks with the same period.

2.5 Run-time system & Tools

In order to be accepted by end-customers, on-vehicle control systems must meet stringent requirements on real-timeliness and reliability. This has lead to a pervading characteristic of robust design in the development of the control application. Many of the VCE tools are used to accomplish a predictable timing behavior and code execution. The software system performs various control and communication functions and functionality is distributed over the system nodes. Functions are implemented as tasks that often have hard real-time requirements. During run-time, tasks execute according to a dispatch table that is generated off-line. Soft real-time tasks are scheduled on-line and use the processor's spare capacity to execute. The application is also quite interrupt intensive.

VCE uses the Rubus operating system that supports hard and soft real-time tasks. A tool named Rubus Configuration Compiler, CC, supports the communication, synchronization, and temporal constructs described earlier. The configuration of the tasks is used by a scheduler that produces an off-line schedule and communication infrastructure for the operational mode. If the temporal properties of the task set cannot be fulfilled, the scheduler rejects the configuration.

Time triggered schedules were used in VCE before the introduction of the scheduler tool. The schedules were then made by hand and as the system complexity increased, making analysis and changes to the system became irksome. The scheduler tool performs an automatic analysis and provides feedback if a change violates the temporal constraints. Note that the scheduling is performed based upon available resources in terms of CPU capacity. (Each task's WCET is dependent on the CPU speed) One could envision a broader method that considers other resources like network or perhaps memory.

The Rubus operating system supports the use of semaphores and preemption (even nested preemptions), but these functions are not used in the VCE approach. Using preemption and semaphores would make scheduling easier, but would integrate the design of timing characteristics in the code implementation phase.

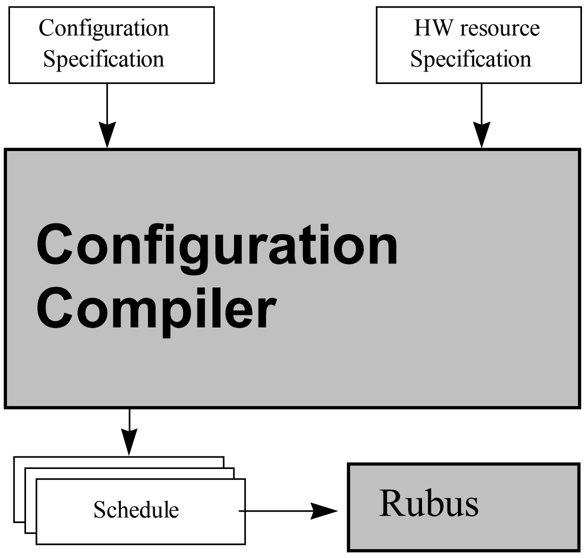

Figure 9. Configuration compiler

Input to the CC is both the configuration specification of the task set and the specification of hardware resources. The tool generates code to accommodate automatic mapping of the specification to a resource structure i.e. hardware resources.

2.6 VCE mechanisms for product variability

The products within a VCE product line are often not functionally or physically identical. Instead, the exact functionality is controlled by variants and options. This stems from the fact that customers request tailored systems to suit their specific needs. Logistically it would be convenient to provide only one type of physical product and provide the variation by software. This is done in many subsystems today, but the product volumes and the cost of mechanical and hardware components make this approach infeasible for some variable functions. Some functions are provided as options where the product is fitted with the necessary mechanical, hardware, and software components. Deciding on whether to mount physical components onto the product or to let the function be an option that must be fitted in the aftermarket is not always easy. The separate costs for components, production, aftermarket, and production volumes should be taken into account.

Current VCE techniques for variability in electronic systems:

- Parameterization and variables
- Optional nodes

Obviously, product specific behavior can be achieved by developing separate executables for each product in the product line. Developing executables totally separate could prove tedious in terms of administering changes if the products have common functionality. Control by pre-processor directives is often preferred in software industry. Separate executables could be obtained by a series of conditional compile directives throughout the code and by inputting product specific flags in a makefile or in build commands.

Advantages of this method are its relative simplicity and also that it ensures rather small executables. (No extra code is included in the executable to accommodate generalization of components)

There are drawbacks to this method however. Firstly, the method has limited flexibility. The number of independent compile directives cannot grow large before the developers loose their overview. A change gives impact in so many products that mistakes are bound to happen.

2.6.1 Parameters

VCE uses parameterization as one way to accommodate variability. Parameterization is a technique that uses run-time switches for controlling program execution. Parameters are kept in permanent memory, E^2. E^2 is logically divided into areas and datasets. A dataset can contain one or more memory areas. This splitting into logical areas and datasets has to do with the wanted ability to keep different levels of security and access for different users. The users can be both human operators and different SW applications.

Hence, we can achieve a product line where all products use the same executable file and product specific behavior is controlled by parameters in persistent memory. One benefit is to have fewer executables to release and administrate. Relative to the method of developing different executables, the testing effort is unaffected. Equal numbers of products must still be tested. Another benefit is that in order to change an existing product, only the parameters must be loaded into the target computer.

Drawbacks include; unnecessarily large executable, dead code could possibly affect safety issues.

2.6.2 Variables

A variation on the idea of parameters is to let the application store certain variables in persistent memory. These variables also control program flow, but the idea is that the application itself calculates values according to some individual environment condition. This can be used for adaptive control solutions where there is a variation in response between different hardware. For instance, different response times in individual computer controlled hydraulic valves. Several challenges exist where adaptive control can be or is being used.

2.6.3 Optional nodes, sensors, actuators, and resources

Parameterization via datasets provides a convenient way to variate product functionality. The technique is especially useful when dealing with pure software options, like variable algorithms. Some functions are also dependent on additional hardware, like sensors, actuators, I/O, and significant amounts of computational resources. This includes functions like lift-weighing and communication systems. Parameterization is often considered a too expensive method for these functions. Fitting hardware and extra computational resources that is not used in the product adds to the product cost. Instead, the function is achieved with an optional kit of physical devices. This can include hardware, mechanics, sensors, actuators, and/or software. The method for providing the option is different for each type of function. Sometimes only an extra sensor is needed, given that there is enough I/O. (Note that I/O is also a resource that increases product cost, but it is often feasible to have a spare capacity in the delivered product.) Sometimes the optional function can be achieved with a separate node including all the hardware needed for the function. One example is the optional tachograph that is a node that can be connected to the network and print work shift reports on paper. One way to achieve a lift-weighing function is with a separate node with display and also the necessary sensors and wiring. Another way would be to use an existing display and use processor, memory, and I/O spare capacity to execute an optional task.

Most often in VCE, the requirement on low product cost is high. Since the product volumes are relatively high, the product cost becomes an important parameter. Methods for estimating the real cost for the product life cycle are often not trivial.

Design decisions on whether to include resources in the product or not need to take into account:

- After market cost - fitting, warehouse cost, knowledge and education, after market tool support.
- Assembly cost - Physical product cost more if it includes more hardware, assembly takes more time.

Configuration management issues cost money for all optional functions. This is true whether the function is included or optionally fitted. In practice it costs money to maintain and test a large number of configurations.

Using an optional node that can be connected to the network bus has one other advantage. This provides a feasible interface, if the node is to be developed by a subsystem supplier.

2.6.4 Logistic overview

In the case of Truck products within Volvo, the central system and the number of configurations are larger than for construction equipment. The salesman has a number of configurations to present to the customer. (Standard configurations may include; logging truck, garbage truck, refrigerator truck etc...) The customer can then choose additional features. This includes both parameter settings and hardware components like climate control for instance. The central system includes functionality such as automatic ordering of components when the customer has ordered the truck. Later in the assembly line, the central system produces a worksheet for this particular configuration.

The after market tools support parameter changes and the possibility to download new software.

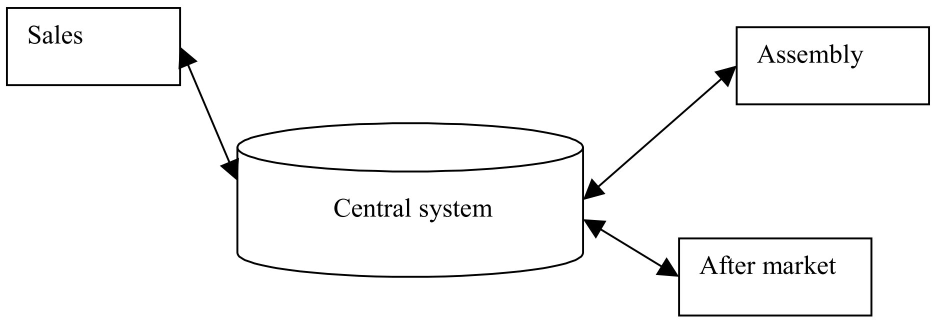

Figure 10. Central system

2.7 Process & Methods Today

VCE is developing on-vehicle software control systems for product lines of construction equipment. The content of electronics/software is increasing in newer vehicle models. As the system complexity grows, the need for structured high-level design methods becomes apparent.

VCE uses a design process that focuses on high-level design and temporal attributes of the system. The process relies on decomposing the system into tasks and specifying the temporal attributes for each task. A configuration tool allows for use of language constructs to specify a high level design such as temporal constraints, communication and synchronization. This clearly separates the design of timing characteristics from the design and implementation of logical functionality. By estimating each task's execution time in an early phase, the process accommodates early shedulability tests.

The process was first presented in [2], but an overview with some additions is presented here.

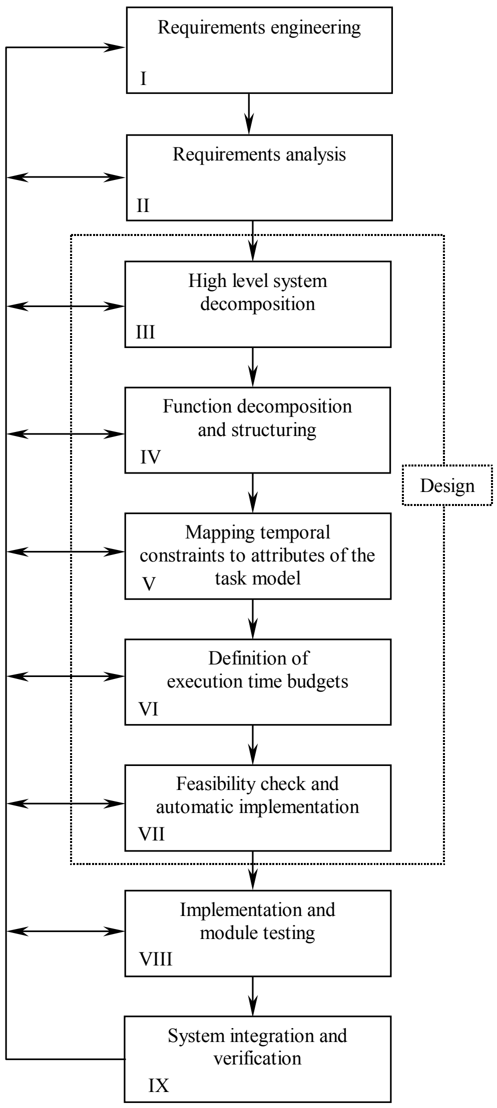
Requirements engineering
I
Requirements analysis
II
High level system decomposition
III
Function decomposition and structuring
IV
Design
Mapping temporal constraints to attributes of the task model
V
Definition of execution time budgets
VI
Feasibility check and automatic implementation
VII
Implementation and module testing
VIII
System integration and verification
IX

Figure 11. Development process

I. **Requirements engineering**

The product company compiles the customer requests and judge importance of each wanted function. The product company may also makes initial guesses on how the wanted function could be implemented. Some functions may obviously be part of the electronic system while others may require combinations of electronics and mechanics.

II. **Requirements analysis**.

Requirement analysis is done in cooperation between the product company and the electronic development department. In this stage the high level functions of the application are identified. It is also important here to determine temporal constraints for these functions. This step is not completely done until a detailed requirement specification exists

III. **High-level system decomposition**.

In this stage the different operational modes of the application are identified together with valid transitions between them. As an example the control system can have different functionality depending on the status of the vehicle, such as driving, startup, failed, and reduced.

IV. **Function decomposition and structuring**.

The functions, for each mode, are decomposed into transactions. Transactions in turn are decomposed into tasks. Tasks are defined by their low-level functions together with the data flow between them.

Example 1: A multirate controller. The transaction consists of two sampling tasks, S1 and S2, a controller task C, and an actuator task A. The data-flow of the transaction can be seen in Figure 2.

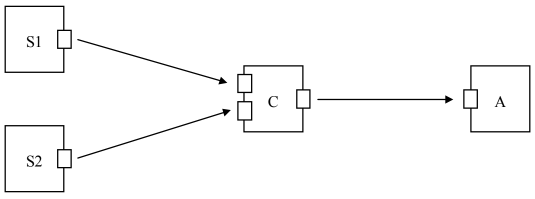

Figure 12. Data flow between tasks

Functionality having high responsiveness requirements or which occurs frequently but with short execution times cannot be implemented as a periodic task as the overhead would be too high. Such low-level functions are therefore implemented as interrupts.

V. **Mapping temporal constraints to attributes of the task model**.

In the previous stage high-level functions were decomposed into tasks and structured according to the interaction between them. In this step the high level temporal requirements are broken down into temporal attributes for tasks and the synchronization between them.

Synchronization can be defined by precedence relationships between tasks or by mutual exclusion of tasks sharing a common resource.

Example 1 (continued): The temporal attributes, derived by the control engineer for the multirate-transaction example are as follows (note that we don't know of the WCET of the tasks as yet, hence the question marks):

- Tasks S1 and S2 have period, release time, deadline, worst case execution time in µs = (1000, 0, 1000, ?).
- Task C = (5000, 4000, 5000, ?).
- Task A = (5000, 4000, 5000, ?).

To enforce order between the controller and actuator task, we specify a precedence relation between C and A, i.e., task C precedes task A.

VI. **Defining Execution Time Budget**.

WCET can be obtained by either measurement or by statically analyzing the code produced for each task. In the VCE approach, however, execution time budgets are estimated, being used later in step VIII as implementation requirements. The reason for this is that a feasibility test for the system, and a possible re-engineering, can be performed at an early stage, thus permitting early detection of design errors related to resource utilization, communication and synchronization.

Example 1 (continued): The question marks are replaced by estimated time budgets. Since this is a delicate issue, it requires highly skilled engineers with long experience.

VII. **Feasibility check and automatic implementation**.

The formally described design can be checked for temporal correctness, with a tool designated Configuration Compiler (CC), even if no actual (low-level) implementation has been produced. The CC maps the design description to a resource structure. The CC is a pre-run-time scheduler that generates dispatch tables for running the tasks and the communication infrastructure for each mode. This constitutes an application skeleton for the running and communication of the tasks. In addition to the mapping of the model, CC also supports specification of architecture-specific attributes such as HW-performance, resolution of the run-time dispatcher, communication times, and the number of nested pre-emptions permitted. The implementation of the CC is based on a heuristic tree search strategy, similar to the one presented in [18]. The major difference is that this scheduler takes into account interrupts [3] and architecture-specific attributes. The current version of CC is adapted to the real-time operating system Rubus (both commercial products, see www.arcticus.se).

VIII. **Implementation and module testing**.

The tasks are simply implemented by traditional programming (coding). In addition to the traditional functional specification, the programmer also has the execution time budget as an implementation requirement, i.e., the programmer must implement the specified function without exceeding the budget. The module testing includes both verification of functional behavior and checks that time budgets are not exceeded. If a time budget cannot be met, a redesign is required. In a simple but common case, a redesign could mean only to change the time budget given that there is enough CPU capacity available.

IX. **System integration and verification**.

The integration phase is usually performed quickly and without problems since the actual integration was performed during the design phase. The major task is the integration testing.

3 Conclusion

In this report we have described the current electronic systems architecture in some of VCE's construction equipment products. The VCE vision is to implement methods, processes, and technology to accommodate cost reduction, coordination of development of many products, and improved system properties. The general idea is to apply a product line approach to cut costs and facilitate coordination. Improved methods for architecture design will target the wanted improvement in system properties. Both solutions are quite complex and involve areas such as organization, process, technology, knowledge transfer, business strategy, and many more.

VCE is also a global company and products are developed and produced throughout the world. This report describes mostly the aspects of electronic development in the Sweden branch of VCE. Before trying to present possible solutions to today's challenges, the architecture needs of all VCE branches must be investigated. This study will continue with looking into aspects of VCE electronic development in VCEK, VCE Compact equipment, VCE Graders.

However, this global business situation with a wide variety of products is even more complex. VCE is one company within Volvo Group. The companies within Volvo envision competition benefits from a company wide coordination and reuse strategy. The potential for strengthening business is very large in a company Volvo's size. There is already some coordination and reuse in the works. Today, the VCE architecture is based upon that of VTC and uses the same service concept, production tools, a number of technologies, and a number of methods. Volvo Bus, Volvo Penta, and Volvo Aero are also facing the same basic challenges of cost and coordination. This study will also seek to find ways to improve and coordinate architecture issues by investigating common solutions.

4 References

[1] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[2] C Norström, K Sandström, M Gustafsson, J Mäki-Turja, and N-E Bånkestad. *Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry.* In proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS01), Washington, US, April 2001. IEEE Computer Society.

[3] Sandström, K., Eriksson, C., and Fohler, G., *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*, In Proceedings of the fifth International Conference on Real-Time Computing Systems and Applications, pp158-165, October 1998. ISBN 0-8186-9209-X.

[4] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg, *Volcano a revolution in on-board communications*, Volvo Technology Report. 98-12-10.

[5] Brownsword, L., Clements, P., *A Case Study in Successful Product Line Development*, Technical Report CMU/SEI-96-TR-016, October 1996.

[6] Bredemeyer, D., *Motivating Software Architectures*, http://www.bredemeyer.com/

[7] Bass, Clements, Kazman, *Software architecture in practice*, Addison Wesley, 1997.

[8] Eriksson, C., Mäki-Turja, J., Post. K., Gustafsson, M., Gustafsson, J., Sandström, K., and Brorsson, E., *An Overview of RTT: A Design Framework for Real-Time Systems,* Journal of Parallel and Distributed Computing, August, 1996.

[9] Josefsson, M., *Programvarukomponenter i praktiken -att köpa tid och prestera mer*, report V040078, Sveriges Verkstadsindustrier, 1-1-1999.

[10] Isovic, D., Lindgren, M., Crnkovic, I., *System Development with Real-Time Components*, In Proc. of ECOOP2000 Workshop 22 - Pervasive Component-based systems, June 2000.

[11] Tran, V., Liu, D., *Component-based Systems Development: Challenges and Lessons Learned*, Software Technology and Engineering Practice, Proceedings., Eighth IEEE International Workshop on, 1997.

[12] Le Lann, G., *An analysis of the Ariane 5 flight 501 failure - a system engineering perspective*, In Proceedings of 4th international conference of engineering of computer based systems, IEEE Computer Society, 1997.

[13] Thane, H., *Monitoring, Testing and Debugging of Distributed Real-Time Systems,* Doctoral Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden 2000.

[14] Clements, P., Northrop, L., *Software Product Lines*, Addison-Wesley, August 2001. ISBN 0-201-70332-7.

[15] Fenton, N. E., Pfleeger, S. L., *Software Metrics: A rigorous & practical approach*, International Thomson Computer Press, ISBN 0-534-95600-9, 1996

[16] Douglas, B.P., *Real-Time UML - Developing efficient objects for embedded systems*, Addison Wesley Longman, Inc, 1998

[17] Wall, A., *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*, IT Licentiate theses, 2000-004, MRTC Report 00/21.

[18] K. Ramamritham. *Allocation and Scheduling of Complex Periodic Tasks*. In 10th Int. Conf. on Distributed Computing Systems, pages 108-115, 1990.