Version: 2005-09-27

MRTC Report

Component-Based Development of Safety-Critical Vehicular Systems

State of the Art, State of the Practice and Research Challenges

Authors:

Ivica Crnkovic, Mälardalen University
DeJiu Chen, Royal Institute of Technology
Jad El-Khoury, Royal Institute of Technology
Johan Fredriksson, Mälardalen University
Hans Hansson, Mälardalen University
Jörgen Hansson, Linköping University
Joel, Huselius, Mälardalen University
Ola Larses, Royal Institute of Technology
Joakim Fröberg, Mälardalen University
Mikael Nolin, Mälardalen University
Thomas Nolte, Mälardalen University
Christer Norström, Mälardalen University
Kristian Sandström, Mälardalen University
Aleksandra Tešanović, Linköping University
Martin Törngren, Royal Institute of Technology
Henrik Thane, Mälardalen University
Simin Nadjm-Tehrani, Linköping University
Mikael Åkerholm, Mälardalen University

Editor:

Ivica Crnkovic

Chapter 2

Title: Vehicular Embedded Control Systems

Authors: Martin Törngren, Ola Larses, Kristian Sandström, Johan Fredriksson, Joakim Fröberg, Christer Norström, Mikael Åkerholm

This chapter focuses on the state of practice of embedded control systems by describing existing software and hardware architectures. Compared to chapter one, this chapter provides more detail on the characteristics and constraints facing vehicular embedded systems. As a whole, the chapter lays a foundation for better understanding how the constraints and requirements of different vehicular systems have affected their architectures, thus forming a basis for the coming chapters. The chapter introduces basic concepts, terminology, fundamental structures of the control systems, and the context of the vehicular embedded systems; the mechanical environment, sensors and actuators. In later sections, current architectures of cars, heavy vehicles, trains and industrial robots are surveyed. The modeling and programming of these systems is further discussed in the following chapters. The chapter ends by summarizing current trends and challenges.

2.1 Fundamentals of embedded control systems

Since this chapter deals with embedded control systems it is appropriate to start out by defining and characterizing these two terms.

An Embedded Computer Systems: A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system. (IEEE, 1992).

The area of embedded systems is very broad and spans a very large number of applications, from TV-sets, over mobile phones and toys, to production machinery, and system types, from single processor to distributed systems. Since all of these applications have very widely varying requirements it does not, in fact, make much sense to talk about embedded systems in general. However, turning to vehicular control systems does provide a useful delimitation. This leads us to the second definition.

A control system: A system in which a desired effect is achieved by operating on various inputs to the system until the output, which is a measure of the desired effect, falls within an acceptable range of values. (IEEE 1992

When implementing a control system today, a designer has a variety of technologies to choose from. For low-end products, analogue technology can be used. Today, however, digital technology is normally used and a typical implementation is based on a microcontroller; a highly integrated electronics device that includes a microprocessor, communication facilities, inputs (digital and analogue), and outputs (digital and analogue).

Embedded control system are in general characterized by the following (see also Fig. 1.4):

- *Feedback control:* This type of functionality matches the definition of a control system as given above. In vehicular systems, it is often the motion of the vehicle itself, or its various parts (e.g. the throttle) that is being controlled. Depending on if the set-point to the controller is fixed, or varies frequently, the controller is called a regulator or a servo. Feedback has the important property that a certain robustness to

disturbances acting on the controlled process can be achieved. However, feedback controllers are sensitive to the timing induced in the feedback loop as a consequence of executing the controller. Discrete time control theory assumes equidistant sampling (i.e. none or negligible jitter) and negligible or constant feedback delays (this delay is counted between the time instants of sampling, and corresponding actuation actions), see e.g. Wittenmark et al. (1995), Åström and Wittenmark (1990).

- *Feedforward control*: Given knowledge of the controlled process, feedforward or open-loop control can be applied. In this case, no feedback is used which is like walking in a room without using any sensory feedback. This is only possibly with an accurate model of the controlled process. With such an accurate model, the required control signals to achieve a desired motion can be computed and applied to the actuators. In applications with well defined environments, such as in industrial robotics, feedforward control is used to a large extent. As no model is perfect, and since the controlled process may have characteristics that vary over-time, most control systems combine feedforward and feedback control.

- *Estimation:* Estimation can be used for several purposes including fusing information from several sensors to obtain better predictions of the system state, using models to predict certain states that can not be directly measured, or to detect a possible faulty behaviour of for example a braking system.

- *Mode logic (or arbitration):* Any non trivial control system will have several modes of operation, which may refer to initialisation, different operating conditions, and to graceful degradation. Consider for example a flight control system which has several air surfaces which are the actuators to be used for controlling the orientation (yaw, pitch and roll) of the aircraft. Given that one or more actuators begin to mal-function, the controller should hopefully detect this (see diagnostics below), and will then need to switch to another flight control law; this is necessary since there is now another aircraft – in terms of dynamic behaviour – to be controlled!

- *Diagnostics and logging:* For the purposes of on-line error detection, and for post-mortem analysis (e.g. debugging) it is necessary to store the state of the control system. This state normally corresponds to the state variables used in the control design which typically includes measured variables (e.g. speed), estimated variables, and the control signal(s). Diagnostics usually refers to on-line analysis of the system state, for example to detect actuator malfunction or for that matter to communicate an upcoming service need to the driver, see e.g. Iserman (2002).

- *Tight coupling to the controlled process:* The tight coupling between the control system and the controlled process is manifested in several ways. Apart from aspects related to physical integration, sensing and actuation, the control system is also fundamentally related to the controlled process. Often, the control algorithms, synthesized from a validated model of the controlled system (model based control systems development). In other cases, the controller parameters are tuned with respect to the controlled process (e.g. the case for PID-controllers). Since the controlled process will change over time due to for example, it is usually necessary to either include adaption in the controller (such that controller parameters can be changed) or to make the controller robust enough to cope with changes in the controlled process. In addition, the speed (or bandwidth) of the closed loop system will provide requirements on the timing of the controller, including the sampling periods that successfully can be used, and feedback delays and jitter that can be allowed. These latter properties can

also be taken into account in the control design, however, providing a sort of a contract between the controller and its implementation.

- *Hierarchy and coordinating control:* Hierarchy is common in complex systems, and also in control systems. Humans have several low level controllers. One of them is located in the spine, and handles basic motion control such as walking. The brain contains several higher order controllers. Taking one example from vehicular systems, consider control of a car, where braking, engine and transmission controllers typically provide low (actuator) level control. The coordination of these is provided by higher level functions such as cruise and spin control. Along with the hierarchies normally follows an increased level of abstraction and slower time constants (frequencies).
- *Autonomy and human control:* Many control systems have an operator "in the loop". This is typical for vehicular systems, and the situation arises where conflicts can occur – who is deciding the motion of the vehicle at any given point in time? Careful analysis is required and special care has to be given to the human/machine interface.
- *Time triggering and event triggering*: A typical control system includes activities which are both time- and event-triggered. In many cases, time-triggering follows naturally from the development of discrete time (sampled data) functions. Sometimes the controlled process is not discrete in time. This is the case for inherently sampled systems, one example being control of injection in a combustion engine; the point in time of injection depends on the speed and angular position of the engine parts, see e.g. Åström and Wittenmark (1990). Event-triggered functions thus include those who are inherently sampled and other functions who are not dictated or preferred to be implemented as periodic activities.
- *Safety related functionality:* Moving systems can be dangerous, vehicular control systems are therefore normally safety related. Not only is the control system required to operate reliably; the design of the system and its context must be carefully analysed to consider what might go wrong, and what the system should do in such cases. In safety critical control systems it is common to use diagnostics and analytical (model-based) redundancy both for detecting, and to some extent for handling errors. The robustness of control systems can also be utilized to handle certain failures.

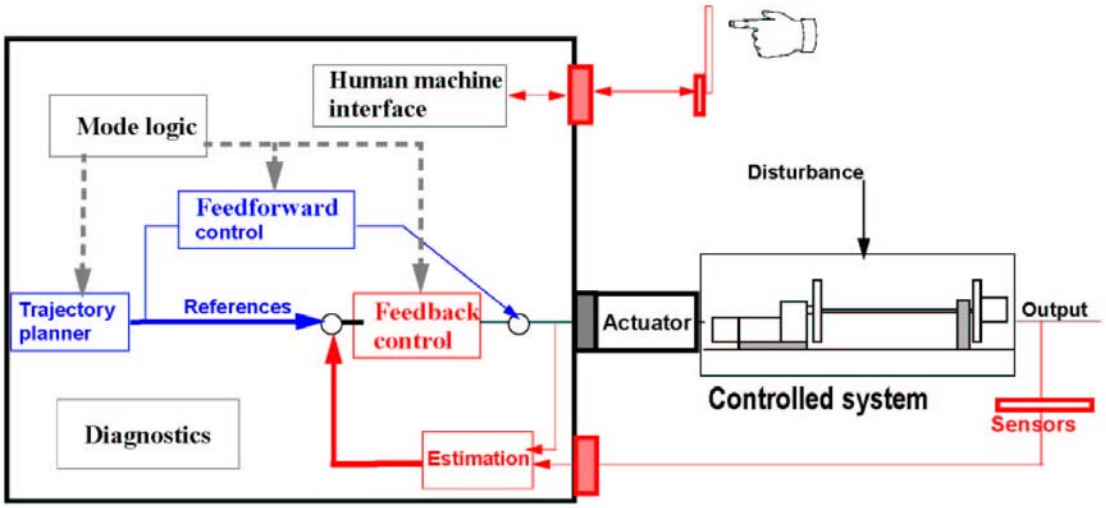


Figure 2.1 *Basic elements of a control system*

As introduced in chapter one, a vehicle today contains several subsystems which usually are dedicated to different applications. One concrete example from the automotive industry is given in Figure 2.2. 1-X. Today, many of the exemplified functions have already been realized or are being developed (the latter is particularly valid for the safety, driver assistance and electrical power supply domains).

It follows from the above discussion that embedded control systems are populated with different functionality, and that this functionality has varying characteristics and requirements. Only a subset of the functions may for example be highly safety critical, the types of timing requirements vary, as will the computational models used for functions (e.g. discrete time equations vs. state machines). While the portions dedicated to motion control usually are associated with strict safety and timing requirements, these portions are typically relatively small compared to all other functionality.

Powertrain and Chassis	Safety	Driver Assistance
Gasoline direct injection	Vehicle dynamic control	Adaptive cruise control
Active suspension control	Occupant and rollover sensing	Stop & go control
On board diagnosis	Electro-mechanical/hydraulic brake	Parking assistance, Parking brake by wire
Combustion control	Collision warning and avoidance	Blind spot detection
Electromechanical valve lift control	Steer by wire, Brake by wire	Video-based traffic sign, lane and obstacle recognition

Body and comfort	Infotainment and Telematics	Electrical Power supply
Keyless Entry / Keyless Go	Dynamic navigation	14/42V power supply
Lock by wire	Speech recognition	DC/DC converter
Free programmable LC-display	Multimedia entertainment	Starter-Alternator
Adaptive light distribution	Internet access, E-mail	Battery management
Security systems	Telediagnosis	Alternator management
Biometrical systems	Roadside assistance	Electrical load management

Figure 2.2 *Trends on new functions as seen in the automotive industry in 1999. (Source: Vehicle Body Electronics, Robert Bosch GmbH, Stuttgart, VDI Berichte 1999).*

2.2 The context of vehicular embedded systems and distributed systems

The development of embedded control systems is strongly linked to the transformation of the traditional control systems to distributed control systems (today more fashionably called *by-wire* control) and to the embedding of the control systems into a quite harsh environment. In the following two subsections, the close relation to the vehicle (physical embedding) and the distributed system/functionality nature of embedded control systems are further elaborated.

2.2.1 The mechanical environment: energy, sensors, actuators and information

In early implementations of engine controllers, the electronic control unit was usually mounted far from the engine. However, the resulting cabling and required connectors came to cause a number of problems. The need to transfer information from sensors using analogue signals a fairly long distance is not appropriate and requires costly cabling to handle the

sensitivity of the signals. In addition, there is the need to reduce the number of connections. As a consequence, engine control units are today mounted directly on the engine, and in fact form a highly integrated part of the engine, see Figure 2.3 for an illustration of this.

The Scania diesel engine controller is designed to be mounted on the engine without cooling or damping. Special emphasis has been placed on the development of the connector to achieve robustness. The number of external connectors has been minimized; the only signal cabling to and from the engine goes through the ECU connectors. The engine ECU apart from engine control and CAN communication, carries out a number of additional functions such as controlling the fan, alternator, engine brake, turbo, and the EGR valve.

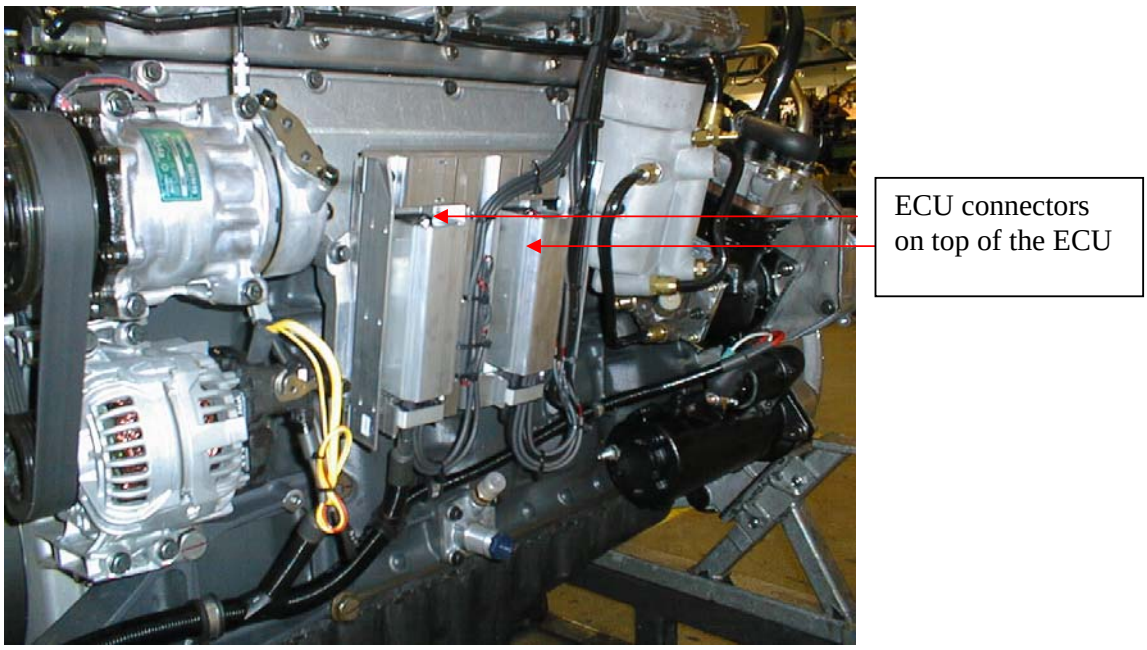


Figure 2.3 *Mechanical embedding exemplified by the Scania diesel engine controller (figure source from Scania)*

Although this type of physical embedding places high requirements on the physical design of the control unit to consider vibrations, varying temperature and humidity, it also provides a number of other benefits. The engine with its integrated ECU forms an intelligent component thus promotes modularisation. The ECU can be used for a variety of purposes such as providing component identity, individual component calibration, local control, and a means for integration with other control units.

The interfaces of such an intelligent actuator thus includes

- Energy supply (diesel in the example)
- Mechanical interfaces
- Information interface (CAN network)
- Energy supply for the ECU

To further illustrate the context of an embedded control system, consider Figure 2.4.

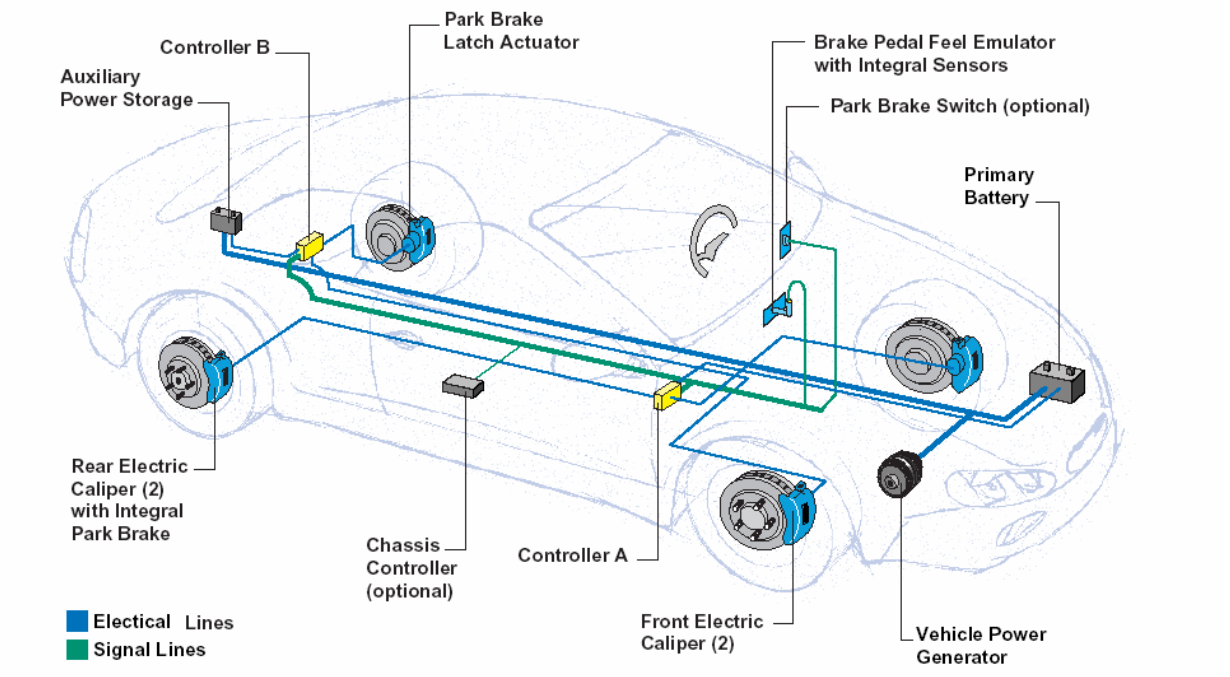


Figure 2.4 *Mechanical embedding of a braking control system illustrating power supplies, power cabling, controllers, communication between controllers, and some sensor and actuator elements.*

The following hardware components appear in Figure 2.4:

- Energy source(s) including a battery and a power generator, and there could of course be other sources or redundant specific sources.
- Energy distribution cabling – referred to as electrical lines in Fig. 2.4, in terms of cabling for supplying various actuators and other energy users with energy.
- Actuators, for transforming energy into motion
- Sensors, to provide information to the control system and/or humans about the current status of the vehicle.
- Information distribution – referred to as signal lines in Fig. 2.4, in a modern vehicle carried out by embedded networks such as the controller area network (CAN).

Note that since there are several actuators it may not be possible to run all actuators at once. Power management is essential in modern vehicles. For safety related functions, high requirements are placed on the reliability of all involved components. There may be inherent redundancy in the vehicle (for example true for the brakes but not for normal steering of a car).

2.2.2 Distributed control systems; towards by wire control

In general, there has over the last decades been a strong trend to connect stand-alone controllers by networks, forming distributed systems. The main driver for this has been cost reduction, since the use of networks makes reduction of the necessary cabling possible, or at least a possibility to bound the increase in length given the drastic increase in the number of control units. Another and closely related trend has been modularisation, where for example, an electronic control unit is physically integrated into an engine, forming a sort of mechatronic module. Combining the concepts of networks and mechatronic modules makes it

possible to reduce both the cabling and the number of connectors, the result of which is facilitated production and increased reliability.

A common definition of distributed systems is as follows:

Nodes part of a distributed system cooperate to accomplish a common goal.

This definition is of course quite general and there are many other definitions. A more fruitful approach is that of discussing how systems are distributed (or decentralized). A useful interpretation was introduced by Enslow who defined three dimensions of distribution according to which a system can be classified (Enslow, 1978). Clearly, there is a need for distributed hardware; but in addition, we may consider different distributions of data and of control (decisions) in the system.

In essence, the concept of distributed systems then becomes that of mapping.; given a description of the system functionality (for example as illustrated by Figure 2.1), and the hardware architecture (for example as illustrated by Figure 2.4), we need to decide how to map the functions to the hardware components. This concept, where functions can be implemented by different technologies is illustrated in Figure 2.5.

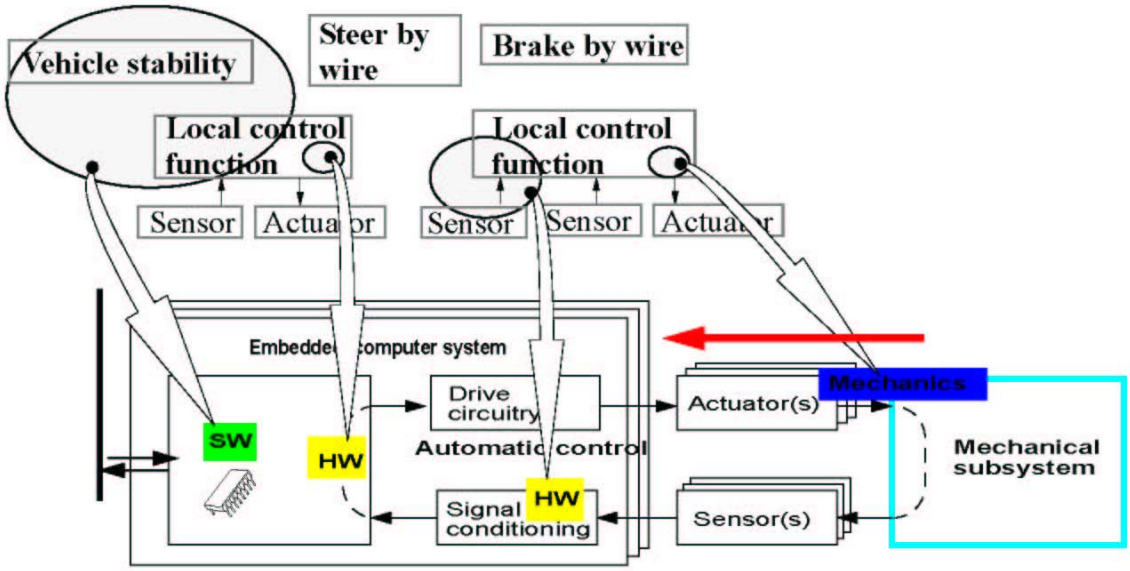


Figure 2.5 *Mapping of functionality to a hardware architecture including mechanical and electronic components, where the electronic components form a distributed computer system*

Each node of a distributed computer system is composed of hardware, supporting system software and application functions, as depicted in Figure 2.6. As described in section 2.1 we normally find hierarchies in the application. This is also true for the supporting software.

If we look in more detail on the mapping of the functions that are implemented in the distributed computer system, we find that there are in principle a wide variety of possibilities, ranging from a master-slave (often called distributed I/O) organization at the application, ranging to more fully distributed systems. Along with this mapping we also need to define or synthesize (see Törngren, 1995)

- the software architecture, including the partitioning of functions to tasks
- the execution strategy, referring to the choice of triggering, synchronization, and scheduling policies

- error detection and error handling strategies.

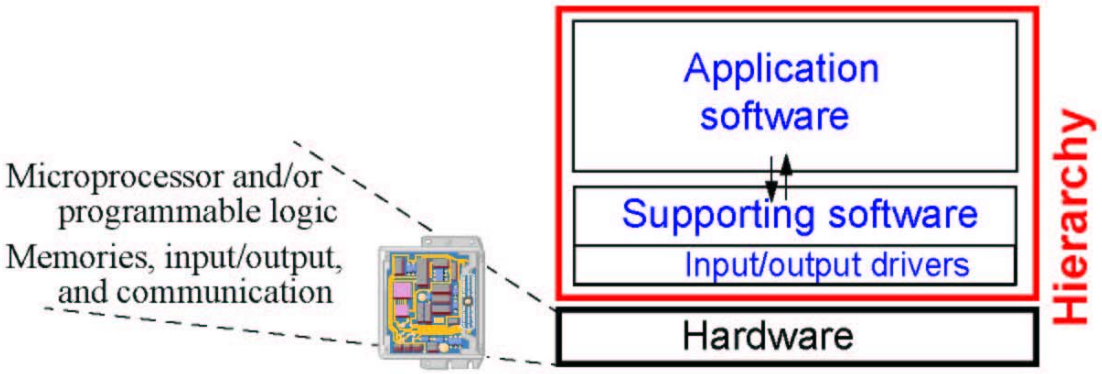


Figure 2.6 *Structure of a node, part of the embedded distributed computer system. The supporting software and the hardware are sometimes referred to as a platform. Note that Enslows classification applies equally well to all functions of the system, including the application and the system level*

Introducing networks also means possibilities to more efficiently carry out diagnostics and to coordinate the operation of the separate systems, thus making new functionality possible. One example of this is the functionality provided by the Electronic Stability Program (ESP). ESP combines control of braking and throttle to implement yaw control to handle situations when a car is beginning to rotate (or spin) out of control for the driver. The technology uses a set of sensors connected to the braking and throttle system. Whenever the danger of a spin is detected, the ESP reacts by selectively applying braking force to the front and rear wheels and reducing or increasing engine torque. (Mercedes, 2003).

2.2.3 By wire control

The term by-wire control is strongly related to distributed control systems. The terminology of by-wire systems has developed organically as new areas of coverage have been identified. We here suggest a structured approach to the terminology, see Larses (2003) for more on this topic. The by-wire terminology can principally be applied on any control system and in any area of application.

First a distinction between by-wire control systems and full by-wire systems need to be made. In a by-wire control system (BWC) some stage of the information transfer is performed through digital communication and digital decision making, not only analogue signals and filtering is used. It is generally assumed that the information or control signals are digitally communicated by an embedded control system. The embedded control system comprises of a set of electronic control units (ECU) that are interconnected in a local network. In a full by-wire system (FBW) all the information is digital and also the power is transferred electrically by the use of electrical motors and electromechanical actuators. Using the terminology is it possible to state that a full by-wire system is a by-wire control system with electromechanical actuation.

A widely and rather informally used term is x-by-wire. X-by-wire is a general term that contains any application with by-wire functionality where ‘x’ is a referral to the actual application. The terminology refers to the application scope of the system and can be arranged hierarchically as shown in Figure 2.7. For example, the term fly-by-wire corresponds to applications in avionics and drive-by-wire covers automotive applications. Within these

categories further refinement is possible by naming specific functions like brake-by-wire and steer-by-wire.

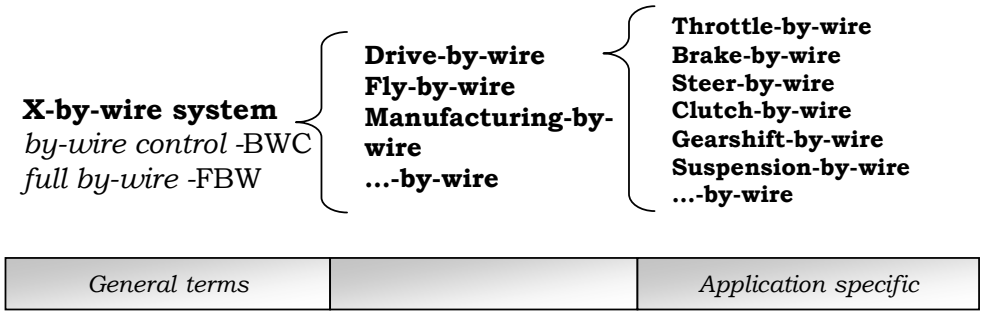


***Figure 2.7** - By-wire terminology*

All the different by-wire classes can obviously be of the type control system or full system, thus rendering the expression variants brake-by-wire control system and full brake-by-wire system. In a brake-by-wire control system is the information transfer from the brake pedal to the breaking actuator handled electronically but hydraulic or pneumatic actuators, probably with electronically controlled valves, perform the actuation. If the actuators are replaced with electrical motors a full brake-by-wire system is created.

By-wire control has been implemented for a long time and also applied to critical applications, most notably in terms of commercial fly-by-wire applications, which has been in use in the Airbus series aircraft since 1983 in the A310 model (Augustine, 2000). The Airbus 320 was the first that depends entirely on by-wire control and it was certified in 1988 (Briere & Traverse, 1993).

The introduction of by-wire systems have many benefits but also drawbacks. Cost and safety are two areas that have been pointed out as potential problem areas in the automotive industry (Feick et al, 2000). These, and similar, uncertainties have different impact in different by-wire application areas and give differences in the requirements on by-wire systems. That is, the requirements of manufacture-by-wire and fly-by-wire are significantly different due to inherent properties of the systems. These inherent properties can be described in three dimensions: The criticality or consequence of failures, the sensitivity to frequency of failure in operations and the cost sensitivity of an application, influences requirements as discussed in Chapter one.

Examples of rather early by-wire systems in the automotive industry include automated manual transmission (AMT). With AMT the mechanical connection to the transmission is eliminated and the gears can be chosen by pushing buttons or allowing a computer to run a gear selection program. AMT systems can be added onto existing manual systems and today exists in several cars like BMW, Mercedes-Benz, Alfa Romeo 156 and VW Lupo. The technology was first developed for motor sports to relieve the driver from using the clutch. (Wagner, 2003). AMT was introduced under the name Tiptronic by Porsche, ZF and Bosch on the Porsche 911 in 1989. Today several solutions are available like iShift from Volvo Trucks and Opticruise from Scania.

Several issues however hamper the introduction of steer by wire in vehicles in general including

- legislation
- user expectations

- cost-efficient embedded control. As discussed in Chapter one, the technological concepts used for by-wire systems in airplanes are not cost-efficient for the automotive industry.

In the automotive industry, full steer-by-wire is not available yet for the above mentioned reasons. However, several prototypes have been developed and intermediate solutions are being developed that typically include some form of mechanical back-up system. Steer-by-wire is commercially available in the form of Quadra-steer, four-wheel steering, from Delphi. This concept uses full steer by-wire for the steering of the rear wheels while using conventional steering on the front wheels. With four-wheel steering is possible to improve the turning radius of the vehicle when the front and rear wheel turn in opposite directions simultaneously. This is the case at low speeds, at high speeds the wheels steer in the same direction creating a sideways displacement of the vehicle without turning it, this functionality is intended for high speed highways improving lane shifting capabilities of the vehicle. (Amberkar et al, 2001).

2.3 Current architectures and standards

In this section we will present architectures, software technologies, and standards used today in vehicle electronic systems. The standards covered in this section are in several cases common within a domain whereas the architectures are specific for the companies that are represented.

2.3.1 Cars

Cars are typically manufactured in volumes in the order of millions per year. To achieve these volumes, and still offer the customer a wide range of choices, the products are built on platforms that contain common technology that has the flexibility to adapt to different kinds of cars. As an example, the Volvo XC90, which appeared in 2002, is based on the same platform as four previous Volvos launched since 1998.

The component technology is to a large extent provided by external suppliers, who work with many different car companies (or OEMs, original equipment manufacturers), providing similar parts. The role of the OEM is thus to provide specifications for the suppliers, so that the component will fit a particular car, and to integrate the components into a product. Traditionally, suppliers have developed physical parts, but in modern cars they also provide software. As the computational power of the electronic control units (ECUs) increase, it will be more common to include software from several suppliers in the same nodes, which increases the complexity of integration.

System architecture

An example of a contemporary car electronic architecture is that of the Volvo XC90 (see *Figure 2.8.*). The boxes in the figure represent ECUs and the lines represent communication buses. The intent of the figure is to show the complexity rather than the details. The maximum configuration contains about 40 ECUs. They are connected mainly by two CAN networks, one for powertrain and one for body functionality. From some of the nodes, LIN sub networks are used to connect slave nodes into a subsystem. The other main structure is a MOST (MOST Specifcation Framework, 1999) 0ring, connecting the infotainment nodes together, with a gateway to the CAN network for limited data exchange. Through this separation, the critical powertrain functions on the CAN network are protected from possible disturbances from the infotainment system. For more information about CAN and LIN see section 7.5.2. The diagnostics access to the entire car is via a single connection to one ECU. The figure shows approximately how the ECUs are placed in various locations in the car. The

partitioning of functionality is decided by the location of the sensors and actuators used, but also by the combinations of optional variants that are possible. If a car is sold with only a subset of the full functionality, the amount of physical hardware installed should be limited to the minimum necessary.

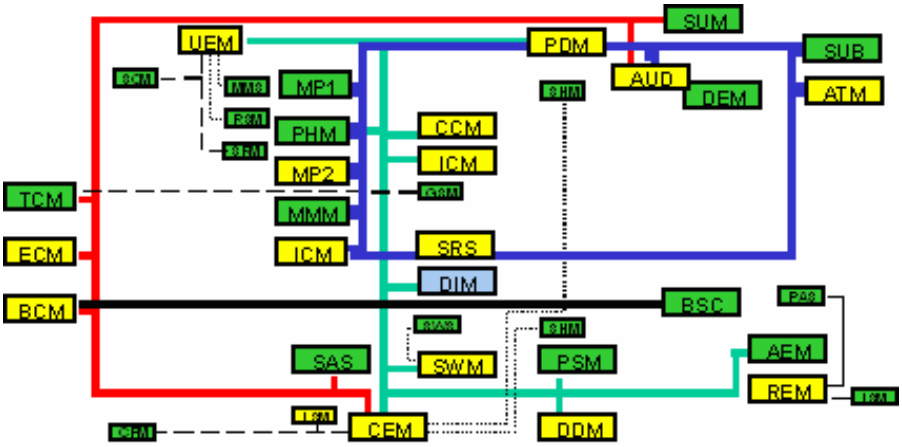


Figure 2.8. *The electronic architecture of Volvo XC90.*

Node architecture

The current development trends in automotive software call for increasing standardization of the software structure in the nodes. In particular, the use of code generation requires a clear interface between the support software and the application, and the need to integrate software from different suppliers in the same node also calls for a well-defined structure. The node architecture (see Figure 2.9) includes several important components:

Diagnostic kernels provide an implementation of the diagnostic services that each node must implement to act as a client towards the off-board diagnostic tool. It relies on the communication software to access the networks and on the operating system to schedule diagnostic activities so that it does not interfere with the application functionality.

Network communication software provides a layer between the hardware and the application software, so that communication can be described at a high level of abstraction in the application, regardless of the low-level mechanisms employed to send data between the nodes. *Volcano* (Casparsson et. Al, 1998) 0is a communication concept used throughout the Volvo Car Corporation for managing network traffic. Currently Volcano supports the CAN and LIN busses. The basic concept in Volcano for communication between software components is signals, where a signal typically represents some engineering data. Through the Volcano API the underlying network technology is hidden from the application engineer. Moreover the engineer is not concerned with issues regarding assignment of signals to network frames. Instead this is done automatically ensuring signal timing requirements.

The Volcano concept also addresses vehicle manufacturer controlled integration of components developed by suppliers. This is done through the use of the Volcano API and by separate specification of the signals used by a component and the network configuration. The network configuration is provided by the integrator and specifies how signals are to be transferred over the network.

Real-Time Operating Systems (RTOS) provide services for task scheduling and synchronization. Traditional real-time operating systems are usually too resource consuming to be suitable for automotive applications, and do not provide the predictable timing that is needed. Therefore the new standard OSEK has been developed (OSEK/VDX OS 2.2, 2003). There are several suppliers of OSEK compliant operating systems.

- **I/O**

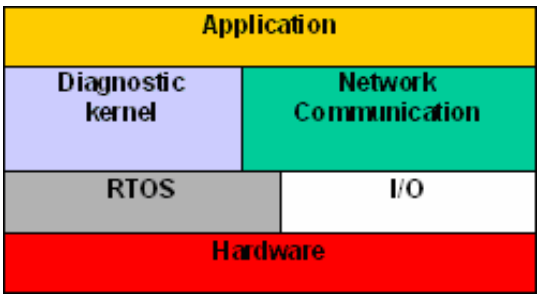


Figure 2.9 *The node architecture.*

All these components interact with each other and with the application, and must therefore have standardized interfaces, and at the same time provide the required flexibility. To conserve hardware resources, the components are configurable to only include the parts that are really necessary in each particular instantiation.

For future system development, an important aspect is to create a more flexible software partitioning. The main use for this is probably not to find the optimal partitioning for each car on a given platform, since that would create too much work on the verification side, but to allow parts of the software to be reused from one platform to the next. This puts even higher demands on the node architecture, since the application must be totally independent from the hardware, through a standardized interface that is stable over time. Therefore, further standardization work is needed, in particular for sensor and actuator interfaces.

2.3.2 Construction Equipment

Volvo Construction Equipment, VCE, develops and manufactures a wide variety of construction equipment vehicles, such as articulated haulers, excavators, graders, backhoe loaders, and wheel loaders.

Compared with passenger cars, construction equipment vehicles are equipped with less complex electronic systems and networks. Also, the focus in product development is somewhat different. The products are to be used in construction sites, and the most important aspect of the vehicle is to provide a reliable machine to increase production.

In-vehicle electronic systems and networks are an important part of the construction equipment product and are crucial to provide end-user functionality, such as automatic gearbox, as well as providing diagnostic and service functions. Using a distributed electronic system also accommodates functions that reduce the cost of the vehicle e.g. reuse of sensors and displays, and adaptive solutions to accommodate cheaper mechanical parts.

System architecture

The focus of VCE's electronic architecture effort is mainly concerned with assuring system properties including scalability to support product variation, reusability and partitioning of SW components to lower development cost, as well as safety and reliability. Moreover it is important to have an architecture that supports working with platforms.

VCE uses no externally developed vendor control units. This gives the possibility to use the same software component model, operating system, and reusing software components. By doing this, the partitioning of functionality is likely easier than in the case of a system with many differently developed control units. Easy partitioning gives the possibility to scale the system with respect to hardware and optimize hardware content in a specific product. For instance, a low-end product with fewer features requires less hardware resources, and can be

realized by placing software components on other nodes and thereby reduce the number of control units. This means that the number of control units can be chosen according to resource needs for a given configuration of a product. Thus, the architecture is reused, but the numbers of ECUs differ between products (see Figure 2.10).

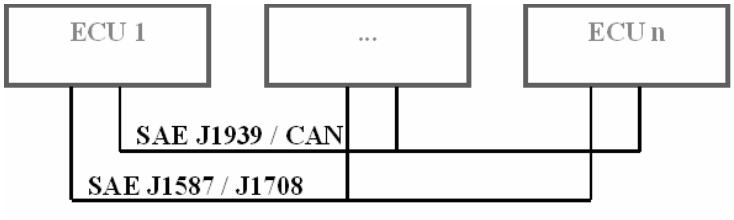


Figure 2.10 *VCE Network Architecture*

In the domain of heavy vehicles, the *SAE J1939* standard (SAE Standard, SAE J1939) 0for communication between components has been created by a number of vehicle manufacturers and sub suppliers. SAE J1939 uses CAN for communication and defines a transport service in the network layer. In the application layer the protocol defines data (signals e.g. vehicle speed) and the packaging of signals in frames. Moreover, J1939 also define the interaction between components, e.g. the interaction between engine and transmission during gear shifting. To provide for vehicle manufacturer specific functionality the protocol also allows some proprietary messages.

Node architecture

As depicted in Figure 2.11, each ECU consists of hardware and two different software layers (Nyström et.al, 2002)0. The top layer is an application layer with specific functionality for different ECU's and products. The middle layers are the interfaces between the application and the hardware. These layers are the communication layer, which handles communication between different nodes, the I/O layer that handles the in and output on the affected node and the real-time operating system Rubus, which handles task scheduling and synchronization.

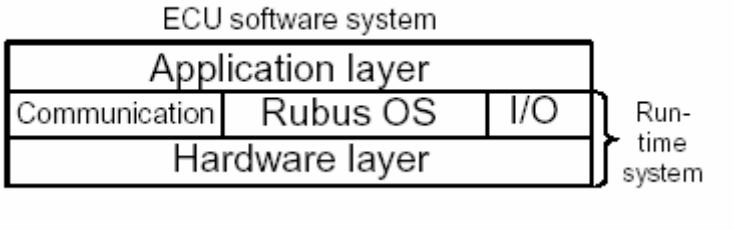


Figure 2.11 *The Node (ECU) Architecture*

A view of the Rubus RTOS used by Volvo Construction is shown in Figure 2.12. Rubus is a real-time operating system with support for both static scheduling and pre-emptive fixed priority scheduling. The static and pre-emptive fixed priority schedulers are referred to as the *red* and *blue* parts of Rubus respectively. The red part only handles hard, static real-time, while the blue part only handles soft real-time. Finally, there is also a green part of Rubus, which is an interrupt handler kernel. It has the highest priority, and distributes and routes the interrupts.

The red part of Rubus always has higher priority than the blue part. Therefore, the red part is used for time critical operations (firm and hard real-time), since it is easier to verify timing properties of the red part. The blue part of Rubus is usually used for more dynamic properties, and soft real-time. The green part always handles all interrupts, and has the highest priority in the system.

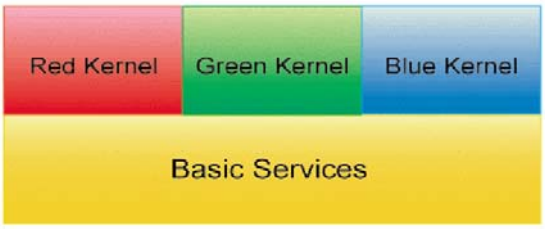


***Figure 2.12** Rubus RTOS Architecture*

Component model
Volvo uses the Rubus component model. A component in that model consists of one or more tasks, which are the run-time entities executed by the operating system. A task in turn is defined by a function pointer, data ports, and some attributes building the current configuration of the task. We can also identify a composite, which is a logical composition of one or more components. Components and composites are technically the same, but an encapsulation of components is logically separated from an encapsulation of tasks by different notions. For a more detailed description of the Rubus component model see section 4.4.3.

2.3.3 Trains

We will have a closer look into systems manufactured by Bombardier Transportation, which is a train manufacturer, with a wide range of products. Some samples from their product line are passenger rail vehicles, total transit systems, locomotives, freight cars, propulsion and controls, and signaling equipment.

System architecture

Bombardier designates the distributed control system Mitrac, and the heart of the system is the *Train Communication Network* (TCN) (Kirrmann et.al, 2001) that defines standards for a data communication network, interconnecting devices both between and within rail vehicles. The TCN is adopted as a standard by both IEC (IEC std. 61375) and IEEE (IEEE std. 1473). The general architecture is shown in Figure 2.13 from (Kirrmann et.al, 2001). Each vehicle has an own vehicle bus connecting on board devices, such as sensors and actuators for doors, brakes and air condition system. Applications which exchange information cannot determine whether its peer resides on the same bus, train or anywhere else in the network. From the train bus view the internal organization of a node is not detectable, since each vehicle bus is treated as an own node on the train bus.

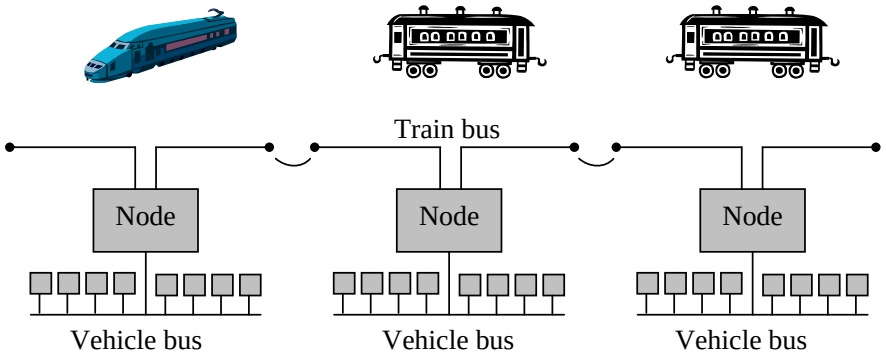


***Figure 2.13**, the train communication network*

The bus architecture is divided into a WTB interconnecting all vehicles, and in each vehicle a MVB, as according to the TCN standard. On each MVB there is between 5-10 processors. It is also common that the MVB actually consists of several buses, since the drive line

consisting of for instance anti spin and brake system often is separated from ordinary vehicle control functionality as door opening, air conditioning and entertainment system. For further details on the Train communication network see section 7.5.2.

Node architecture

A physical node in a control system either belongs to the drive line or to the ordinary vehicle control. A node belonging to the driveline is slightly more advanced, both types of nodes are based on the M360 board, but drive line nodes often has an additional Field Programmable Gate Array (FPGA) and more I/O units. The FPGA is used for tasks with high performance demands.

Each node in the system has approximately between 0.5 to 4 MB binary application code. It is roughly organized as in Figure 2.14. The three main parts are as shown in the figure basic software, run-time system, and application code. The basic software and run-time system is together denoted Common Software Structures (CSS), which is the basic platform, e.g., the operating system. The basic software in turn includes the three components Bootcode, SiMon and Standalone Download. These three components are downloaded to the node during the hardware manufacturing process and are consequently already in the node when it is delivered from the factory. Another part is the run-time system, which can be further decomposed:

- Run-time system starter, is initialization and start-up routines.
- Run-time services, implemented by the operating system.
- Underlying real-time kernel, which is the third party kernel VxWorks.

Eventually, the application code which utilizes the underlying layers are situated on top in the figure.

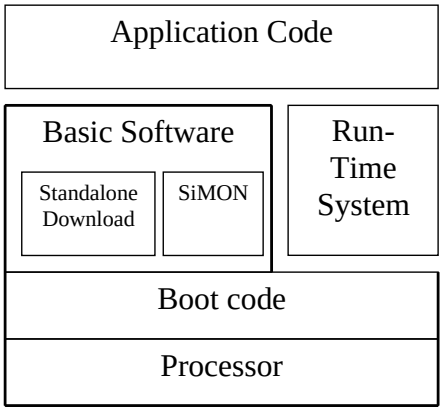


Figure 2.14. *A rough decomposition of a typical node*

Bombardier uses the programming languages defined in the IEC 61131-3 standard, for programming their devices. The primary language used is the graphical language called function blocks, although the standard defines three graphical and two textual programming languages.

The basics of the language are that an application is divided into a number of function blocks. Each function block has a set of in and out ports, and a hidden internal implementation. Each function block can contain both data and an implementation as a function. The intention with the function blocks is that they should be equivalent to integrated circuits, representing a specialized control function. The external data interface of a function block is ports and the function block implements one single function. Ports are either in or out ports, and have a type and a name. For a more detailed description of the IEC61131-3 Function block diagrams see section 4.4.6.

2.3.4 Industrial robotics

Industrial robots are typically manufactured in volumes of 10 thousands per year for the leading manufactures. That figure includes different kinds of robots. To make the manufacturing of the control system cost efficient, the same controller type is often used for different kinds of robots. Therefore, the control system has to be configurable to a large extent.

The core part of a robot system is the motion control, the application packages (welding, assembling, etc) and the way to programming the robot. Several other parts like I/O and communication protocols are provided by suppliers in many cases. Therefore, the system has to be open to enable easy integration of for example new communication protocols. Hardware is also provided by suppliers. Since the software will survive many versions of hardware, the software has to be easy to port.

From the users' point of view, the reliability is the single most important property. Today the mean time between failure (MTBF) is required to be greater than 60000 h.

Industrial robots are most often part of a larger system, e.g. an assembly line for cars, which includes PLC systems and production systems from other vendors. Therefore, each robot manufacturer has to fulfil the communication standards set by that specific customer. Consequently, all robot manufacturers have to fulfil several field bus protocols e.g., several variants of Profibus, Interbus, Foundation fieldbus, FIP, etc. Several of these protocols are described in Section 7.5.2.

System architecture

The specific architecture we will look into is the ABB Robotics control system S4. This control system was initially developed in the beginning of the nineties.

Each robot contains three closely connected nodes, a main node that generates the path to follow, the axis node, which controls each axis of the robot, and finally the I/O node, which interacts with external sensors and actuators, see Figure 2.15.

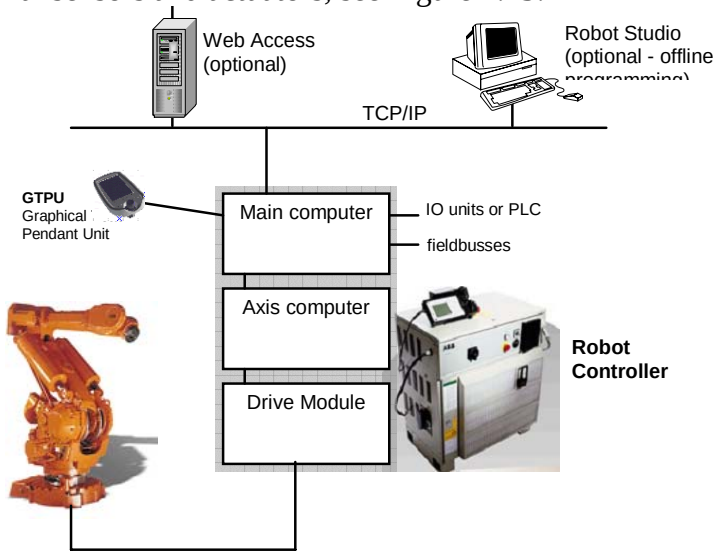


Figure 2.15 *The system architecture of a robot controller.*

Node architecture

ABB Robotics uses an object oriented approach, although most of the code is written in the imperative C-language. The controller consists of approximately 2 500 KLOC divided on

400-500 components organized in 15 subsystems. The software architecture is divided into a number of major components

Somewhat simplified each major component represent a subject area in the organization, which is responsible for that particular component. The static software architecture can ideally be visualized as a layered architecture shown in Figure 2.16. The layers are:

- an application layer,
- a high level application programming language layer,
- a core control layer, and
- a platform isolation layer.

Starting from the lowest level, the isolation layer is a general supporting layer, which implements I/O and file system routines. But also interfaces to the operating system to decouple the rest of the application from operating system internals, the operating system can be either VxWorks for the real controller or a Windows platform for a simulation application.
The second layer is the control layer, which controls the motion control of the robot. Next layer is the high level application programming layer, and it contains some sub layers and subject areas. It contains controller interfaces and external interfaces, which provide interfaces to the lower levels in the hierarchy, for access from higher levels or from external components. It also contains the robot language layer, which implements the RAPID language. Highest in the hierarchy the application layer contain the application designed for each robots particular task, it also deals with man and machine interaction.

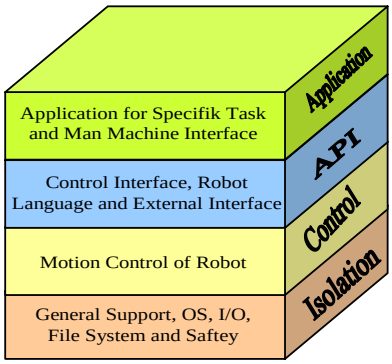


Figure 2.16 *Idealized static system architecture*

The execution model is described as event driven, built around an internal software based inter-process communication bus. The inter process communication model is based on the analogy of a parallel HW bus like the VME-bus, where the bus has a finite number of free slots for boards. A task in the system can be seen as a board that is connected to the bus via one of the available slots, see Figure 2.17.

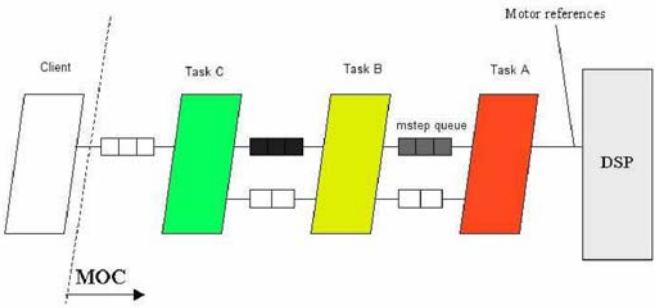


Figure 2.17. *The main on-line events are processed on the virtual communication bus*

The success of this approach is that the control system has been ported several times, from Motorola based architecture to Intel based architecture. Further, the simulation tools run the same software as the real controller.

The main challenge in the future is to move one step further in openness. To allow third parties to add their specific application add on. To open the controller for third party is technically easy, but to still guarantee an MTBF on 60000 h is the challenge.

2.4 Conclusion

The development of vehicles is going through a dramatic evolution, in their transitioning from pure mechanical devices to mechatronic machines. It is today stated by Daimler Chrysler that 90% of the innovation in the automotive lies in electronics and software. The evolution is mirrored by the fact that mechatronic modules are appearing, for example, an engine with integrated control unit.

Vehicular embedded control systems are challenging applications that have parts that are safety related, come along with a variety of real-time requirements, and that are tightly related to the process they are controlling. The control systems are increasingly being implemented in distributed computer systems and require a multitude of competencies to be developed to meet quality requirements in a cost-efficient way.

In vehicles, embedded control systems have grown from stand-alone controllers to distributed control systems. The growth has been “bottom-up” in nature. The results is that of a sudden increase in both possibilities and interaction problems. The resulting complexity is illustrated by Figures 2.1, 2.4, 2.8 and so on, showing a number of structures (mechanical, electronics hardware, software, and functional) and indicating intricate relations among them.

Standardized hardware and software platforms are emerging for aeroplanes, but for other vehicles, it is only the networks that have been standardized. Consequently, the main component used in todays embedded control systems is that of the “electronic control unit” – a complete hardware node. This today means that adding a new function is very often translated into adding a new node to a system that incorporates this functionality.

Complexity arises also in the development of these systems, and this stems not only from the product complexity, but also from the complexity of the involved organizations; a vehicle is composed of hardware components coming from a multitude of companies. It is not uncommon that a car manufacturer today in-house only develops a few control units – a very low percentage of some 70 control units indeed. This means that system integrators in some cases are trying to regain control and development of the control units because of their large impact on the vehicle. Systems integration is complicated by the fact that manual specifications are used, leaving room for misinterpretations, causing costly iterations, and highly difficult systems integration.

A number of challenges lies ahead for the development of such systems:

- Systems integration is today a central problem in development. Well defined architectures and standards are here expected to improve the situation.
- Model based development (MBD) is seen as a necessary technology, to master both the product and the development complexity. However, while MBD is practiced for subsystems, there is no holistic MBD available yet, and there are plenty of research and industrial efforts trying to develop MBD further, see e.g. Artist (2003).

- The architectures needs to take several dependability aspects into account, including safety, reliability and security. The physical integration, and both information and energy flows are here essential to consider.
- Extending the hardware oriented component approach to software components; There are several efforts striving for software component based development (see e.g. East-EAA, 2003).
- The balance in automation needs to be carefully considered. To this end, better modelling and analysis of human machine interactions is required.

References

Amberkar S, Czerny B, D'Ambrosio J, Demerly J & Murray B. 2001. A Comprehensive Hazard analysis technique for safety-critical automotive systems. SAE 2001 World Congress. Detroit, Michigan. March 5-8, 2001 SAE 2001-01-0674.

Artist (2003). Artist roadmap on Hard Real-time Systems: http://www.systemes-critiques.org/ARTIST/Roadmaps/A1-roadmap.pdf

Åström, K. J. and Wittenmark B., Computer-Controlled Systems, Theory and Design. Second edition, Prentice-Hall International, inc, 1990, ISBN 0-13-172784-2.

Augustine N R. 2000. Today… Tomorrow… of multidisciplinary systems of systems. IEEE Aerospace and Electronics Systems Magazine. Vol 15. No 10. October 2000. p 137–144.

Briere D & Traverse P. 1993. AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems. Digest of Papers, The Twenty-Third International Symposium on Fault-Tolerant Computing. August 1993

Casparsson (1998) L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg, *Volcano a revolution in on-board communications*, Volvo Technology Report. December 1998.

EAST-EAA. East-EAA – an ITEA project: *http://www.east-eea.net/*

Enslow P.H, 1978. What is a distributed system?. Computer, January 1978, pp. 13-21.

Feick S, Pandit M, Zimmer M & Uhler R. 2000. Steer-by-Wire as a Mechatronic Implementation. SAE 2000 World Congress. Detroit, MI. March 6-9. 2000. SAE 2000-01-0823.

IEEE (1992). The new IEEE standard dictionary of electrical and electronics terms. IEEE std. 100-1992. 5th edition.

Isermann (2002). Rolf Isermann et al. Fault-tolerant driv by wire systems. IEEE Control Systems. October 2002, Vol. 22, No. 5.

Kirrmann (2001). H. Kirrmann and P. A. Zuber, The IEC/IEEE Train Communication Network, IEEE Micro, March-April 2001, pages 81-92.

Larses (2003). Ola Larses. Dependable architectures for automotive electronics; Philosophy, Theory and Practice. PhD thesis, Division of Mechatronics, Dept. of Machine Design, Royal Institute of Technology, Stockholm, Sweden.

Mercedes. 2003. Innovation – Research & Technology. www.mercedes.com/e/innovation/rd/ accessed 2003-01-20.

MOST (1999). MOST Specifcation Framework Rev 1.1, MOST Cooperation, www.mostnet.de, November 1999.

Nyström (2002). Dag Nyström, Aleksandra Tesanovic, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad, Data Management Issues in Vehicle Control Systems: a Case Study, In Euromicro Real-Time Conference 2002, June 2002.

OSEK/VDX OS 2.2, http://www.osek-vdx.org

SAE Standard, SAE J1939 Standards Collection, www.sae.org

Törngren (1995). Martin Törngren. Modelling and design of distributed real-time control applications. Doctoral thesis, Dept. of Machine Design, KTH, ISSN1400-1179, ISRN KTH/MMK--95/7--SE.

Wagner G. 2003. Transmission options. SAE Automotive Engineering Online. www.sae.org/automag/features/transopt/ accessed 2003-01-20.

Wittenmark Björn, Nilsson Johan and Törngren Martin (1995). Timing Problems in Real-time Control Systems: Problem Formulation. American Control Conference, June 1995, Seattle, Washington.

Chapter 3
System and Software Life Cycle Process

Ivica Crnkovic, Mikael Åkerholm, Johan Fredriksson, Christer Norström

The chapter givens an introduction to system and software life cycle processes with focus on software development. In particular component-based approach and model-based development are addressed. The chapter discusses the advantages and disadvantages of these approaches and a possibility of their combination. The state of practice is illustrated by several case studies. Finally, the chapter summarizes the challenges and needs for further research in the domain of system and software development when applying component-based approach.

3.1 Introduction

By a system life cycle we mean all activities related to a system during its entire life: A system life cycle starts with its initiation by appearance of ideas for its development, goes true its development process, and later its delivery, maintenance and support and ends with a disposal of the system. Although the support and maintenance phase for most of systems occupies a majority of time and costs, the development phase has been in focus of research communities and main concerns in industry. The increasing maintenance costs and a need for adoption of existing products to new requirements and new environments emphasize a need for methods and technologies that support continuous evolvement of products. This is in particular for software systems or computer-based systems. New technologies such as component-based development and model-based development focus on shorter development cycles and on provide better prerequisite for efficient maintenance. However these approaches require processes that differ from traditional approaches. In these chapters we summarize the basic characteristics of these processes and discuss whether these processes are feasible for development on embedded systems. To show complexity in development and maintenance of safety-critical development systems the chapter gives overviews of several cases studies – from automotive and train industry.

3.2 System, software and hardware life cycles

3.2.1 System Lifecycle

Many products have a long life, of which the “normal” user (so called end user) only sees a small fraction. To be aware of an “entire life" of a product we must understand all the processes involved in the product’s development and operation, and of all activities of the people involved in the processes. The mechanism used to structure these processes and to identify the major activities is called a *system life cycle mode* or *product life cycle*. A system lifecycle is divided in different phases. Each phase is characterized by its inputs, results, its activities, support provided, the roles of the different people involved in the activities and the different technologies and techniques used in them. Independent of the product type, we can identify six generic phases [CR03]: the business idea of the product, requirements management, development, production, operation and maintenance, and finally disposal (see Figure 3.1).

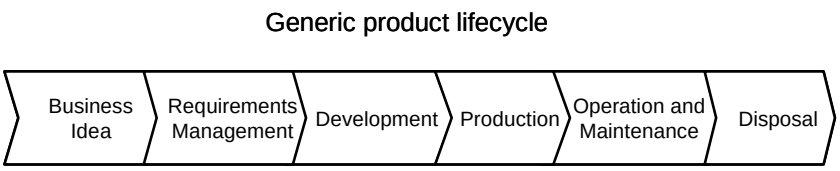


Figure 3.1. Product lifecycle

The *business idea* phase begins with a perception for a new product. It continues with an assessment of the market and technology and with the identification of the key requirements of the product. At this stage a feasibility study is performed; e.g. all available information is collected and with this as a basis it is decided if it is feasible to develop the product.

The *requirements management* phase focuses on a further identification of requirements, their analysis and their specification. The result of this phase is a product requirements specification.

The next phase, *development*, includes design and implementation activities. The result from this phase is the implementation of all the artifacts needed for production.

The *production* phase (also called manufacturing phase) is significantly different for software and hardware products. In the case of pure software products, this phase is automated to a high degree and has very low costs in comparison with the other phases; as an absolute minimum it becomes a matter of only downloading the software product. On the other hand, for a product with hardware elements this is probably the most demanding and the most costly of all the phases. Therefore, for hardware products, much effort is invested in coping with the production requirements and in keeping production costs low.

Operation and maintenance is the phase in which the product is used by consumers, and often the only phase they will see. To ensure correct operation, the product may require continuous support and maintenance.

The final phase of the lifecycle of a product is its *disposal*. The significance of this phase depends very much on the product. Different aspects not related to the product itself but to its surroundings must be taken into consideration; for example its impact on the environment, the question of its replacement etc.

3.2.2 The Development Phase

Generic system lifecycle models can be implemented in different ways. There will be particular differences in the cases of software and hardware products. To describe the main steps of pure hardware development, we can adopt a generic development process from Ulrich and Eppinger [CR03] in which the process contains six steps, as depicted in Figure 3.2 (the lower part). The development phase is preceded by *the detailed requirements management*. A detailed requirement solicitation and specification is closely connected to the development, in particular to the *conceptual development* in which the product concepts are generated. At the *system-level design*, the architecture of the product is decided, this including the identification of sub-systems and components. The components are further designed during the *detailed design*. *Testing and refinement* includes the building of product prototypes to test both the product and the production system. During the *production ramp-up*, the production system is used for serial production of the product, beginning at a low rate, but then increasing to full production.

The software development process consists of similar phases, expressed in a most illustrative way in the Waterfall model, shown in Figure 1.3 (the upper part). The model includes the following phases: Requirements analysis, design (overall and detailed), implementation, integration and test, and finally release. The *detailed requirements* analysis phase is completed with a requirements specification. This specification is an input to the system *design* which consists of

two steps; the overall design where the system architecture is designed, the components and their interfaces are identified, and the detailed design in which the component implementations are specified. In the *implementation* phase the developers follow the design documentation and implement the system in the form of algorithms and data structures specified and written in different programming languages. In the *integration and system test* phase, the integrators build the system from the components. The system is then ready for the system test. The final development phase in this model is the *release* management in which the product is packaged in an appropriate form for delivery and installation at the customer's premises.

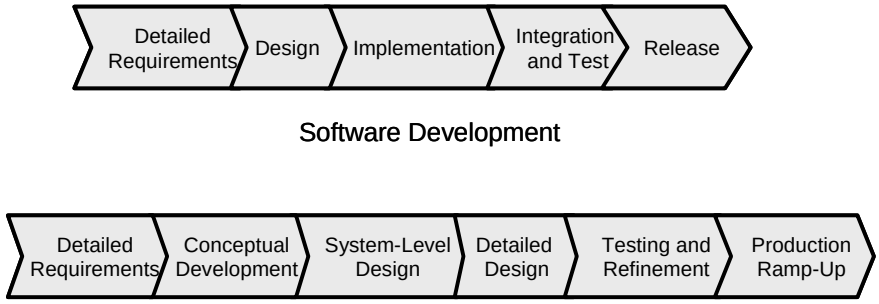


Figure 3.2. Software and hardware development processes

The waterfall model rests on the notion that it is possible to define and describe all the system requirements and software features beforehand, or at least very early in the development process. This model has an important place in software engineering. It provides a template into which methods for analysis, design, implementation, integration, verification, validation, and maintenance can be placed and clearly localized. Although in practice never used in its pure (and rather naïve) form, the sequential model has remained as the most influential software development process model. For example, the waterfall model in combination with prototyping, or V-model [So01] has been extensively used for the entire lifecycle or as a part of other models, covering particular phases of the entire process model. In particular V-model which connects specification with verification and validation phases is an established procedure in safety-critical domains. V-model is shown of figure 3.3.

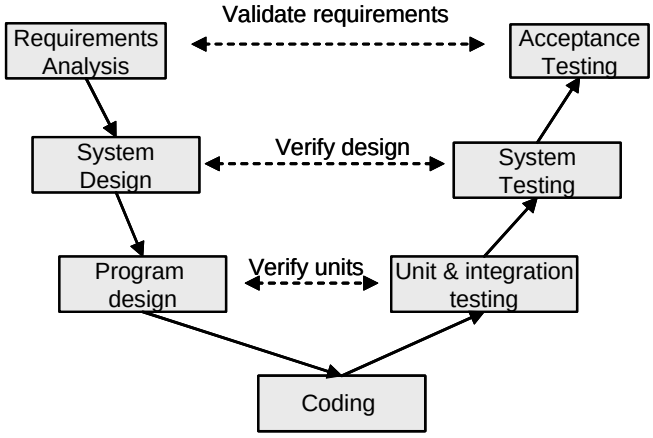


Figure3.3. V model

Both waterfall and V-model belong to a category of sequential development processes. A sequential model requires a complete specification of requirements and there can therefore be difficulty in accommodating the natural uncertainty that exists at the beginning of many projects. It is also difficult to add or change requirements during the development, as, once performed, activities are regarded as completed. In practice, requirements will change and new features will be called for and a purely sequential approach to software development can be difficult and in many cases unsatisfactory. For this and other reasons there exists other approaches. Examples of these are evolutionary approaches: incremental, iterative or spiral development.

An important question is which development processes are feasible and which apropriate for component-based approach or for the model-based approach

3.3 Component-based Development Processes

Component-based approach specifically focuses on questions related to components and in that sense it distinguishes the process of “component development” from that of “system development with components”. There is a difference in requirements and business ideas in these two cases and different approaches are necessary. Components are built to be used and reused in many applications, some possibly not yet existing, in some possibly unforeseen way. A component must be well specified, easy to understand, sufficiently general, easy to adapt, easy to deliver and deploy and easy to replace. The component interface must be as simple as possible and when used strictly separated (both physically and logically) from its implementation. On the other hand a complete specification of a component must include not only information about which services it provides but also which services and resources it requires, as well as information about other properties – this being especially true for real-time components. These requirements increase the efforts needed for the component development. According to certain experience [CL00] building a reusable component takes at least three times more efforts that building software with the same functionality but not as an reusable unit. System development with components is focused on the identification of reusable entities and relations between them, beginning from the system requirements and from the availability of components already developed [CL02]. Much implementation effort in system development will no longer be necessary but the effort required in dealing with components; locating them, selecting those most appropriate, testing them, etc. will increase [MO00].

We not only recognize different activities in the two processes, but also find that many of these activities can be performed independently of each other. In reality the processes are already separate as many components are developed by third parties, independently of system development. Even components being developed internally in an organization which uses these very same components, are often treated as separate entities developed separately.

The separation of development of components from systems has severe implications. The understanding of the components must be sufficient to successfully build the systems without looking in component internals (for example source code). A component is treated as a black box, with an extensive specification. The specification must be able to specify all properties of the component required in the system building process. Further, it must be possible to verify these properties. These facts have implications on development processes, development of systems and development of components. The sections below describe theses processes in more details.

3.3.1 Component-Based System Development Lifecycle

Development with components is focused on the identification of reusable entities and relations between them, starting from the system requirements. The early design process includes two essential steps: Firstly, specification of system architecture in terms of functional components and

their interaction, this giving a logical view of the systems and secondly, specification of a system architecture consisting of physical components. In many domains, in particular those with large volumes and many variations, principles of product-line architecture are applied. The basic characteristic of this approach is to maintain a basic architecture and obtain variability by integration of different components, or different component variants.

Different lifecycle models, established in software engineering, can be used in the system development process. These models can be modified to emphasize component-centric activities. Let us, consider, for example, the waterfall model using an extreme component-based approach. Figure 3.1 shows the waterfall model. Identifying requirements and a design in the waterfall process is combined with finding and selecting components. The design includes the system architecture design and component identification/selection.

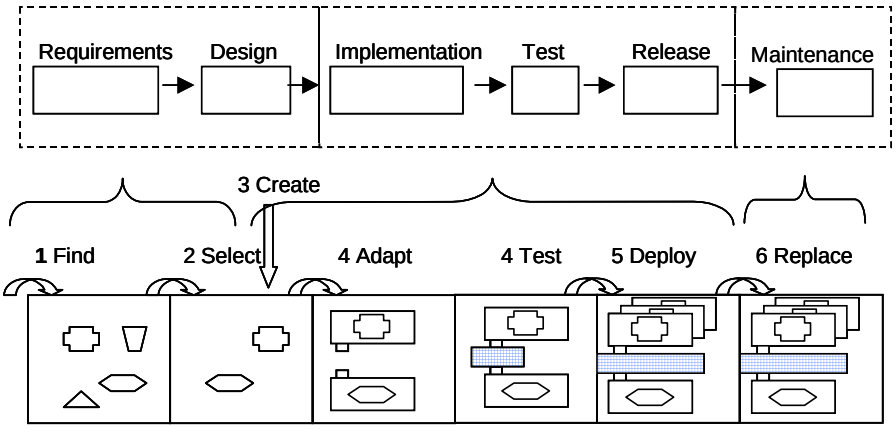


Figure 3.4 The development cycle compared with the waterfall model.

In a component-based approach the system detailed design will include *selection of the components* that meet or are close to meet the requirements or specification derived from the requirements. The selection process starts with *searching* for the components that are candidate for the selection. Alternatively a new component can be created. In a component-based development process this procedure is less attractive as it requires more efforts and lead-time. However, the components that include core-functionality of the product are likely to be developed internally as they should provide the competitive advantage of the product. The component development goal is not only to produce the component for the particular application, but also for the future applications. The next step in the selection process is component *adaptation*. Thos step may be required if the selected component does not completely match the requirements or doe not fit in the system architecture. Such a component can be *composed* with other (*instantiated*) components and in a particular phase of the system development *deployed* into the system. The system maintenance and evolution is based on the component *replacement*.

There are many other aspects of component-based development which require specific methods, technologies and management. For example, development environment tools [CL02, PO01], testing, software metrics, legal issues, project management, development process, standardization and certification issues, etc.

Requirements Analysis and Definition

The analysis activity involves identifying and describing the requirements to be satisfied by the system. Using a component-based approach, an analysis will also include specifications of the

components which are to collaborate to provide the system functionality. To be able to do this, the domain or system architecture which will permit component collaboration must be defined. In component-based development, the analysis include capturing of the system requirements and the definition of the system boundaries, the definition of the system architecture and definition of component requirements to permit the selection or development of the required components. In many cases, for example in product-line architectures, the architecture is a constant that remains unchanged, while components are evolved. In this way the process of development process is simplified as it is reduced on building or selecting new components. On the other hand the initial efforts when a new architecture is being developed may be considerably large, even to that extend that the enterprise developing it cannot afford it.

However a pure top-down approach is an optimistic and an idealized view of the process. It assumes that the component requirements can be precisely defined and that it is possible to find components which satisfy these requirements. In reality we meet several problems [Wal01]; the process of finding components to meet requirements may be very difficult. We can be almost certain that we will not find any component which exactly satisfies the requirements. The next problem that we will meet is the problem of incompatibility of selected component; even if we find components which meet the component requirements defined, it is not at all certain that they will interact as intended when assembled. Very often, when beginning from specific requirements, it is found that there is no component which meets the requirements and the only way to continue the system development is to develop a new component. In reality the process of design and even requirements engineering will be combined with component selection and the evaluation process. Experiences from COTS-based development projects at NASA show that COTS selected drives the requirements to at least some extent [MO00]. In some cases new functionality was discovered in COTS that was useful although not originally planned.

Component Selection and Evaluation

To perform a search for suitable components and make their identification possible, the components must be specified, preferably in a standardized manner. Again, this may often not be the case. The component specifications will include precisely defined functional interfaces, while other attributes will be specified informally and imprecisely (no method is developed for this) if specified at all. The components selected must therefore be evaluated. The process of evaluation will include several of both technical and non-technical nature. Technical aspects of evaluation include integration, validation and verification. Examples of non-technical issues include the marketing position of the component supplier, maintenance support provided, alternative solutions, etc.

An important method that decreases the risk of selecting a “wrong” component is to identify several component candidates, investigate these, reject those not compliant with the main requirements, continue with the evaluation of the reduced number of candidate components, if necessary refine the requirements, and repeat the evaluation process. This process is known as procurement-oriented requirements engineering [HE01].

In many cases, it may be more relevant to evaluate a set of components composed as an assembly than to evaluate a component. The availability of assemblies is more common than might be expected. Some assemblies may be purchased as functional packages which disintegrate into several components when deployed. Another example of such an assembly is a set of components which, only integrated together, constitutes a functional unit. In such cases, it is necessary to evaluate the assembly; evaluation of the individual components is not sufficient. This implies that an investigation of the integration procedure may be a part of an evaluation.

If a component is selected which only partly fulfils the specification on which the selection is based, there are two immediate alternatives: either the component is adapted to the particular specification, or the specification is adapted to the component. A third possibility is the development of a new component. From the system development point of view, this is a bad choice, if this activity were not planned at the beginning of the development process. The consequence of such a decision would require additional resources and development time.

System Design

In traditional development, the design of the system architecture is the result of the system requirements, and the design process continues with a set of sequences of refinements (for example iterations) from the initial assumptions to the final design goal. In contrast with traditional development, many decisions related to the system design will be a consequence of the component model selected. The initial architecture will be a result of both the overall requirements and the choice of component model. The component model defines the collaboration between the components and provides the infrastructure supporting this collaboration. The more service is provided by the component framework, i.e. by particular component model, the less effort on component and hence system development will be required. Thus, the choice of component model is very important. While many domains will use standard and de-facto standard component models, within particular domains with specific requirements, the specific component models providing particular services will be used, and even internally developed. This is especially true for different embedded systems domains. The design activity is very much determined by the component selection procedure. This procedure begins with the selection of component candidates and continues with consideration of the feasibility of different combinations of these. Consequently the design activity will not be a sequence of refinements of the starting assumptions, but will require a more dynamic and exploratory, and consequently evolutionary approach: The goal will be to find the most appropriate and feasible combination of the component candidates. In this way the results of the design activity may be less predictable, but on the other hand, components will automatically introduce many solutions on the design detail level [PR00,MC00,Le97].

System Integration

In an ideal component-based development the implementation by coding will be reduced to the creation of the "glue-code" and to component adaptation. Note that if the components selected are not appropriate or an inappropriate model is used, or if the components were not well understood, the costs of glue-code and component adaptation may be larger than that of the development of the components themselves! The effort for the development of glue-code is usually less than 50% of the total development effort, but effort per line of glue-code is about three times the effort per line of the application's code [Ba01]. It should be also noted that it may still be necessary to design and implement some components – those that are business-critical or unique to a specific solution and that some components will require refinement to fit into a given solution.

Integration is the composition of the implemented and selected components to constitute the software system. The integration process should not require great resources, as it is based on the system architecture and the use of deployment standards defined by the component framework and by the communication standard for component collaboration. This is however valid only for syntactic and partially semantic integration. There are several other aspects to be taken into consideration. These are component adaptation, reconfigurations of assemblies and controlling of emerging properties of assemblies integrated into the system.

- *Component adaptation*. In many cases a component must be adjusted to system requirements or to the particular architecture of the system. This adjustment can be achieved in different

ways – for example by defining component's parameters, or by building a wrapper which will manage the component's inputs and outputs must be developed, and in some cases even a new component that will control particular components and will guarantee the fulfillment of the system requirements.

- *Reconfigurations of assemblies*. Different assemblies (or composite components) can include common basic components. By introducing assemblies into the system, conflicts between the basic components can occur. In such a case some components should be replaced, or updated, which may lead to a situation that the optimal or only possible configuration consists not of a set of optimal components.
- *Emerging properties*. An important fact is that it is not possible to discover all the effects of a component composition until the integration is performed. For this reason it is necessary to include test procedures as a part of the integration, both for component assemblies and the entire system.

Verification and Validation

This last step before system delivery is similar to the corresponding procedures in a traditional development. The system must be verified and validated. Verification is a process which checks if the system meets its specified functional and non-functional requirements. A validation process should ensure that the system meets customer's expectation.

In component-based development we distinguish validation and verification of components from validation and verification of systems.

- Component Validation and Verification

  When a component is selected it should be tested to check that it functions in accordance with its specification. An evaluation of a component alone (which includes verification) is not sufficient; component assemblies in the system context must be tested. Similarly, when a component is dynamically integrated in a system, we must ensure the correct operational behavior of the system, even with failure of the component. To achieve this, we can incorporate in the system, different mechanisms such as wrappers which detect component run-time failure, and prevent its propagation to the system. We should also note that the component validation is strongly related to system validation, as the role of validation is to check out the overall system characteristics.

- System Verification and Validation

  System verification and validation process is similar to the processes for non-component-based systems. The difference is that the implications of the processes may be different – new component requirements may emerge and it may happen that the selected components cannot meet the new requirements.

System Operation Support and Maintenance

The purpose of the operational support and maintenance of component-based systems is the same as that of monolithic, non-component-based systems, but the procedures might be different. One characteristic of component-based systems is an existence of components even at run-time, which makes it possible to improve and maintain the system by updating components, or by adding new components to the system. This makes possible faster and more flexible improvement – it is not necessary to rebuild a system to improve it. In a developed component market it also gives end-users the opportunity to select components from different vendors. On the other hand,

maintenance procedures can be more complicated, as it is not necessarily clear who is supporting the system, the system vendor, or the component vendors.

3.3.2 Component Development

The component development process is in many respects similar to system development; requirements must be captured, analyzed and defined, the component must be designed, implemented, verified, validated and delivered. When building a new component the developers will reuse other components and will use similar procedures of component evaluation as for system development. There are however some significant differences: Components are intended for reuse in different products, many of them yet to be designed. The consequences of these facts are the following:

- There is greater difficulty in managing requirements;
- Greater efforts are needed to develop reusable units;
- A precise component specification is required.

Designing for Reusability

For a component to be reusable, it must be designed in a more general way than a component tailored for a unique situation. Components intended to be reused requires adaptability. This will increase the size and complexity of the components. At the same time they must be concrete and simple enough to serve a particular requirement in an efficient way. This requires much more design and development effort. Developing a reusable component requires three to four times more resources than developing a component which serves a particular purpose[CL01.

Component Specification

As the objective is to reuse components as much as possible, and as a producer is in principle not the same as the consumers, it is important that the component is clearly and properly specified. This is especially true if the component is delivered in a binary form. The consumer must be able to understand the component specification. This is where the importance of using a standardized mode of expression for component specification is evident. Component specification is discussed in Chapter 4.

3.4 Development processes for embedded systems

In widely used component technologies, the interfaces are usually implemented as object interfaces supporting polymorphism by late binding. While late binding allows connecting of components that are completely unaware of each other beside the connecting interface, this flexibility comes with a performance penalty which may be difficult to carry for small embedded systems. Therefore the dynamic component deployment is not feasible for small embedded systems. Taking into account all the constraints for real-time and embedded systems, we can conclude that there are several reasons to perform component deployment and composition at design time rather than run-time [CL00]:

- This allows composition tools to generate a monolithic firmware for the device from the component-based design.
- This allows for global optimizations: e.g., in a static component composition known at design time, connections between components could be translated into direct function calls instead of using dynamic event notifications.

- Design-time composition could be the instance of specific adaptation of components and generated code towards specific micro controller families and real-time operating systems APIs.
- Verification and prediction of system requirements can be done statically from the given component properties.

Design time composition presupposes a composition environment that specifically provides the following functionalities.

- Component composition support;
- Component adaptation and code generation for the application;
- Building the system by including selected components and components that are part of the run-time framework;
- Static verification and prediction of system requirements and properties from the given component properties.

There may also be a need for a run-time environment, which supports the component framework by a set of services. The framework enables component intercommunication (those aspects which are not performed at design time), and (where relevant) control of the behavior of the components.

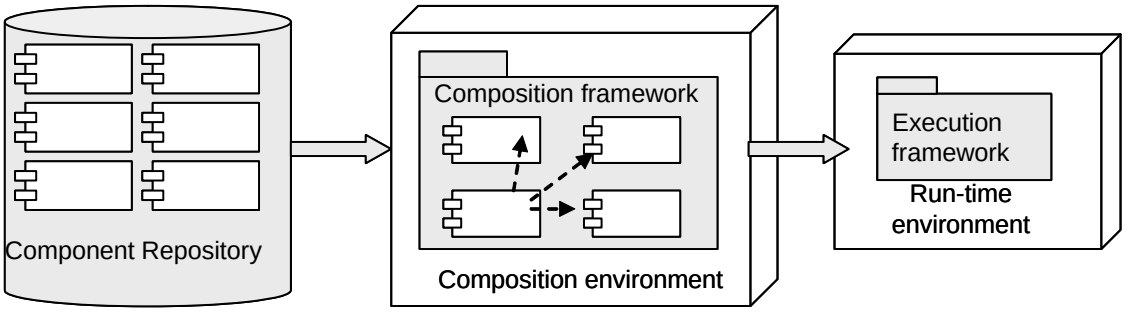


Figure 3.5 shows different environments in a component life cycle.

3.5 State of the practice

3.5.1 ABB Robotics

ABB Robotics develops industrial robots, including control systems and tools. The robots exist in several different sizes and variants for different purposes. Currently the smallest is IRB 140 with an own weight of 98 kg and a maximum lifting weight of 5 kg, while the biggest is IRB 7600 with a weight of 2500 kg and it is able to lift 500 kg and stretched it can reach up to 3,5 m. The control system is a Product Line Architecture (PLA), with small variations for the different models. The physical architecture of the control system is divided in three processors or computers, the main computer that generates the path for the robot arm, the axis computer that controls the axis of the robot, and the I/O computer, which interacts with external sensors and actuators. However, the next generation of the

control system will probably have the I/O computer and the main computer merged into one computer. The software system consists of approximately 2.5 million lines of C source code, divided into 500 classes organized in 15 subsystems. The controller is also programmable by the end customers in a vendor and domain specific language called RAPID.

Development process

The main process at ABB Robotics is shown in Fgure 3.6. It consists of several parallel processes, with different departments involved in each of the different processes. The process ensures that a continuous development takes place, all the time considering new requirements which in turn result in new releases of the products. The different processes are (i) product planning process, (ii) and (iii) product development process, hardware and software, (iv) delivery and logistics process and finally (v) sales support process.

The whole organization is built around the process and the division into departments is done according to the process; departments are organized and designated according to the subject areas. The subject areas are also visible in the resulting products, since each project is typically organized within a subject area and each development project is as independent of other projects as possible. For instance, the internal structure of the software in the controller reminds very much of the organization of the departments involved in the software development.

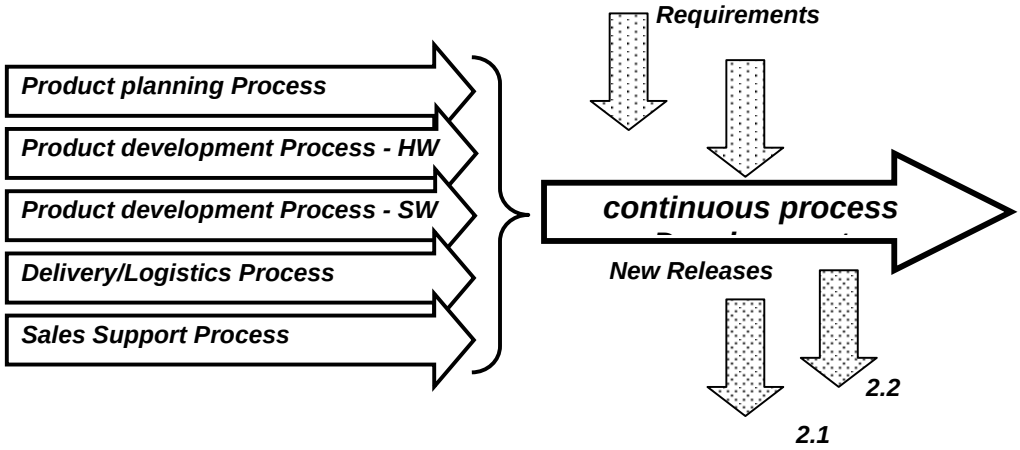


Figure 3.6, the main process and the sub-processes at ABB Robotics

The process flows through the different departments (subject areas) and process in the company is shown in figure 3.7. Development and maintenance orders, derived from the market demands are entered through the planning process. The orders are split into smaller parts, which can be given to the different departments responsible for different sub areas where the actual development will take place. The development orders are organized in several development projects which are executing in parallel. Each development project in turn is executed using the traditional V-model, and the planning and integration steps can be viewed as a shared extension to the traditional V in the V model on each of the ends.

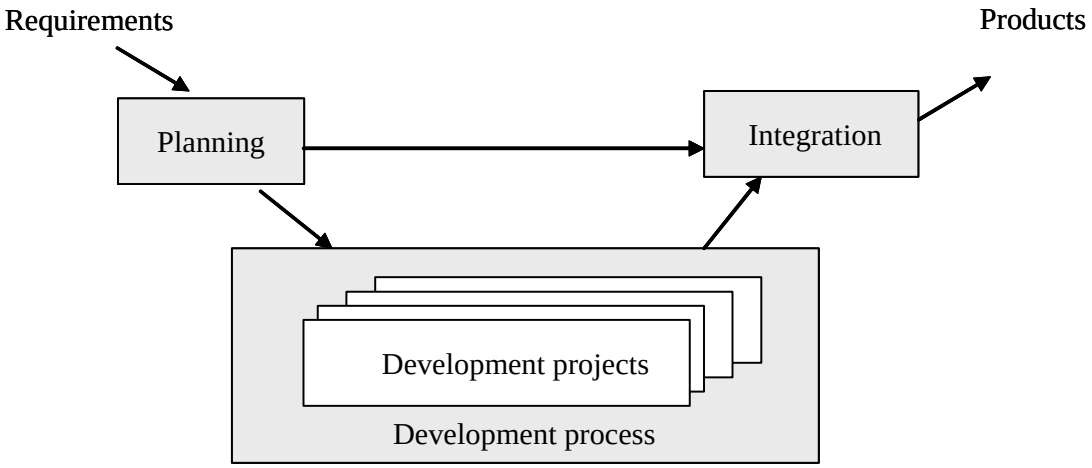


Figure 3.7, the development process

As in other cases where the V-model is applied, there are both a vertical and a horizontal flow between the different activities which is visualized in figure 3.3. The right part of the V-model at ABB Robotics includes test activities. During the last test step test centers and selected customers at different sites can be involved. This step is performed on a so called beta releases, and is designated as a beta test.

3.5.2 Volvo Construction Equipment

Volvo Construction Equipment (VCE) develops construction equipment vehicles such as of the product range of wheel loaders, excavators and articulated haulers. The architecture of the electronic control systems in these vehicles can be described as embedded distributed systems built of a number of electronic control units (ECUs) connected to each others by two buses. Functionality can be implemented within a single ECU as well as across a number of ECUs. The functionality is implanted in software and has high demands on timeliness, reliability, and for hardware cost reasons a low resource usage. A rough view of an ECU is shown in Figure 3.7: the part of the system used at run-time is the hardware layer, with I/O system, the Rubus OS and the executable part of the application layer. During design-time a development model is used, a design description is created, which is used to generate the application layer. The application layer is a high abstraction during design-time, but then transformed and optimized to low level entities interacting with the Rubus OS during configuration time.

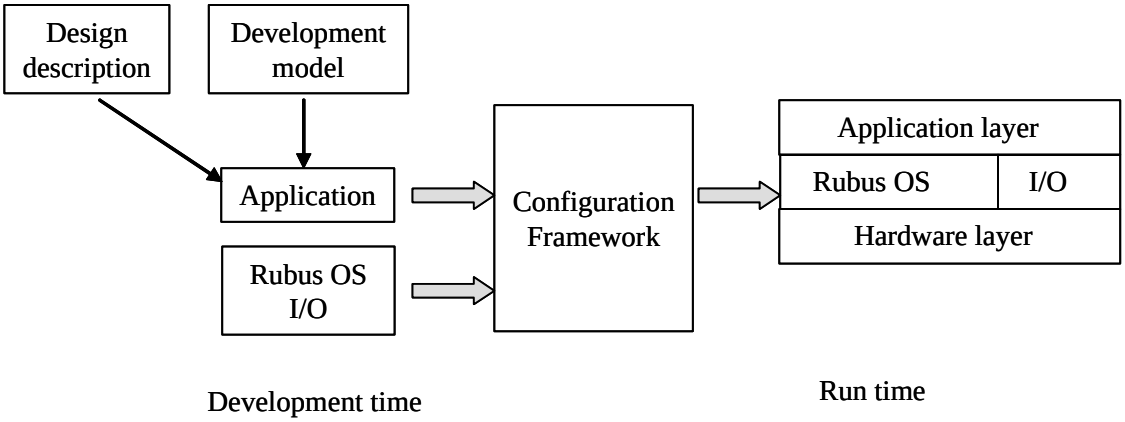


Figure 3.8 Development and run time frameworks for an ECU

Development model

The development process is divided into several stages, as figure 3.8 shows.

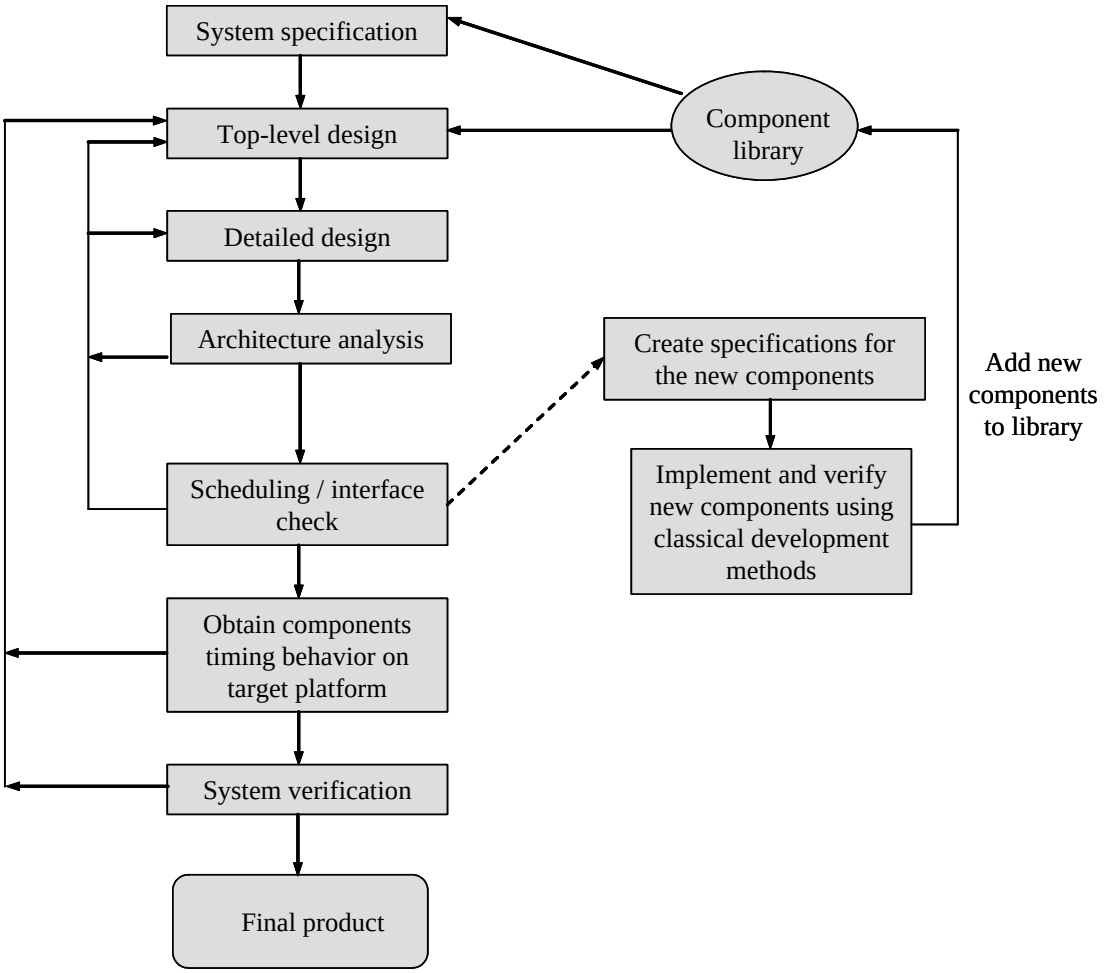


Figure 3.9 Development steps at VCE

Top-Level Design is the first stage in the actual development cycle. It involves decomposition of the system into components. The interfaces between the components are determined and functional and safety issues are associated with each component. The component library is browsed to identify a set of candidate components that could be used to build the system.
Detailed Design is the second step, which includes selecting which components to be used from the set of candidates. Once all components are identified, they are assigned attributes such as *deadline, period, release time, precedence* etc. Each component is also assigned a *time budget* in which it has to complete its execution. In other words, the time budget is a *forced worst-case execution time*.
Architecture Analysis is the step in which the *extra-functional properties*, such as *maintainability, reusability and testability*, are checked. Different approaches to perform the check include scenario-based methods, simulation-based, mathematical model based and experience based methods.
Scheduling is the step where the temporal requirements of the system are checked. A schedulability analysis is performed based on the temporal requirements of each component. If the scheduling analysis fails, changes are necessary. It may be sufficient to revise the detailed design, and replace some components with other from the candidate set. However, it may be necessary to return to the high-level design to select other components from the library or specify new components.
Timing behaviour on target platform is derived from WCET verification on the target platform. The WCET verification can either be based on measurements or static analysis of the source code. However, no tools for static analysis are available; therefore dynamic verification by running the code on the target platform is used. It is not trivial to obtain the worst case execution time from a component especially if the source code is not available. Several test executions of a component will give different execution times, and the longest execution time is obviously the WCET.
System Build and Verification is the last step before a final product. The functional and extra-functional properties must be verified for the entire system. If the verification fails, the development must return to the relevant stage of the process and correct the error.

3.6 Challenges of Component-based processes

From the process point of view the main challenges for component-based approach for embedded systems are:

- Adequate specifications of components that are developed in a separate process;
- Efficient configuration tool that can automate component selection and building procedures;
- Advanced test environment in which the components will be recognized at run-time;
- Organization that supports separation of the component development process from the integration process;
- Transition from the system requirements to the component requirements;
- Development of complex and distributed functions that may be executed on different nodes, i.e. on different system components.

3.7 Conclusion

System and software life cycle, and in particular software development process is somewhat specific for embedded systems, especially when it is about large complex systems which is a category to which vehicular systems belong. Component-based approach has a long tradition in these domains. Component-based approach of software development is however not completely established. In most of the cases system components (including hardware and software) encapsulate embedded software that is not treated separately from the hardware part. This concept works successfully so far the functions and services are directly related to these parts. When functionality and complexity of the system grow, a need for managing components that is beyond the boundaries of the system components increases. In these cases the importance of software components increases.

A separation of development process of system components is working well, while a separation of development process of software components is still not established. The same is true for separation of development of software components from hardware components. In future we can expect increase of utilization of this approach, similar what has happened in the development of general-purpose computers and operating systems. This approach has many advantages, but its success will depend on ability of the technologies and methodoligies to keep control of quality attributes related to safety, reliability, security on one side and flexibility, integrability, portability and maintainability on the other hand. Component-based design is an approach that might meet these challenges with somewhat changed basic principles which are valid today for non-embedded and non-real-time systems.

3.8 References

[Ba01] Basili V.R. and Boehm B., COTS-based Systems Top 10 List, IEEE Computer, issue May, 2001.

[CL00] Crnkovic I. and Larsson M., "A Case Study: Demands on Component-based Development", In Proceedings of 22nd International Conference on Software Engineering, ACM Press, 2000.

[CL02] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. ArtechHouse, 2002.

[CR03] I. Crnkovic, U. asklund, A. Persson Dahlqvist, *Implementing and Integrating Product Data Management and Software Configuration Management*, ArtechHouse, 2003

[HE01] Heineman G. T. and Councill W. T., Component-based Software Engineering, Putting the Pieces Together, Addison Wesley, 2001.

[Le97] Leach R., Software reuse methods, models and costs, McGraw-Hill Companies, Inc., 1997.

[MC97] McClure C., Software Reuse Techniques, Prentice-Hall Inc., 1997.

[MO00] Morisio M., Seaman C. B., Parra A. T., Basil V. R., Kraft S. E., and Condon S. E., "Investigating and Improving a COTS-Based Software Development Process", In *Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000.

[PO01] Popov P., Riddle S., Romanovsky A., and Strigini L., "On Systematic Design of Protectors for Employing OTS Items", In Proceedings of 27th Euromicro Conference 2001 Proceedings, 2001.

[Pr00] Pressman R. S., Software Engineering — A Practitioner's Approach, McGraw-Hill International Ltd., 2000.

[So01] Sommerville I., *Software Engineering*, Addison-Wesley, 2001.

[Wa01] Wallnau K. C., Hissam S. A., and Seacord R. C., Building Systems from Commercial Components, Addison Wesley, 2001.

Chapter 4
Component-based Software Development

Authors:Ivica Crnkovic, Jörgen Hansson, Mikael Åkerholm, Johan Fredriksson, Aleksandra Tešanović

This chapter gives an overview of different component models developed as research experiments, prototypes and certain models used in industry. Their basic characteristics are described. Further the chapter discusses which principles are important for component models for real-time and embedded domains. The component specification, through several types of interfaces is discussed. Also the composition principles are elaborated.

4.1 Introduction

The need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [Bos00].

Component-based software development (CBSD) is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usages. Developing systems out of existing components offers many advantages to developers and users. In component-based systems [Bos00, CL00a, CLL00, ,Fle99]:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- System evolution is eased because system built on CBSD concepts is open to changes and extensions, i.e., components with new functionality can be plugged into an existing system.
- Quality of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and, thus, changes in a component are beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user. Component-based development also raises many challenging problems, such as [LC99]:

- Building good reusable components. This is not an easy task and a significant effort must be invested to produce a component that can be used in different software systems. In particular, components must be tested and verified to be eligible for reuse.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly. The same problem arises when a new component needs to be integrated into an existing system.

- Verification of reusable components. Components are developed to be reused in many different systems, which makes the component verification a significant challenge. For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.
- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

4.2 Basic Definitions in Component Based Software Engineering

In CBSE community, there is often confusion about the basic terms. For example, component models are intermixed with the concept of component frameworks. To be able to proceed with this chapter we clarify a common interpretation of the basic terms that will be followed. Bachman et al. [BAC00] depict the relations among some of the terms in a very illustrative way as shown in Figure 4.1. The figure defines a component model as the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types. A component framework provides a variety of deployment and run-time services to support the component model.

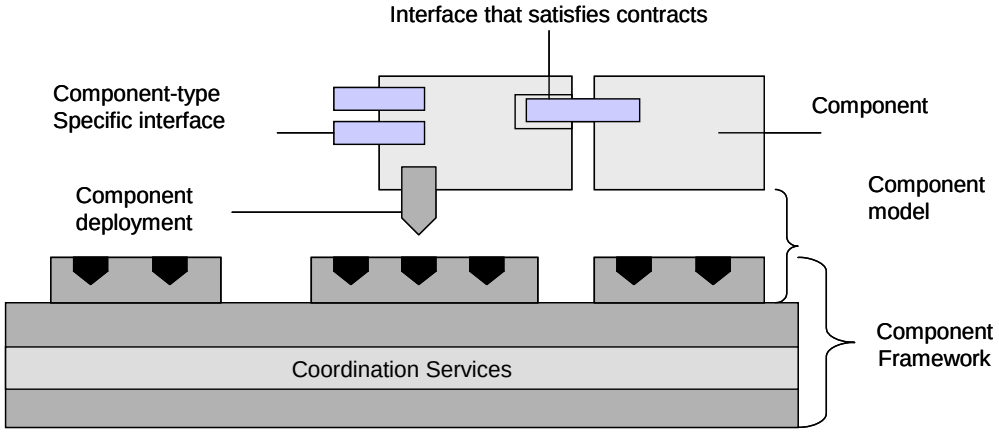


Figure 4.1. Relations between basic concepts of component technology

4.2.1 Component model

The only way that a component can be distinguished from other forms of packaged software is through its compliance with a component model. Furthermore, no agreement on what should be included in a component model exists, but a component model should specify the standards and conventions imposed on developers of components. Common is that component models deals with different component types, interaction schemes between components and clarifies how different resources are bound to components. Important parts of a component model are consequently component definitions, component interfaces and component composition.

4.2.2 Component Interface

An interface of a component can be defined as a specification of its access point [SZY98]. The clients access the services provided by the component using these points. It is important to note that an interface offers no implementation of any of its operations. Instead, it only names a

collection of operations, and provides only the descriptions and the protocols of these operations. This separation makes it possible to 1) replace the implementation part without changing the interface, and in this way improve the system performance without re-building the system, and 2) add new interfaces (and implementations) without changing the existing implementation, and in this way improve the component adaptability.

Interfaces defined in standard component technologies (for example Interface Definition Language in CORBA or COM) can only express functional properties and this only partially, focusing on the syntax part. In general, functional properties include a signature part in which the operations provided by a component are described, and a behavior part, in which the behavior of the component is specified. In this type of notation, we can distinguish export/import interfaces (also called required and provided interfaces) to/from environments that may include other components. An exported interface describes the services provided by a component to the environment, while an imported interface specifies the services required by a component from the environment.

4.2.3 Component Composition

Composition is to bring together components so that they give the desired behaviour. The possibilities for composition should be defined by the component model. Typically the possible interaction patterns are component to component and component to framework. Under composition, resource binding are also treated, in terms of early or late.

It is during composition the system is formed and it is probably at this moment predictions of run-time properties can be done by supporting tools.

4.2.4 Component frameworks

A component framework can be imagined as a small operating system which components require. It might be a standard operating system, but it is often implemented above a standard operating system, forming a middleware. A component framework typically supports one single component model. Based on arguments on what is needed to create a robust market for CBSE, arguments for standardized frameworks for different application areas exist. Two other possibilities are discussed in the literature, and all of them could be used in conjunction, one is making components easy to port between different frameworks, and another is making frameworks programmable to support different application areas.

4.2.5 Component

Software components are the core of CBSD. However, different definitions and interpretations of a component exist. In general, within software architecture, a component is considered to be a unit of composition with explicitly specified interfaces and quality attributes, e.g., performance, real-time, and reliability [Bos00]. In systems where COM [MsC01] is used as a component framework, a component is generally assumed to be a self-contained binary package with precisely defined standardized interfaces [MB+99]. Similarly, in the CORBA component framework [OMG01], a component is assumed to be a CORBA object with standardized interfaces.

A component can be also viewed as a software artifact that models and implements a well-defined set of functions, and has well-defined (but not standardized) component interfaces [DG00].

Hence, there is no common definition of a component for every component-based system. The definition of a component clearly depends on the implementation, architectural assumptions, and the way the component is to be reused in the system. However, all component-based systems

have one common fact: components are for composition [SZY98]. This definition is also the best accepted definition in the software industry in the world [SZY98]:

A component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition.

Common for all types of components, independent of their definition, is that they communicate with their environment through well-defined interfaces, e.g., in COM and CORBA interfaces are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL. Components can have more than one interface. For example, a component may have three types of interfaces: provided, required, and configuration interface [Bos00]. Provided and required interfaces are intended for the interaction with other components, whereas configuration interfaces are intended for use by the user of the component, i.e., software engineer (developer) who is constructing an application out of reusable components. Each interface provided by a component is, upon instantiation of the component, bound to one or more interfaces required by other components. The component providing an interface may service multiple components, i.e., there can be a one-to-many relation between provided and required interfaces.

When using components in an application there might be syntactic mismatch between provided and required interfaces, even when the semantics of the interfaces match. This requires adaptation of one or both of the components or an adapting connector to be used between components to perform the translation between components.

Independently of application area, a software component is normally considered to have *black box* properties [Fle99,DG00]: each component sees only interfaces to other components, thus, internal state and attributes of the component are strongly encapsulated.

Every component implements some field of functionality, i.e., a domain [Bos00]. Domains can be hierarchically decomposed into lower-level domains, e.g., the domain of communication protocols can be decomposed into several layers of protocol domains as in the OSI model. This means that components can also be organized hierarchically, i.e., a component can be composed out of subcomponents. In this context, two conflicting forces need to be balanced when designing a component. First, small components cover small domains and are likely to be reused, as it is likely that such component would not contain large parts of functionality not needed by the system. Second, large components give more leverage than small components when reused, since choosing the large component for the software system would reduce the cost associated with the effort required to find the component, analyze its suitability for a certain software product, etc. [Bos00]. Hence, when designing a component, a designer should find the balance between these two conflicting forces, as well as actual demands of the system in the area of component application.

4.3 Component-based Approach for Embedded Systems

A basic idea in component based software development is to structure a system into components. In classic engineering disciplines, a component is a self-contained part or subsystem that can be used as a building block in the design of a larger system. It provides specified services to its environment across well-specified interfaces. Ideally, the development of a component should be decoupled from the development of the systems that contain it. Components should be reusable in different contexts.

Szyperski [SZY98] tends to emphasize that components should be delivered in binary from, and that deployment and composition should be performed at run-time. However in the second edition of his book [SZY02] Szyperski extends the binary form to a more general “executable”.

In the domains of embedded systems this definition is largely followed, in particular the separation between component implementation and component interface. However the demands on the binary or executable from is not directly followed. A component can be delivered in a form of a source code written in a high-level language, and allows build-time (or design-time) composition. This more liberal view is partly motivated by the embedded systems context, as will be discussed in below.

Component-based development is an attractive approach in the domains of embedded systems. As for other domains there are two main benefits specific to component technology. First, it gives structure to system design and system development, thus making system verification and maintenance more efficient. Second, it allows reuse of development effort by allowing components to be reused across products and in the longer term it enables building a market for software components. In particular for the development of high volume and many variants of products the component-based approach is attractive. In spite of this attractiveness the adoption of component-based technologies for the development of real-time and embedded systems is significantly slower. Major reasons are that embedded systems must satisfy requirements of timeliness, quality-of-service, predictability, that they are often safety-critical, and can use severely constrained resources (memory, processing power, communication). The widely used component technologies such as EJB, .NET, CORBA component models are inherently heavyweight and complex, incurring large overheads on the run-time platform; they do not in general address timeliness, quality-of-service or similar extra-functional properties that are important for real-time systems. In their present form they start to be deployed in large, distributed, and not safety critical systems, e.g., in industrial automation, but are not suitable for deployment in most embedded real-time environments

In most of the cases embedded systems are real-time systems. In many cases embedded systems are safety or mission critical systems. Embedded systems vary from very small systems to very large systems. For small systems there are strong constraints related to different recourses such as power or memory consumption. For these as well as for large embedded systems the demands on reliability, robustness, availability and other characteristics of dependable systems are important. Finally, in many domains, the product life cycle is very long – it can stretch to more than several decades.

All these characteristics have strong implications on requirements. The most of the requirements of embedded systems are related to non-functional characteristics (better designated as extra-functional properties). These properties can be classified in run-time and life cycle extra-functional properties. The most important properties are real-time properties, resource consumption, dependability, in particular reliability, safety, availability, security and integrity, and life cycle properties such as maintainability, portability, and modifiability.

Importance of extra-functional properties has an implication that development and maintenance of such systems are very costly. In particular activities related to verification and guaranteed behaviour (formal verification, modelling, tests, etc.) and maintenance (adaptive maintenance, debugging, regressive testing, etc.) require a lot of efforts. For these reasons the technologies and processes that lead to lower costs for these activities are very attractive and desirable.

4.3.1 Components for Embedded Systems

Many important properties of components in embedded systems, such as timing and performance, depend on characteristics of the underlying hardware platform. Kopetz and Suri [Kop03] propose to distinguish between *software components* and *system components*. Extra-functional properties, such as performance, cannot be specified for a software component in isolation. Such properties must either be specified with respect to a given hardware platform, or be parameterized on

(characteristics of) the underlying platform. A system component, on the other hand, is defined as a self-contained hardware and software subsystem, and can satisfy both functional and extra-functional properties.

Further we can distinguish *encapsulated software components* that are encapsulated in system components and *distributed software components* which specify a particular function or service, but may be distributed in several system components.

4.3.2 Component Interfaces Suitable in Embedded Systems

The component interface summarizes the properties of the component that are externally visible to the other parts of the system. An interface may list the signatures of operations, in which case it can be used to check that components interact without causing type mismatches. An interface may contain additional information about the component's patterns of interaction with its environment or about extra-functional properties such as execution time; this allows more system properties to be determined when the system is first designed.

The information in component interfaces facilitates the check for interoperability between components. Additional component specification enables verification of system requirements and prediction of system properties from properties of components. This allows several system properties to be verified and predicted early in the development life cycle, enables early design space exploration, and saves significant effort in the later system integration phase.

4.3.3 Component Deployment and Composition in Embedded Systems

In widely used component technologies, the interfaces are usually implemented as object interfaces supporting polymorphism by late binding. While late binding allows connecting of components that are completely unaware of each other beside the connecting interface, this flexibility comes with a performance penalty which may be difficult to carry for small embedded systems. Therefore the dynamic component deployment is not feasible for small embedded systems.

Taking into account all the constraints for real-time and embedded systems, we conclude that there are several reasons to perform component deployment and composition at design time rather than run-time [CL02]:

- This allows composition tools to generate a monolithic firmware for the device from the component-based design.
- This allows for global optimizations: e.g., in a static component composition known at design time, connections between components could be translated into direct function calls instead of using dynamic event notifications.
- Design-time composition could be the instance of specific adaptation of components and generated code towards specific micro controller families and real-time operating systems APIs.
- Verification and prediction of system requirements can be done statically from the given component properties.

Design time composition presupposes a composition environment that specifically provides the following functionalities.

- Component composition support;
- Component adaptation and code generation for the application;

- Building the system by including selected components and components that are part of the run-time framework;
- Static verification and prediction of system requirements and properties from the given component properties.

There may also be a need for a run-time environment, which supports the component framework by a set of services. The framework enables component intercommunication (those aspects which are not performed at design time), and (where relevant) control of the behaviour of the components.

Figure 4.2 shows different environments in a component life cycle. The figure is adopted from [CL02].

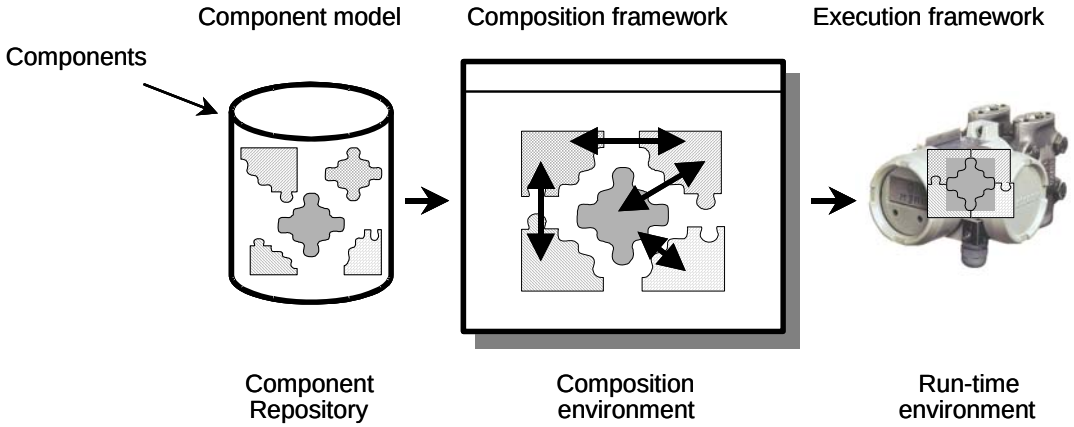


Figure 4.2 Component technology for embedded systems

4.4 Existing Component Technologies for Embedded Systems

In this section existing component technologies for embedded systems are described, the technologies are both from academia and industry. The selection criterion for a component technology has firstly been that there is enough written information, secondly that the authors claim that the technology is suitable for embedded systems, and finally to achieve a combination of both research and industry examples. Technologies that have been included are PECT, Koala, Rubus Component Model, PBO, PECOS, IEC 61131, CORBA based technologies and Vest.

The descriptions of each technology focus on the two main parts component model and component framework. The component model is easiest explained by the component definition, the component interfaces and rules for composition. While the component framework is defined as the necessary run-time mechanisms, and under this part development tools are also mentioned. In the end of each component technology the advantages and shortcomings according to the suitability for usage in typical mission critical resource constrained embedded systems with real-time requirements.

4.4.1 PECT

A Prediction-Enabled Component Technology (PECT) [WAL03] is a development infrastructure that incorporates a component technology, development tools and analysis techniques. PECT is ongoing research at the Software Engineering Institute (SEI) at Carnegie Mellon University. The idea is that any component technology can be used in the bottom but composition rules enforced by the development tools guarantee critical runtime properties, thus a PECT enforces that

predictable construction patterns are used. What is allowed by a user and what is required by the underlying component technology are determined by the available analysis methods and prediction goals.

Component model

When describing a PECT as in figure 4.3, a component technology contains a component model and a runtime environment. A component model mainly specifies component types, interfaces and interaction mechanisms. A runtime environment enforces aspects of the component model, serving the context in which the components execute analogous to the role of an operating system. A PECT is an abstract model of a component technology, consisting of a construction framework and a reasoning framework. The construction framework is mainly development tools utilising a construction language. The reasoning framework is analysis methods for different runtime aspects supported by the construction language.

Component definition

Components are defined to be implementations in its final form that provide interfaces for third party composition and are units of independent deployment. The reason why to define a component as an implementation in its final form, is to distinguish it from abstractions in architectural description languages. The term *final form* is used because the component should generally be delivered in the form to be executed, rather than source code.

The term *unit of independent deployment* is quite subtle. It is described as “all the components dependencies on external resources are clearly specified” and a component shall “conceivably be a substitute for some other component”.

The graphical notion of a component is shown in figure 4.4.

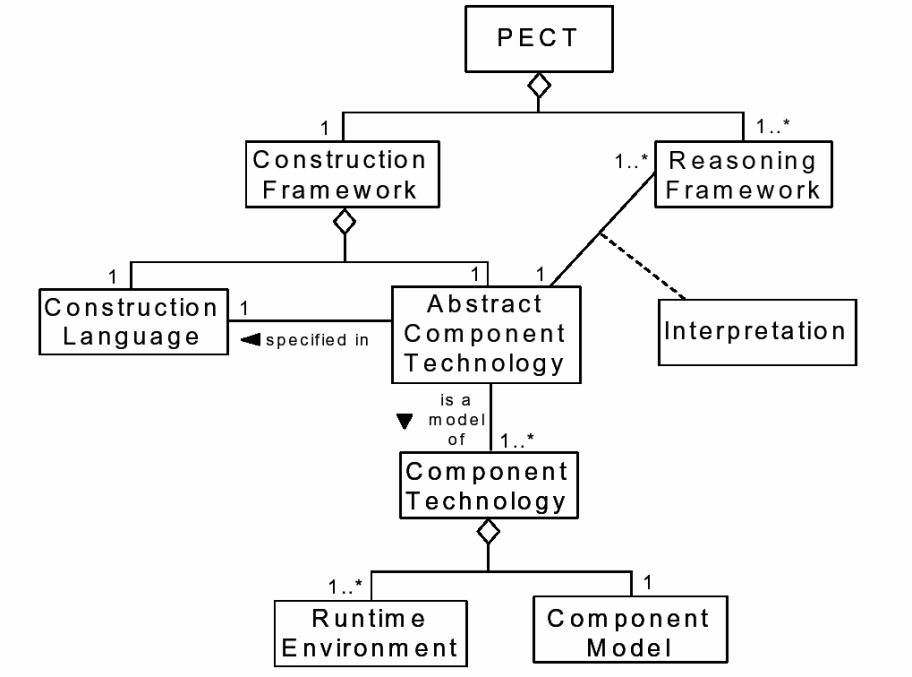


Figure 4.3, UML class diagram of PECT concepts

Component interface

As figure 4.4 shows a component is denoted as a box with a label that indicates the name of the component, to the left and to the right of a component there are incoming and outgoing arrows which are the pins. Incoming arrows are sink pins and denote incoming events or other interactions such as procedure calls, while outgoing arrows are source pins and represents outgoing events or procedure calls. The labelling convention of sink pins are to start with an 'S' indicating that it is a sink pin, followed by an index and possibly a ':' with the name of a thread after 'tx'. Thread does not denote a particular implementation concept, but is a unit of concurrent execution in the component technology. If a particular thread is specified in the label of sink pin it means that the required action is performed by that particular thread, otherwise the action is performed in the context of the caller's thread. Threads may be shared within the sink pins of a particular component, but not across component boundaries.

Component composition

The construction framework defines an Abstract Component Technology (ACT), which should be used by a user regardless of the actual underlying component technology. The usage of the ACT restricts the usage of the underlying component technology so that assemblies are analysable within the reasoning framework. A graphical language for assembling components is proposed, the language uses a component abstraction with sink pins and source pins.

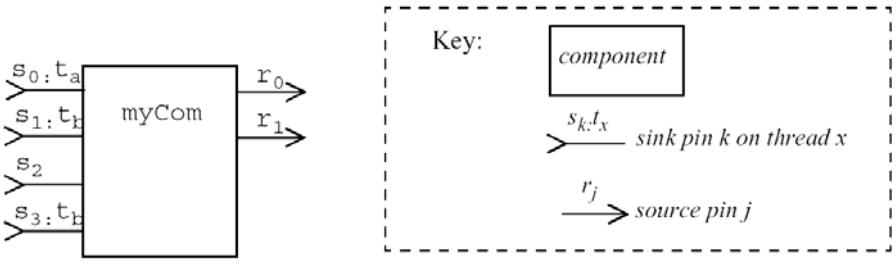


Figure 4.4, Graphical notion of components

Being able to predict the runtime behaviour of an assembly requires knowledge about the behaviour of a component. To specify behaviour reactions are used. To make the predications automated, a requirement is that the reaction is specified in a parsable syntax. Currently the CSP process algebra [HOA85] is used, but less complex alternatives are explored. Composition of components is carried out by connecting sink and source pins, the behaviour of a composition is predicted from the interacting reactions.

During the development phase in a project using a PECT technology, the developers have tools helping them to predict run-time properties. The tools should be generated according to appropriate reasoning frameworks. A reasoning framework contains a property theory, an automated reasoning procedure and a validation procedure. Imagine a simple Fixed Priority Scheduled (FPS) real-time system with a Rate Monotonic (RM) priority assignment, then an execution time analysis reasoning framework would consist of (1) A property theory for predicting execution time, e.g. Liu and Layland 1973 [LAY73], (2) An automated reasoning procedure, simply an implementation and (3) an validation procedure, the analysis depend on measured execution times so statistical confidence can be used.

Component Framework

As mentioned the intention with PECT research is to provide a development framework to be applied above an existing component technology. So the component framework in this case must be the one provided by the underlying component technology. PECT enforces rules on how that

framework can be utilised in a predictable manner, according to the prediction goals and existing reasoning theories, rather than providing one on its own.

4.4.2 Koala

The Koala component technology [OMM00] is developed and used by Philips for development of software in consumer electronics, currently by more than 100 developers. Typically, consumer electronics are resource constrained systems since they are using cheap hardware components to keep development costs low.

Component Model

Koala is a light weight component model, tailored for Product Line Architectures. Currently no third party components are integrated in Koala based products; all components are developed in house by Phillips.

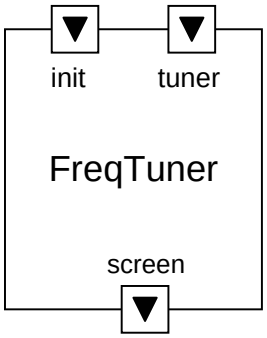
Component Definition

The Koala components are units of independent design, development and reuse, they can interact with the environment or other components through explicit interfaces only. Because of this, no two basic Koala components have any dependencies to each other. Furthermore the source code of koala components are fully visible for the developers they are not binary or in any other intermediate format.

Component Interface

There are two types of interfaces in the Koala model, namely provides and requires interfaces. A component may have multiple interfaces, which can be seen as a good way of handling evolution and diversity.

Provides interfaces specifies methods to access the component from the outside, figure 4.5 show a graphical notion of a simple Koala component, the corresponding component definition and interface definition. The graphical notation indicates that the init and tuner are provides interfaces by the direction of the arrows inside the connection pins, while screen is of requires class. The bounding interface of the component is described by a Component Description Language (CDL) to the upper right in the figure, and each interface is specified in a Interface Definition Language (IDL) at the bottom in the figure. Both the IDL and CDL have C influenced syntaxes.




```
component FreqTuner{
provides  IInit init;
          ITuner tuner;
requires  IScreen screen;
}
```

```
interface IInit{
void initialize(void);
}
```

```
interface ITuner{
int getFrequence(void);
void setFrequency(int
f);
}
```

```
interface IScreen{
void updatePicture
(void);
}
```

Figure 4.5, a Koala component

Component composition

Components can be composed from other components and an application is called a configuration, which consist of components with matching interfaces connected to each other. All requires interfaces of a component must be bound to exactly one provides interface, while each provides interface can be bound to zero or more requires interfaces. All communication in a configuration built with Koala components is carried out utilising these interfaces, even calls to the operating system.

The bindings between components in Koala are static and resolved during compile time. Static binding is suitable for resource constraint systems, it is efficient, predictable and 90% of the bindings are known at design time and it is known which 10 % of the bindings that must remain flexible and also to which extent [CL02]. Koala has some features that allow for additional binding flexibility and easier evolution. When two components are bound through a connection of their provides and requires interfaces their functions are connected to each other on the basis of their name.

Koala allows that provides interfaces are wider than requires interfaces, the provides interface must implement at least what the requires interface specifies but it may implement more than that. This feature makes evolution of components easier, since it is possible to add new functionality into a component still keeping compatibility with former versions.

When interfaces do not match, but a developer needs to connect two components anyway it is possible to add glue code that forms a connector between the two components. Glue code can be written in C or in a limited expression language within Koala.

The necessary degree of binding flexibility can be achieved with switches. A switch chooses between provides interfaces offered by different components at run time, but is examined by the compiler at compile time. The compiler tries to perform certain optimizations, such as reducing the switching possibilities and remove unused connections and perhaps even components. A switch can also be reduced to a straight binding by the compiler, if the switch is set to known position at run-time without being touched.

Component Framework

The component framework is modelled in the Koala based design as a set of koala components. The framework can be divided into two different layers, one computing layer which provides an API with a higher abstraction to the computing hardware, and another layer that provides an API to the audio and video hardware in the products. Components from an application can use arbitrary components from any layer in the framework, and within the framework components from the audio video layer can use components from the computing layer but not in the other direction.

The most basic functionality in the framework is located in the computing layer, which can be compared to a very simple real-time kernel with priority driven pre-emptive scheduling. Another run-time mechanism provided to keep the number of threads low, which in turn affect the memory utilization, is a technique called thread sharing. This results in a form of late binding used for autonomous activities. The implementation is through pumps, which are message queues with a function to process messages in the context of a thread.

4.4.3 RUBUS component model

Rubus component model supported by Rubus Visual Studio, the development environment shipped with the Rubus RTOS is tailored for resource limited systems with real-time requirements. The purpose and main objective with the model is to make it easier to reuse parts of systems and to maintain small differences between similar products, which is essential when their customers maintain their product lines. Rubus has a red and a blue part for hard and soft real-time respectively. The red part (red kernel) is used for time-critical applications. The red-part is therefore time-triggered. The blue part on the other hand is event-triggered, and used for less time-critical applications.

Component model

A basic software component consists of behaviour, a persistent state, a set of in-ports and out-ports and an entry function. The thread of execution for a component is provided by a Rubus task. The component technology is port-based and uses state-based communication. The communication between time-critical components is unbuffered.

Component definition

A component consists of one or more tasks, which are the run-time entities executed by the operating system. A task in turn is defined by a function pointer, data ports, and some attributes building the current configuration of the task. A composite can also be identified, which is a logical composition of one or more components. Components and composites are technically the same, but an encapsulation of components is logically separated from an encapsulation of tasks by different notions. Figure 4.6, is a UML meta-model showing the described items and relations. The abstractions above the task component and composite is only design time structures.

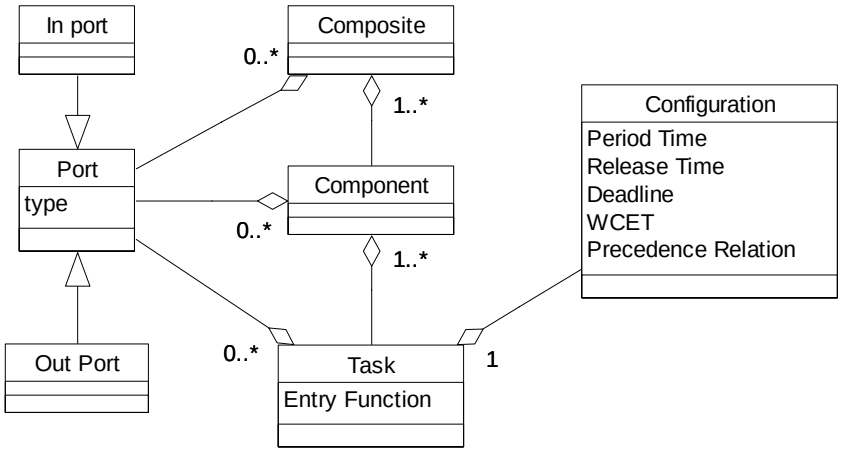


Figure 4.6, A UML meta-model of the relations between different items

Component interface

The component interface in the Rubus component model is port based. A port is a shared variable. Each component has in and out ports. The communication in the time-critical part (red part) is unbuffered. In Rubus there are also non-functional interfaces, with which timing properties are defined. The non-functional interfaces are specified by release-time, deadline, worst case execution times and a period. No functional properties are defined by any interfaces. However, during system design it is possible to define precedence relations, ordering and mutual exclusion.

Component composition

As in figure 4.6, components and composites can be used as units of independent reuse, although the run-time structures of components are a set of tasks. The tasks communicates with each others through typed data ports, on activation a task reads the data on its in port, executes its function and finally writes the result of the computation to its out port. From a design view the communication scheme yields a loose temporal coupling between tasks. A loose coupling in time is realized, since send and receive actions may take place asynchronously completely independent on each others. On the other hand the scheme creates a hard coupling in space since the sender and receiver of a data have to be specified during design time, when connecting ports between tasks.

The Configuration Compiler (CC), is a tool used to verify and construct scheduling schemes for the Rubus operating system. Input to the tool is the design described in a formal specification language and the specification of the available CPU capacity. If the CPU capacity is enough the tool generates a schedule, for the Rubus operating system which specifies when each task shall be executed. The scheduling algorithm in CC is based on a heuristic tree search strategy, with interrupt handling [SAN98]. CC is configured to not allow any preemption, which means that a task cannot be premepted by another. This in turn means that any protection of shared resources becomes unnecessary, since tasks are separated in time.

Component framework

The Rubus component model is developed on top of the Rubus Real-Time Operating System. Rubus is a real-time operating system with support for both static scheduling and pre-emptive fixed priority scheduling. The static and pre-emptive fixed priority schedulers are referred to as

the *red* and *blue* parts of Rubus respectively. The red part only handles hard, static real-time, whilst the blue part only handles soft real-time. Finally, there is also a green part of Rubus, which is an interrupt handler kernel. It has the highest priority, and distributes and routes the interrupts.

The red part of Rubus always has higher priority than the blue part. Therefore, the red part is used for time critical operations (firm and hard real-time), since it is easier to verify timing properties of the red part. The blue part of Rubus is usually used for more dynamic properties, and soft real-time. The green part always handles all interrupts, and has the highest priority in the system.

4.4.4 Port Based Objects

Port Based Objects (PBO) [STE97] combines object oriented design, with port automaton theory. PBO was developed as a part of the Chimera RTOS project at the Advanced Manipulators Laboratory at Carnegie Mellon University. Together with Chimera, PBO forms a framework aimed for development of sensor-based control systems, with specialization in reconfigurable robotics applications.

Component model

A pronounced design goal for a system based on PBO is to minimize communication and synchronization, thus facilitating reuse.

Component definition

A PBO is also called a control module, and is a software component. A PBO is defined as an object with various ports for real-time communication. As any object it has a state, but unlike ordinary objects the methods are hidden, it is only the ports that are visible from the outside. To be strict you could say that a PBO is neither an object nor a true software component, but with a more practical interpretation of definitions it can be depicted as both.

Component interface

The ports of an object may be classified as input-, output- or resource ports. Input and output ports are used for communication between collaborating objects, while resource ports are aimed for communication with sensors, actuators or other external devices or sub-systems. A PBO may have an arbitrary number of ports of each class. A graphical notation of a PBO is shown in figure 4.7. A PBO is drawn as a round-corner rectangle, with its input- and output ports are drawn as arrows entering/leaving the sides of the rectangle. To the top and bottom we have configuration constants and resource ports marked as double directed arrows respectively.

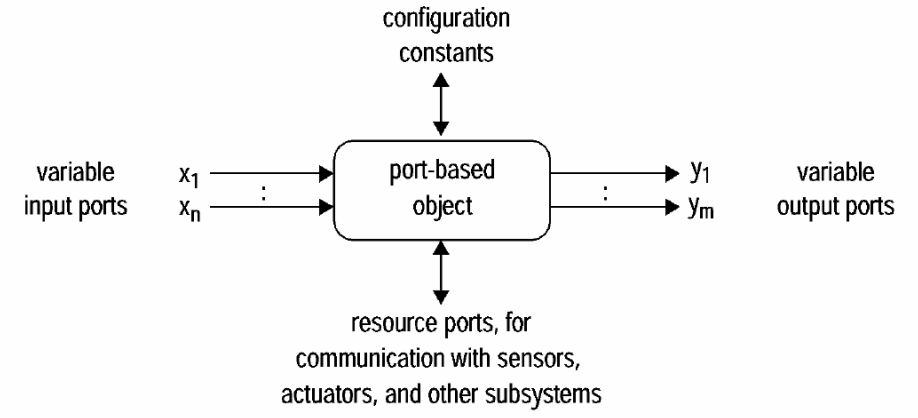


Figure 4.7, Architectural view of a port based object

Component composition

When building an application of PBOs, all existing ports must be connected for the configuration to be valid. The state variable communication between PBOs are created by the user by connecting input and output ports, the details are hidden and taken care of by the framework. The framework builds tables with state variables, every I/O port and configuration constant is stored as a state variable in shared memory. Every PBO is served through a local table with its own subset of state variables, there is no synchronization needed for read and writes to the local table since it is only accessed by one PBO, which can thus execute independently of other PBOs. Consistency between the local and global table is maintained by a simple execution semantic. State variables corresponding to input ports are updated just before executing a PBO, and are copied from the global table to the local table. State variables corresponding to output ports are updated after the execution of a PBO, and are copied in the opposite direction from the local table to the global table, so during its execution a PBO can write to its output ports at any time.

The port-automaton theory provides methods for modelling transfer functions for each PBO, describing the output response for a given input. A closed or open loop system can be analysed by applying the transfer functions according to the connection scheme. Timing analysis based on each PBOs Worst Case Execution Time (WCET) and analysis of the state variable communication are also available both for single and multiprocessor environments. A graphical view of a typical control application built with PBOs is shown in figure 4.8. In the bottom of the figure, the resource ports are drawn and connected to various sensors, actuators and sub-systems. The PBOs are connected to each other by in and out ports to the sides of them, forming the look of a classical closed loop system familiar for control engineers. To the top of the gfwdkin, ginvkin and mms objects some configuration constant connections are drawn.

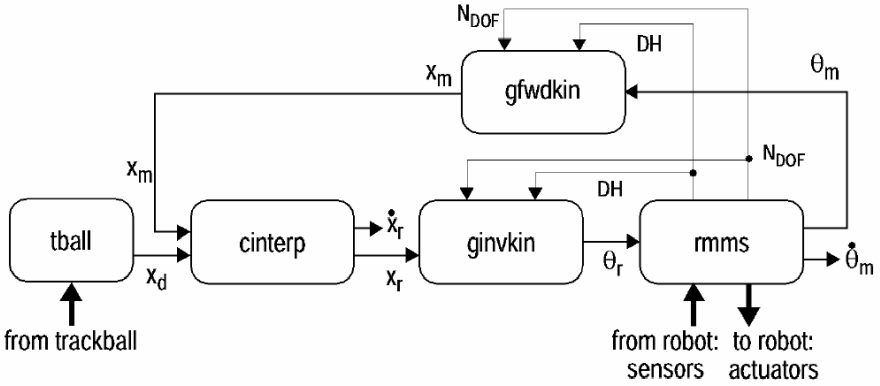


Figure 4.8, a control application based on PBO

Component Framework

The PBO model is aimed for execution on the Chimera RTOS [KHO92]. Chimera is a real-time operating system with multiprocessor support designed especially to support the development of software for robotic and automation systems. The code is compiled and linked with an ordinary SUN C compiler and linker.

Chimera provides typical RTOS kernel task management features, such as creating, suspending, restarting, pre-empting and scheduling. The kernel schedules the tasks using virtual timers, all based on a hardware timer. Chimera supports both static and dynamic scheduling of real-time tasks. The default scheduler supports the rate monotonic scheduling algorithm (static scheduling), the earliest-deadline-first scheduling algorithm (dynamic scheduling) and the maximum-urgency-first scheduling algorithm (static and dynamic scheduling).

A task can communicate or synchronize with any other task through shared memory, high-performance semaphores or user signals. Above this many different types of interprocess communication and synchronization mechanisms are built in as layers, in purpose to simplifying the development of complex applications. In particular for applications built with PBO it is the state variable mechanism described in the composition chapter that is used as a mapping for the data ports.

4.4.5 PECOS

PECOS [NIE02] is a collaborative project between industrial and research partners. The goal for the PECOS project is to enable component-based technology for embedded systems, especially for field devices. The project tries to consider non-functional properties very thoroughly in order to enable assessment of the properties during construction time.

Component model

As many other component technologies PECOS separates the development of the application and the components. The component model in PECOS is aimed to express, functional interfaces, e.g., procedural interfaces and non-functional properties and constraints.

Component definition

A component in the PECOS model is a computational element with a name, a number of property bundles and ports, and a defined behaviour. The ports of the component represent data that may be shared with other components. The behaviour of a component consists of a procedure that reads and writes data available at its ports.

The formal definition of components in PECOS is realized with a language called CoCo, developed within the PECOS project. CoCo is intended to be used for both the specification of the components and the specification of field device applications, built as compositions. CoCo supports the basic elements components, ports and properties, and in addition for composite components instances and connectors. Properties are the characteristics of a component and are represented in CoCo with a tag and a value, example of properties is period time and execution time.

The type of a component can be of one of the three passive, active or event. Passive components are as the name reveals a component without a thread of control. They are scheduled by their closest active ancestor. Active components are the opposite of passive components, and have their own thread of control. Event components are components that are triggered by an event and have a thread of control. They are often used to model hardware that frequently emits events.

Figure 4.9, shows a UML Meta model of a component, defining relations between different types of component and properties characterising a component.

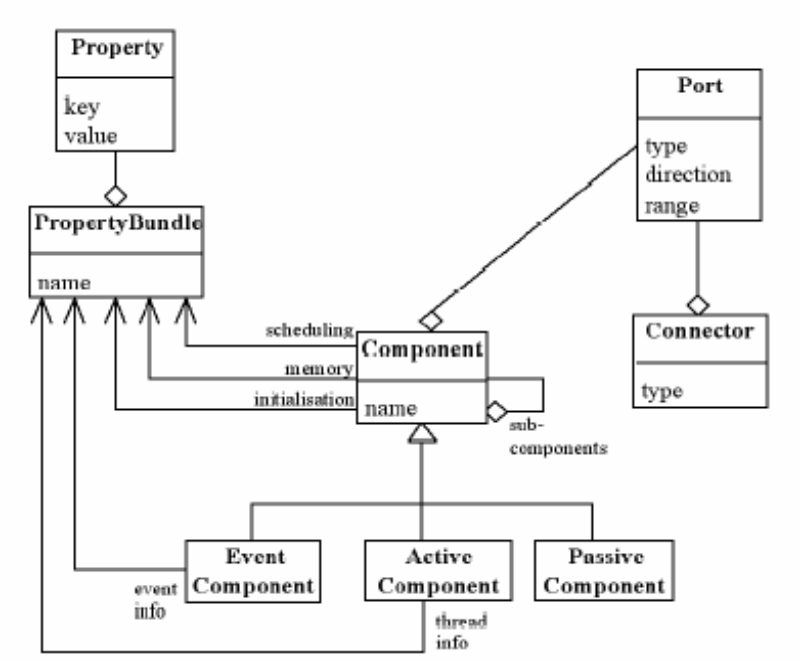


Figure 4.9, UML diagram showing the structure of a component

Component Interface

The interface is as indicated port based. A port is a shared variable that allows a component to communicate with other components. Connected ports represent the same shared variable. A port specifies a name, a type, a range (minimum and maximum), and a direction (in, out or inout). The direction indicates if the component reads, writes or does both on a specific port.

Ports can only be connected if they have the same type, and their directions are complementary. Thus, an out port can only connect to an in port. Internal ports (within a composite component) however, can be connected to an external port with the same direction, e.g., an external in port can be connected to an internal in port.

Component Composition

Components are categorized into leaf components and composite components. A leaf component is a “black box” component not further defined by the model, but rather directly implemented in the host programming language.

A composite component on the other hand contains a number of sub-components connected with the internal ports. The external ports of the composite component are connected to suitable internal ports. The sub-components are not visible outside the composite component. The software can be modelled as a component hierarchy, i.e., a tree of components with a single active composite component at its root.

In figure 4.10, a composite component is built from a couple of sub components. The sub component FQD is an event component while ModBus is an active component, the component ProcessApplication on the other hand is a passive component. There are a set of internal ports between the components, and one external port setPoint.

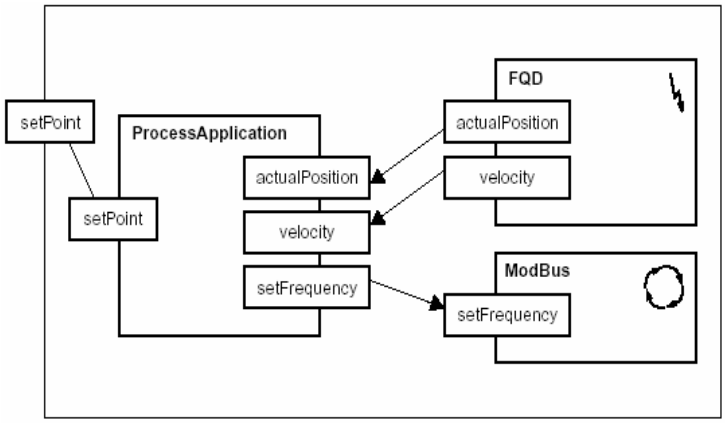


Figure 4.10, A composite component

Component Framework

There is no special run-time environment developed in the PECOS project. Instead there are requirements on platform independence, or at least portability. In [MÜL01] it is claimed that source level portability is sufficient. This requires some agreement on the language (e.g. ANSI C or C++). The run-time environment used in this case is any arbitrary RTOS.

4.4.6 IEC 61131-3

Because of the prior problems with the lack of standards for PLCs (Programmable Logic Controllers), IEC instituted a standard in 1993. The name of the standard is “IEC 61131: Programmable Controllers” [IEC92], and part 3 of the document refers to programming languages. At that time several well established techniques for programming PLCs existed, so the authors of the standard found it necessary to include several different programming methods. The standard describes three graphical and two text based languages, it concentrates on the syntax and leave the semantics less definitive.

In figure 4.11, the principles of the standard are visualized. The standard covers the construction of whole systems. At the top level, the configuration defines the entire system. Within a configuration one or more resources are defined. A resource is a processing unit, which contains one or more Tasks. Tasks in turn are defined by any of the programming languages included in the standard.

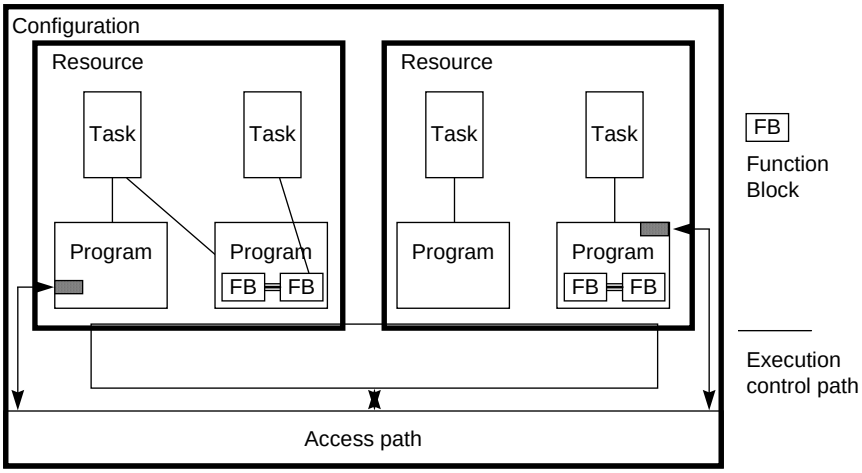


Figure 4.11, a graphical view of the elements covered in the IEC 61131 standard

Component model

One of the graphical languages included in the standard can be called a component language, the Function Block Diagram (FBD) language. It is a graphical language that can be used to define applications in terms of control blocks, which also can be imagined as components. As a short orientation the two other graphical methods included are ladder diagrams and sequential function charts, and the remaining two textual languages are called instruction lists and structured text.

Component definition

In the IEC 61131-3 FBD approach, an application is divided into a number of blocks. Each functional block has a set of in and out ports, and a hidden internal implementation. Each function block can contain both data and an implementation as a function. Furthermore, IEC 61131-3 requires strong data typing. The intention with the Function Blocks is that they should be equivalent to integrated circuits (IC), representing a specialized control function. An IC is a component in the hardware world, and in this survey we treat it as a simple software component also.

Component interface

The external interface of a component (function block) is ports. The interface is then purely functional. Ports are either in or out ports, and have a type and a name. The standard specifies 20 elementary data types, as for instance 7 variants of the integer type with variation points as signed, unsigned, short and long. It is also possible to include derived data types and arrays of any type.

Component composition

When using the graphical FBD approach, composing a system is a matter of connecting components, similar to electronic circuit diagram designs. Components can also be composites of sub components, a picture derived from the standard visualizing a composed component declaration with external interface and body is shown in figure 4.12. The figure consists of two parts the external interface definition in the upper half and the body with interconnected sub components in the lower part. Notice that ports inside the component in the upper part of the figure has the same name as the boundary ports in the lower part of the figure, when specifying the internal structure.

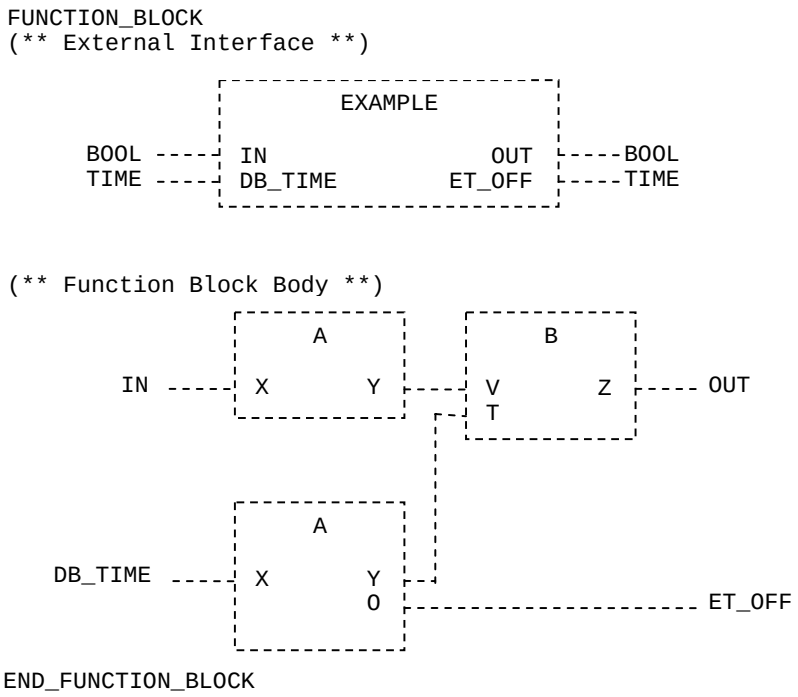


Figure 4.12, graphical function block declaration

Component framework

There is no special construction that can be treated as a component framework defined by the standard, since components are mapped to ordinary tasks before runtime. It is the operating system that is responsible for task execution, it is not much mentioned about the operating system but part 1 of the standard (IEC 61131-1, with general information) defines the some restrictions like no pre-emption of tasks.

4.4.7 CORBA technologies

The *Common Object Request Broker Architecture* (CORBA) is a standard that provides a set of rules for writing platform independent applications. The CORBA standard is developed by the *Object Management Group* (OMG). CORBA permits the developer to hide much of the low-level complexity, and offers a platform independent interface. However, a major drawback is that CORBA implementations often results in both bulky and computation intensive systems, and therefore too large to fit devices with limited resources.

In order to be able to use CORBA within smaller more resource constrained systems OMG has suggested a subset of CORBA, called minimum CORBA. Minimum CORBA omits many of the resource intensive features that are not typically essential to a basic CORBA implementation. Most dynamic features have been omitted from the minimum CORBA standard since resource constrained systems tend to make commitments at design-time rather than at run-time. For instance all dynamic features, such as *dynamic invocation interface, dynamic skeleton interface etc.,* have been omitted from the minimum CORBA standard. However, for full compatibility with the CORBA standard, the minimum CORBA supports full IDL (*Interface Definition Language)*.

Another approach taken by OMG to deal with the expression and enforcement of real-time constraints on end-to-end execution is the *Real-Time CORBA*. In Real-Time CORBA the ORB is extended with real-time capabilities. Examples of Real-Time CORBA implementations are TAO [CORBA1] and NRaD/URI [CORBA2].

Whilst *minimum CORBA* is a variant of CORBA with a lot of functionality omitted, the *Real-Time CORBA* is a set of CORBA add-ons to be able to handle clock synchronization, bounded execution times etc. It could still be stated that Real-Time CORBA is not really suited for resource constrained systems with limited calculation capabilities and huge requirement of memory resources.

Component model

It is difficult to define a component model within the scope of CORBA since applications do not have to follow any given model. However, the *Interface Definition Language* (IDL) could be seen as a loose component model. The IDL defines one way only of access of an object, at least between the client and server side objects. CORBA also makes use of an interface repository, from where interfaces can be invoked both dynamically during run-time and statically during compile-time. In minimum CORBA the interface repository is omitted, as is the dynamic invocations [CORBA3].

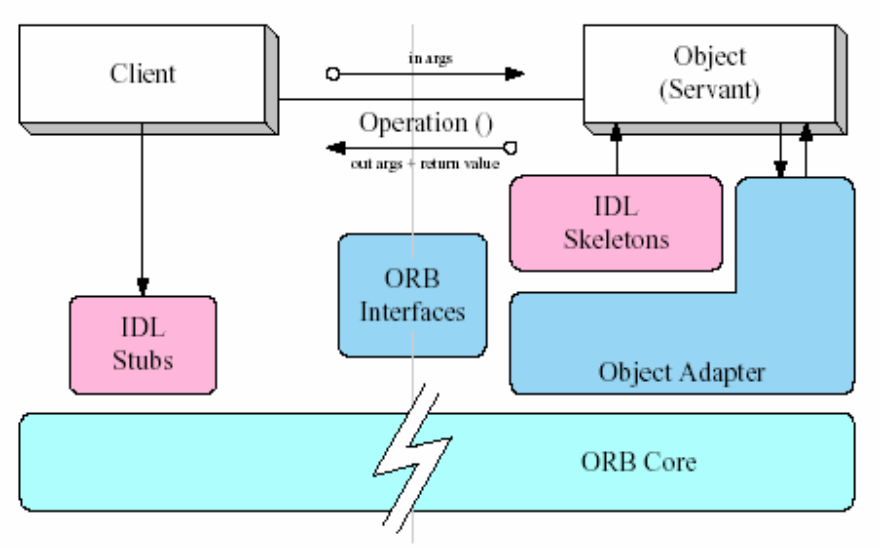


Figure 4.13, CORBA Architecture diagram

Component definition

According to the given model of a component, it is defined by the *Interface Definition Language*. IDL is a C++ like language that defines the interfaces related to target objects. It is not a programming language like C++ or Java in the sense that objects and applications cannot be implemented in IDL. The latter merely allows object interfaces to be defined in a fashion that is independent of any particular programming language. This is vital to the CORBA goal of supporting heterogeneous systems and integrating separately developed applications. IDL has a set of language mappings which defines how the IDL object interface definitions are translated into the different programming languages supported by CORBA standards. Implementation languages currently supported by CORBA are C, C++, Smalltalk, Ada and Java.

Component Interface

Interfaces are stated and designed by the user at design-time. These are the interfaces between the server and client objects. There are several levels of interfaces in CORBA. There are actual interfaces and conceptual interfaces. The actual interfaces are between the application and the ORB. However, these interfaces are automatically generated with tools and the *Interface Definition Language* (IDL).

Component Composition

Because the components reside on different nodes, client-side and server-side, traditional composition may not be possible. However, on a higher level, the composition of these components could be precedence at design time.

Component frameworks

The run-time framework for CORBA is the *Object Request Broker* (ORB), which is a middleware technology that manages communication and data exchange between objects. The ORB promotes interoperability of distributed object systems because it enables users to build systems by connecting objects from different vendors that communicate via the ORB. The ORB locates a server object implementation during run-time and establishes connections to the server and interprets object references.

4.4.8 VEST

VEST [STA01] is an aspect-based real-time embedded system composition tool. The main contributions of the tool are the ability to provide a compositional framework for developing component-based distributed embedded and real-time systems. VEST is based on the GME (Generic Modelling Environment), hence, applying the model-based development approach (discussed in chapter 3) to component-based real-time and embedded systems. As VEST enriches the CORBA component model with real-time aspects that are considered during the system composition, it can be viewed as a compositional extension of the CORBA component model and component framework to distributed embedded and real-time systems.

Component model

VEST adopts the component model of CORBA in development of component-based embedded and real-time systems. Although main features of the CORBA component model are preserved in VEST, the model itself is extended with, so-called real-time aspects. Namely, for each component from a component library that is used for assembling the overall system, a set of properties relevant to real-time and embedded environments is defined. These properties are denoted aspects to indicate their crosscutting nature and are classified as follows:

- *Perspective aspects* model the component run-time properties, e.g., WCET, memory consumption, and CPU consumption.
- *Aspect checks* provide support for dependency checks among components. Aspect checks provide compositional rules for components as they allow checking assembly of components with respect to its eligibility for use in a certain embedded and real-time platform. The information from perspective aspects e.g., WCETs, are used by aspect checks to determine, for example, if the components in assembly are schedulable.

Component interfaces

The interfaces of components on VEST are described using extended CORBA IDL. Namely, as CORBA components are used as underlying components in the development and they are enriched with additional component information that is essential for correct assemblies in real-time environments, the interfaces in VEST could be viewed as an extension of CORBA IDL. For example, perspective aspects describe the run-time properties of components and are used without access to component internals to support correct system assemblies. Hence, they can be treated as run-time interfaces of CORBA components. Moreover, aspect checks can be viewed as real-time compositional rules for assemblies enriching the interface definitions with compositional rules for real-time system assembly.

Component framework

In VEST, the broader framework to the one used in CORBA is presented, as the frameworks both relies on ORBs and the compositional rules that are checked during the system composition before system becomes operational, i.e., at compile time.

4.5 The Needs and Priorities in Research

If you want to get automatically proper numbering of sections (so that in Chapter 2 all the sections starts with 2.) do the following: click on Introduction and select Format→styles and Formatting, select the "1 Heading 1" paragraph, select Modify, Format, Numbering, Customize, change Start to the appropriate number (for example 2), and than click Ok. Maybe there is a simple way, but I co not know it.

Major needs for the further development of component technology for embedded systems are the following [BRI02].

- Need for adopted component models and frameworks for embedded systems. A problem is that many application domains have application-dependent requirements on such a technology.
- Need for light-weight implementations of component frameworks. In order to support more advanced features in component-based systems, the run-time platform must provide certain services, which however must use only limited resources.
- Uniform Specification of component properties: Current specification techniques for contracts use notations and models that are quite different. It is very desirable to achieve unification and uniformization of such notations.
- Obtaining extra-functional properties of components: Timing and performance properties are usually obtained from components by measurement, usually by means of simulation. Problems with this approach are that the results depend crucially on the environment (model) used for the measurements may not be valid in other environments, and that the results may depend on factors which cannot easily be controlled. Techniques should be developed for overcoming these problems, thereby obtaining more reliable specifications of component properties.
- Platform and vendor independence: Many current component technologies are rather tightly bound to a particular platform (either run-time platform or design platform). This means that components only make sense in the context of a particular platform.
- Efforts to predict system properties: The analysis of many global properties from component properties is hindered by inherent complexity issues. Efforts should be directed to finding techniques for coping with this complexity.

- Component certification: In order to transfer components across organizations, techniques and procedures should be developed for ensuring the trustworthiness of components.
- Component noninterference: Particularly in safety-critical applications, there is a need to ensure separation and protection between component implementations, in terms of memory protection, resource usage, etc.
- Tool support: The adoption of component technology depends on the development of tool support.

The clearly identified priorities of CBSE for embedded systems are:

- Predicting system properties. A research challenge today is to predict system properties from the component properties. This is interesting for system integration, to achieve predictability.
- Development of widely adopted component models for real-time systems. Such a model should be supported by technology for generating necessary runtime infrastructure (which must be light-weight), generation of monitors to check conformance with contracts, etc. The trend towards open integrated systems implies that it should be possible for a system to use both a component model specific for real-time systems, and some of the widely used component technologies.

4.6 Possible Direction for Composition Technologies

In this section we give a wider view, as an overview of the software engineering techniques primarily focusing on system composition. Figure 4.15 provides hierarchical classification of composition-oriented approaches [Ass02].

Research in the component-based software engineering community increasingly emphasizes composition of the system as the way to enable development of reliable systems, and the way to improve reuse of components.

Component-based systems on the first level, e.g., CORBA, COM and JavaBeans, represent the first generation of component-based systems, and are referred to as ``classical" component-based systems [Ass02]. Frameworks and standards for components of today in industry primarily focus on classical component-based systems. In these systems components are black boxes and communicate through standard interfaces, providing standard services to clients, i.e., components are standardized. Standardization eases adding or exchanging of components in the software system, and improves reuse of components. However, classical component-based systems lack rules for the system composition, i.e., composition recipe.

The next level represents architecture systems, e.g., RAPIDE [LK+95] and UNICON [Zel96]. These systems provide an architectural description language (ADL), which is used to specify the architecture of the software system. In an architecture system, components encapsulate application-specific functionality and are also black boxes. Components communicate through connectors [AAG93], and a connector is a specific module that encapsulates the communication between application-specific components. This gives significant advancement in the composition compared to classical component-based systems, since communication and the architecture can be varied independently of each other. Thus, architecture systems separate three major aspects of the software system: architecture, communication, and application-specific functionality. One important benefit of an architecture system is the possibility of early system testing. Tests of the architecture can be performed with ``dummy" components leading to the system validation in the early stage of the development. This also enables the developer to reason about the software system at an abstract level.

Classical component-based systems, adopted in the industry, can be viewed as a subset of architecture systems (which are not yet adopted by the industry), as they are in fact simple architecture systems with fixed communication.

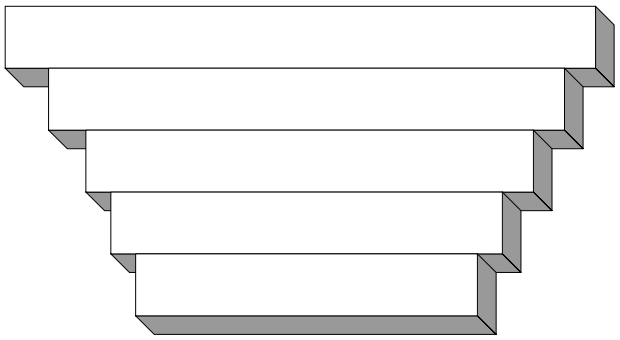


Figure 4.14, Classes of component-based systems

The third level represents aspect systems that are developed using the AOSD principles [KL+97]. Aspect systems separate more concerns of the software system than architecture systems. Beside architecture, application, and communication, aspects of the system can be separated further: representation of data, control-flow, memory management, etc. Temporal constraints can also be viewed as an aspect of the software system, implying that a real-time system could be developed using AOSD [BB99]. Only recently, several projects sponsored by DARPA (Defense Advanced Research Projects Agency) have been established with the aim to investigate possibilities of reliable composition of embedded real-time systems using AOSD [Dar01]. The projects include AIRES [AIR03], ISIS PCES [PCE03], and FACET [FAC03]. In aspect systems, aspects are separated from core components; they are recombined automatically through weaving. In AOSD, a core component is considered to be a unit of system functional decomposition, i.e., application-specific functionality [KL+97]. Weavers are special compilers that combine aspects with core components at so-called joint points either statically (at compile time) or dynamically (at run-time). Weaving breaks the core component (at joint points) and cross-cuts aspects into the component and the weaving process results in an integrated component-based system. Hence, core components are no longer black boxes; rather they are white boxes as they are cross-cut with aspects. However, aspect weavers can be viewed as black boxes since they are written for a specific combination of aspects and core components, and for each new combination of aspects and core components a new aspect weaver needs to be written. The process of writing an aspect weaver is not trivial, thus, introducing additional complexity in the development of aspect systems and usability of aspects. Compared to architecture systems, aspect systems are more general and allow separation of various additional aspects, thus, architecture systems can be viewed as a subset of the class of aspect systems. Having different aspects improves reusability since various aspects can be combined (reused) with different core components. The main drawback of aspect systems is that they are built on special languages for aspects, requiring system developers to learn these languages.

At the fourth level are systems that provide composition operators by which components can be composed. Composition operators are comparable to component-based weaver, i.e., a weaver that is no longer a black box, but is also composable out of components, and can be re-composed for every combination of aspects and components, further improving the reuse. Subject-oriented programming (SOP) [OT93], an example of systems with composition operators, provides composition operators for classes, e.g., merge (merges two views of a class), and equate (merges

two definition of classes into one). SOP is a powerful technique for compositional system development since it provides a simple set of operators for weaving aspects or views, and SOP programs support the process of system composition. However, SOP focuses on composition and does not provide a well-defined component model. Instead, SOP treats C++ classes as components.

Finally, the last level includes systems that contain a full-fledged composition language, and are called composition systems. A composition language should contain basic composition operators to compose, glue, adopt, combine, extend, and merge components. The composition language should also be tailorable, i.e., component-based, and provide support for composing (different) systems, in the large. Invasive software composition [Ass02] is one approach that aims to provide a language for the system composition, and here components may consist of a set of arbitrary program elements, and are called boxes [Ass02]. Boxes are connected to the environment through very general connection points called hooks, and can be considered grey box components. Composition of the system is encapsulated in composition operators (composers), which transform a component with hooks into the component with code. The process of system composition using composers is more general than aspect weaving and composition operators, since invasive composition allows composition operators to be collected in libraries and to be invoked by the composition programs (recipes) in a composition language. Composers can be realized in any programming or specification language. Invasive composition supports software architecture, separation of aspects, and provides composition receipts, allowing production of families of variant systems. Reuse is improved, as compared to systems in the lower levels, since composition recipes can also be reused, leading to easy reuse of components and architectures. An example of the system that supports invasive composition is COMPOST [COM01]. However, COMPOST is not suitable for systems that have limited amount of resources and enforce real-time behaviour, since it does not provide support for representing temporal properties of the software components. Also, COMPOST is language-dependent as it only supports Java source-to-source transformations.

4.7 Conclusion

Component-based models for embedded systems and real-time systems are different from those widely used in other domains. The principles and the technologies have not yet fully developed although there are several cases in industry of a successful use of component models, and there exist some promising research results. For full exploitation of component models a number of challenges must be met. These challenges are related to real-time requirements, limited resources, and difficulties of expression of extra-functional properties and modeling of their compositions. In contrast to general-purpose component models, most of the component models for embedded systems will keep configuration separated from run-time frameworks. This decreases some advantages (such as flexibility) but increases the predictability of the composed systems run-time behavior.

4.8 References

[AAG93] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. ACM Software Engineering Notes, 18(5):9--20, 1993.

[Air03] Aspects in real-time embedded systems ({AIRES}). Project web-site at http://www.dist-systems.bbn.com/projects/AIRES/, February 2003.

[Ass02] U. Assmann. Invasive Software Composition. Springer-Verlag, December 2002.

[AVI01] Avižienis A., Laprie J-C., Randell B., Fundamental Concepts of Computer System Dependability, IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable, Robots in Human Environments, 2001.

[BAC00] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, Volume II. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, May 2000

[BB99] L. Blair and G. Blair. A tool suite to support aspect-oriented specification. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99, pages 7--10, Lisbon, Portugal, June 1999.

[Bos00] J. Bosch. Design and Use of Software Architectures. ACM Press in collaboration with Addison-Wesley, 2000.

[BRI02] E. Brinksma et al., ROADMAP - Component-based Design and Integration Platforms, W1.A2.N1.Y1, Project IST-2001-34820, ARTIST - Advanced Real-Time Systems, 2003.

[CL00] I. Crnkovic and M. Larsson. A case study: Demands on component-based development. In Proceedings of 22th International Conference of Software Engineering, pages 23--31, Limerick, Ireland, June 2000. ACM.

[CLL00] I. Crnkovic, M. Larsson, and F. Luders. State of the practice: Component-based software engineering course. Proceedings of 3rd International Workshop of Component-Based Software Engineering. IEEE Computer Society, January 2000.

[CL02] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. ArtechHouse, 2002.

[COM01] Germany University of Karlsruhe. Compost. Documentation available at: http://i44w3.info.uni-karlsruhe.de/~compost/, June 2001.

[Dar01] DARPA ITO projects. Program composition for embedded systems. http://www.darpa.mil/ito/research/pces/index.html, 7 August 2001.

[DG00] K. R. Dittrich and A. Geppert. Component Database Systems, chapter Component Database Systems: Introduction, Foundations, and Overview. Morgan Kaufmann Publishers, 2000.

[FAC03] Framework for aspect composition for an event channel (FACET). Project web-site at http://www.cs.wustl.edu/~doc/RandD/PCES/, February 2003.

[Fle99] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pages 75--84, Saint-Malo, France, May 1999. IEEE Computer Society Press.

[HOA85] C. A. R. Hoare. Communicating Sequential Processes, Englewood Cliffs, Prentice Hall, 1985.

[IEC92] International Standard IEC 1131. Programmable controllers, 1992.

[KHO92] P. K. Khosla et al., The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications, IEEE Transactions on Systems, Man and Cybernetics, 1992.

[KL+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, and J. Irwin. Aspect-oriented programming. Proceedings of the ECOOP, volume 1241 of Lecture Notes in Computer Science, pages 220--242. Springer-Verlag, 1997.

[Kop03] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In Proc. 6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC), Hokkaido, Japan, May 2003.

[LC99] M. Larsson and I. Crnkovic. New challenges for configuration management. Proceedings of System Configuration Management, 9th International Symposium (SCM-9)}, volume 1675 of Lecture Notes in Computer Science, pages 232--243, Toulouse, France, August 1999. Springer-Verlag.

[LAY73] J. Layland and C Liu. Scheduling algorithms for multiprogramming in hard real-time environments, Journal of the ACM, January 1973.

[LK+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering}, 21(4):336--355, April 1995. Special Issue on Software Architecture.

[MsC01] Microsoft. The component object model specification. Available at: http://www.microsoft.com/com/resources/comdocs.asp, February 2001.

[MB+99] A. Munnich, M. Birkhold, G. Farber, and P. Woitschach. Towards an architecture for reactive systems using an active real-time database and standardized components. Proceedings of International Database Engineering and Application Symposium (IDEAS)}, pages 351--359, Montreal, Canada, August 1999. IEEE Computer Society Press.

[MÜL01] P. Müller, C. Zeidler, C. Stich, A. Stelter. PECOS — Pervasive Component Systems, Workshop on Open Source Technologie in der Automatisierungstechnik, GMA Kongress 2001.

[NIE02] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, R. van den Born, *A Component Model for Field Devices* Proceedings of the First International IFIP/ACM Working Conference on Component Deployment, Germany, June 2002.

[OMG01] OMG. The common object request broker: Architecture and specification. OMG Formal Documatation (formal/01-02-10), February 2001. Available at: ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf.

[OMM00] R. van Ommering, F. van der Linden, and J. Kramer. The Koala component model for consumer electronics software. IEEE Computer, 33(3):78–85, March 2000.

[OT93] H. Ossher and P. Tarr. Subject-oriented programming: a critique of pure objects. Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications, pages 411--428, Washington, USA, September 26 - October 1 1993. ACM Press.

[PCE03] Constraint-based embedded program composition. Project web-site at http://www.isis.vanderbilt.edu/Projects/PCES/default.html, February 2003.

[SAN98] K. Sandström, C. Norström, G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System, In Proceedings of the 5th International Conferance on Real-Time Computing Systems and Applications, October 1998.

[STA01] J. A. Stankovic. VEST A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2001.

[STE97] D. B. Stewart, R. A. Volpe, P. K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, IEEE Transactions on Software Engineering, December 1997, pages 759-776.

[SZY98] C. Szyperski. Component Software: Beyond Object-Oriented Programming. ACM, Press and Addison-Wesley, New York, N.Y., 1998.

[SZY02] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Second edition, ACM, Press and Addison-Wesley, New York, N.Y., 2002.

[WAL03] K. C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components, Technical report, Software Engineering Institute, Carnegie Mellon University, April 2003, Pittsburgh, USA.

[Zel96] G. Zelesnik. The UniCon Language User Manual. Carnegie Mellon University, Pittsburgh, USA, May 1996. Available at http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/.

Chapter 5

Safety-critical and dependability aspects

Simin Nadjm-Tehrani and Henrik Thane

1.1 Introduction

Although a consumer-driven development of today's systems necessitates evaluating systems from a range of perspectives, e.g. usability or cost-effectiveness, it remains a matter fact that dependability is a major criteria for evaluating vehicular systems. The public and the legislators demand reliable and safe computer control systems, equal to or better than the mechanical or electromechanical parts they replace.

1.2 State-of -practice

The aerospace industry is certainly the most advanced sector with respect to safety process maturity among the vehicular systems. Public interest in enhancing safety has been a leading factor in defining standards, best practices, post-accident studies, and regulatory measures over almost a century. In comparison to development of cars, for example, a new aircraft under development cannot even be rolled out on a runway without strict permissions and safety-related procedures prior to that. The very moving on a runway is considered as a stage of flight and is precisely dependent on measures taken for showing airworthiness of an aircraft. That should be compared to many real tests performed on pilot models of cars in restricted test areas.

While general safety-related standards and procedures are actively adopted in all branches of transportation industry, including trains and automotives, in this chapter we provide a more detailed account of the development of the airborne systems, as they provide the most complex and publicly regulated example for vehicular systems.

1.2.1 The notion of component

So what is the notion of a component in an aircraft system? A review of the possible components in the Saab Aerospace division, for example includes all of the following instances:

1. A complete and self-contained subsystem (Swedish: Nykelfärdigt system). Examples are a ventilation system or a fuel system. Here, the supplier provides the mechanic/hydraulic hardware, the electronic hardware, and the application software, all tested and delivered as an integrated component.
2. A device or component that is not completely ready to install in an aircraft, but which is only lacking the application software. Examples of such systems are certain actuators and sensors. They include the digital hardware and interfaces (system software) on top of which the desired functionality is added.
3. A component that a customer provides in order to be built-in in the final product that the airborne system comprises. This type of unit that is delivered by the customer might in turn be embedded in a subsystem developed by a third party while the responsibility of the safety subsystem as well as the integrity of the imported unit is with the developer of the embedding product. Examples of such units are cryptographic units, certain pilot equipments (e.g. helmets, data transfer units - Swedish: datastavar).
4. A Ground support system that is supplied by the customer, and should be integrated with the complete airborne system. These are per se not part of the flight system but are an

integrated part of the safety system, typically used for pre-flight checking, and for maintenance procedures.

5. Pure COTS. A piece of software that is used in integration with other developed software – typically not in the safety-critical parts of an aircraft. One example that in fact is used in the safety-critical context would be a certified Ada runtime kernel, delivered by a third party e.g. Rational, Aonix, GreenHill.
6. Software code or electronics that is outsourced to other developers. Typical example is Field Programmable Gate Arrays (FPGA) implementing control logic.
7. Software code that is developed in-house, and later integrated into the complete system.
8. Hardware (other than digital) components that are either produced in-house or purchased from third parties. These components are considered least relevant from the point of view of the goals of the study that this book addresses. Some example are body elements, pumps, pipes etc.

The most complex subsystem in an aircraft, that in fact has a special status in that its safety is not subject to assurance at the aircraft manufacturer, is the engine. It is the engine supplier that has the full responsibility for engine function contribution to aircraft safety and certification.

The difference between the ways the above types of components are treated by an aircraft vendor is reflected in the steps that are taken in procurement, requirements specification, verification and safety evaluation.

A change of supplier for a product within category 1 to 5 renders the component a 'new' component. All the steps that are taken to ensure that the product does not create a threat to safety have to be repeated. In some restricted cases 'qualified similarity' is used as an argument to perform fewer tests or repeat only a relevant portion of the tests, but these decisions are highly qualified decision taken on the basis of experience and to the extent allowed by mandatory regulations. For components in category 5 one must in addition ensure that any functions that exist in the delivery, but which were not part of the ordered set of requirements, are not detrimental to safety and do not affect the integrity of the rest of the system. In components of type 6 there are two ways to ensure that component function and implications for safety are assured:

- Detailed, precise and complete documentation of requirements at the start of the procurements stage.
- Review of the vendors capabilities, followed processes, adherence to required standards such as DO 178B or DO 254, approval and evaluation procedures, track record.

Certain components are never bought-in as independent units (Swedish: nyckelfärdiga) without extensive qualification data. Examples are complete weapon systems or flight control system.

1.2.2 Safety and functional verification process

The next interesting question is how to ensure that a *change* in a subsystem or component should be reflected in the tests performed to ensure functional correctness and system safety? If a small change is imposed in the design or coding of a software component, does it mean that *all* the earlier verifications have to be repeated? The answer is: in principle, yes!

To avoid the costly effect of this procedure within system development, one utilises the following possibilities:

- The performed tests are automated as much as possible.
- The dependency relationship between the old function and the rest of the system is compared with the same relationship between the changed version and the rest of the system. Estimations of these dependencies are used to repeat a only a restricted subset of the old tests where possible.
- No matter how small the change was, a minimum set of 'essential' tests are repeated. Examples are scenarios covering safe landing or so-called safe-return-to-base even the imposed change had nothing to do with the landing and related systems.

The above tests cover both functional and safety-related evaluation of a system. As well as functional requirements, those requirements that enhance safety, e.g. replication to promote fault tolerance are clearly specified in a requirements specification for a component, and any subsequent change to the component must be matched against the earlier safety-related requirements (e.g. "this component must provide two separate output signals, the values of which should not differ by more than … in the non-faulty scenarios").

1.2.3 Common source of modifications

A component that is developed and plugged into a system's architecture may found to be less than adequate at some testing/assurance stage. It would be interesting to find out which are the circumstances that most often lead to design/code modifications. Or rather, what are the typical errors found in development of new airborne systems?

According to aerospace experts most errors are found in the interface between components; either because the original specification was incomplete (had forgotten to specify some aspect), or that it simply made wrong assumptions. A typical case is that one forgets the dependencies between several components, and when one component is updated/changed, the potential changes in other subsystems are not fully considered, or corresponding changes introduced there. An example is an attribute such as measured wheel velocity. If one reduces the number of pulses per rotation the resolution of the measurement is decreased. This might be favourable in terms of costs in the landing gear system, but the change might affect other consumers of the information. So the supplier of the landing gear system may move on to a cheaper realisation not considering the changes implied in the flight control system or pilot information system. The way such changes are propagated in the system are by administrative processes: meetings, agreements, reviews.

1.2.4 Infrastructure components

An interesting issue is whether there are inherent differences between application components and infrastructure components. By infrastructure components we mean standard units such as operating systems, communication systems (buses) etc. This question is only relevant if there is extensive use of infrastructure components in an airborne system. In the type of aircraft we are considering, however, there are very few such units. Even if one buys a communication bus one does not buy the schedule that runs on it, which constitutes the real content of the bus. So the content is to be verified as most other components, and there seems to be no reason for special treatment. Implicitly, one not consider the risk that there are errors in the bus protocol implementations as high, and therefore safety assurance for such units is not a major issue. Other purchased units such as transponders are standardised and simply imported into the assembly, but they are not safety-critical.

1.3 Dependability terminology

Dependability of computer systems, being hardware or software intensive, has been treated extensively in the research community in the last two decades. The notions of dependability have been clarified by the International Federation for Information Processing (IFIP) working group 10.4 (WG 10.4) in terms of a number of major attributes in a series of documents in early 90's – though some of the basic notions have existed since late 60's.
WG 10.4 defines **dependability** as the property of a computer system such that reliance can justifiably be placed on the services it delivers [IFIP]. The *service* delivered by a system is its behaviour as perceived by its user(s); a *user* is another system (physical, human) that interacts with the former. Dependability can be viewed according to different complementary properties that enable the *attributes* of dependability to be defined as follows:

- the readiness for usage leads to **availability**
- the continuity of service leads to **reliability**
- the non-occurrence of catastrophic consequences on the environment leads to **safety**
- the non-occurrence of unauthorized disclosure of information leads to **confidentiality**
- the non-occurrence of improper alterations of information leads to **integrity**
- the aptitude to undergo repairs and evolutions leads to **maintainability**

Associating integrity and availability with respect with respect to authorized actions together with confidentiality, leads to **security**.

A system **failure** occurs when the delivered service deviates from fulfilling the system function, i.e. what the system is intended for. An **error** is that part of the system state that may potentially lead to subsequent failure. A **fault** is what is judged or hypothesized to be the cause of an error. For example, the (latent) existence of year 2000 bugs in software can be considered as faults that could have caused errors in presence of certain conditions (the end of millenium approaching and the system running as before), and could have caused failures in many systems.

Abstractions or designs that do not execute, but have states like models, diagrams, programs, etc., do not fail, but can be erroneous. Failures occur when the designs are realized into concrete machines and the machines operated.

Software does not fail; it is a design for a machine, not a machine or physical object. The computer system can however fail, either due to failures in the hardware or due to errors (bugs) in the design (program) when it is executed. The software-related error typically manifests within a given execution sequence inside the computer system (see figure 1).

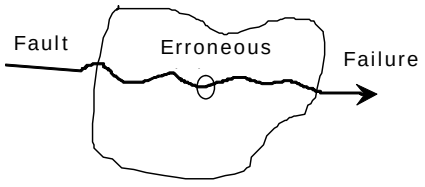


Figure 5.1 *An execution thread passes through an error that results in a failure.*

The development of a dependable computing system requires methods for fault prevention, fault tolerance and removal (detection, correction or containment). This can be done with emphasis on one or more of the above dependability attributes.

In this part of the book we concentrate on attributes of dependability that are essential for safety-critical systems, not the least for certification purposes. Although different applications might put a bigger emphasis on some attribute, all vehicular systems have strict *reliability* and *safety* concerns. Thus, we focus our attention on these two aspects and the interplay between satisfying functional requirements that can in principle be traced down to a component level, and safety-reliability requirements that are system-wide properties and pose special problems when developing systems with components. In both cases we reuse the definitions according to those adopted by the IFIP WG 10.4 on dependability.

Availability is usually quantified by *1- MTTR/MTTF*, where *MTTR* is the mean time to repair the system and *MTTF* the mean time to failure. The scope of this book chapter does not allow an in depth review of this related and important area. The interesting reader is referred to a recent survey on the topic for replication-based approaches to achieve high availability [NS03, SN03].

1.4 System reliability and software reliability

Seminal work on hardware reliability shows that reliability of a system is the product of the reliability of its components [Ekr03]. However, the topic of reliability for software is a wide-open area for research.

Hardware reliability (e.g. for mechanical components) can be defined as [Oco95]:

The probability that an item will perform a required function without failure under stated conditions for a stated period of time. *O'Connor(1995)*

Leveson [Lev95] adopts a similar line when considering reliability of software components. However, the interaction complexities of software make the compositional reliability of software harder to quantify. Reliability is often quantified by MTTF – Mean Time To Failure. Say, 115 000 years or as a frequency 10^{-9} failures/hour.

Typical factors that affect hardware reliability are wear and tear, external environment conditions and design errors that show up over continuous use. Software is different in the sense that it does not change with time, and its original properties persist over its lifetime (unless affected by upgrades). Thus, it may seem that software reliability is a binary attribute; either faults exist and will eventually show up, or they do not exist (or never show up). A definition of software reliability that is in line with the above definition is [Lyu95]:

The probability of failure-free software operation for a specific period of time in a specified environment.

Lyu (1995)

According to this definition, evaluation of software reliability is only possible if its tendency to fail is measured over a range of inputs that represent "the environment". However, combinatorial explosion makes the systematic analysis such scenarios very

difficult. While there have been successful steps towards improving techniques and processes that lead to software reliability measurement and assurance, there is no consensus among researchers around a general model for software reliability.

A widespread myth about software is that the smaller the software unit the higher its reliability is. A study by Jones shows for example that the number of delayed and cancelled projects dramatically increases as the number of function points (a measure of size) increases beyond 5000 [Jon95]. At this level of complexity (roughly corresponding to 500k lines of code in languages like Fortran) 79% of the projects are cancelled or delayed by over 6 months. Thus, the ability to deliver a software intensive product that satisfies the specification efficiently is obviously a major problem.

A valid question is therefore: can we increase the reliability of a system by breaking it down to small manageable components? A follow-up question being: if we have demonstrated/estimated the reliability for a component how can we derive the reliability for the whole system based on a composition of the reliability measures for the parts. There are some initial attempts for answering these questions based on historical studies of modular designs. Hatton shows for example that size-complexity-fault frequency relation is not linear and there are some medium sized components that exhibit higher reliability compared to both smaller and larger components [Hat97]. With regard to aggregation at system level, Hamlet et.al. propose a theory for compositional calculation of reliability metrics based on component metrics. Nevertheless, they contend that the theory needs to be validated in experimental settings [Ham01].

To the practitioner's aid, there exist recommended practice guidelines, such as the IEEE PI1633\AIAA R-013A on Software Reliability that has been applied to the Space Shuttle avionics software.

1.5 Functions: man or machine?

Even if the main goal of reliability evaluations is considered to be adherence to functional requirements, the "discrete" nature of changes to the state captured in software results in an insurmountable task. Moreover, building software systems from imported components does not make the task easier. We believe that for safety-critical systems the analysis of software components alone is not sufficient. Heimdahl et.al. cite that 20-35% of safety-related errors in two spacecrafts were caused by incorrect/insufficient specification of the interface between the software and its environment. Misconceptions about the operating environments, how the hardware operates, and failure to detect and respond to inputs that were outside the normal operating range were among the major causes [HTC98]. This leads to the need for extending the component interface descriptions to include aspects that are outside the traditional scope of software; at least for components that are in direct interaction with humans and the physical environment.

To the list of difficulties one can add that system (and software) do not operate in a vaccum and are often dependent on actions of human operators for successful delivery of preconceived outcomes. Thus, treatment of safety without considering human actions is seldom possible. There are several approaches on how to characterize and evaluate the effects of human action on behaviour of computer-based systems [Hol03].

In the *human factors* school, one concentrates on how humans contribute to failure of systems. One distinguishes error of omission (failure to perform a required operation) from error of commission (actions wrongly performed), and extraneous errors (wrong act is performed). This is an end result oriented approach that has little emphasis on why the operator acted in one way or the other.

In the *human information processing* school one characterizes the ability of an operator to act on a machine in terms of a number of layers that model the operator's ability to react to environment signals. Using the quantitative models for this approach one attempts to predict the likelihood that certain erroneous actions will take place.

In the *cognitive systems engineering* school considers the behaviours to emerge from a *joint* cognitive system rather than separate human and technology agents. It emphasizes the modelling of humans and machines on a par, and their interaction in enough detail that allows analysing what could go wrong in which contexts.

1.6 System safety

Safety is the ability of a system to avoid harm to people and environment. Hence, a car that never starts may seem to be safe by definition (although not quite reliable!). However, a car that does not start can also pose a threat to safety if it happens to stall on a railway crossing. In both cases the car fails. In the latter case it creates a potential threat to safety. Thus, safety is a property that very much emerges from the behaviour of the system under design and the conditions in its environment. The analysis is not made any easier considering the fact that many systems are designed to harm people or environment under certain conditions[1].

The above examples should be sufficient to justify the fact that software (or digital systems in general) cannot be considered to be safe – neither can they be considered unsafe. Software and digital hardware are typically embedded in a mechanical environment that operates together with humans, other components and so on. Since there is no way that software on its own harms people or environment, it is incoherent to allocate attributes such as safety to a piece of software or digital hardware. Software or digital hardware can only be examined in terms of the ways they may contribute towards appearance of hazard. *Hazards* are failures that may potentially lead to violation of safety. Hence, traditional analysis of system safety typically starts by considering the potential unsafe scenarios, characterizing the risks for the hazard to take place (both in terms of probability and in terms of severity of consequences), and make a quantified decision on which scenario to consider as one that should never happen – no matter how the constituent components in the system are designed developed or operated.

Hence, traditional analysis of system safety rests on techniques that focus specifically on "things that may go wrong". Fault-tree analysis (FTA) and Failure modes and events analysis (FMEA) are old techniques that grew within the era of building systems from hardware (mainly mechanical) components. One can contrast FTA and FMEA by considering one a top-down and the other as bottom-up. In other words, in FTA analysis one is interested to know given a potential failure in the system (a top level event) what are the combinations of conditions that necessarily cause that event. In FMEA, however, one tries to consider each and every constituent of the system, and trace the effects of errors manifesting in that constituents. Defining the

[1] Voas exemplifies a weapon system that is designed to destroy, and as such is reliable when it does so. The same system is however, considered to be unsafe if it misses the foe, does a U-turn and destroys the submarine which launched it [Voa95].

boundaries of the system (what is an important constituent, is the operator part of the system or not, etc) is a major problem when deciding how much time and effort should be spent on FMEA. We now go on to describe FTA in more detail.

1.6.1 Fault tree analysis

As mentioned above, fault tree analysis is primarily a method for finding causes of hazards, not identifying hazards. FTA tries to reduce the number of behaviours that need to be considered in order to get assurance that the design is safe. FTA is an analytical method using proof by contradiction. An undesirable state is specified – a hazard. The system is then analysed, in a top-down manner, with the objective to find all possible paths that can result in the undesirable state. Boolean logic is used to describe how combinations of single faults can lead to a hazard. It is like detective work in the spirit of Sherlock Holmes. If the deduction cannot find any causes the assumed hazard cannot occur. If causes are identified then preventive actions can be taken.

If a quantitative measure is desired, the probability for the top event – the hazard can be computed out of the Boolean expression represented by the fault tree. All atomic root causes must then be assigned a probability of occurrence. This is however, a questionable pursuit since the probability of occurrence of some causes cannot be quantified. For example, how can design errors in software be quantified? Program errors remain until removed, no design errors appear or disappear during runtime.
The use of FTA is better suited when applied qualitatively. As one can imagine, a fault tree may end up with a number of leaves any of which (in presence of other conditions) could lead to occurrence of the top event. In traditional electrical/mechanical systems the leaf typically classifies a physical condition in the component. If the leaf is a program component then it is sometimes possible to characterize the exact misbehaviour of the program that causes the hazardous event. Then, in fact, formal verification can in some cases be used to eliminate the risk that the particular condition does not appear in the design of software. The problem will, however, reduce to the standard problem of combinatorial complexity. If a potential cause of a hazard has to be eliminated it has to be eliminated in all potential states of a software component (or a digital implementation in hardware). Another limitation of FTA analysis is that one can only analyse and eliminate the causes for those top-level hazards that are explicitly considers. If the system safety analyser misses a potential major failure, then the causes for the failure have no chance of being tolerated or eliminated.

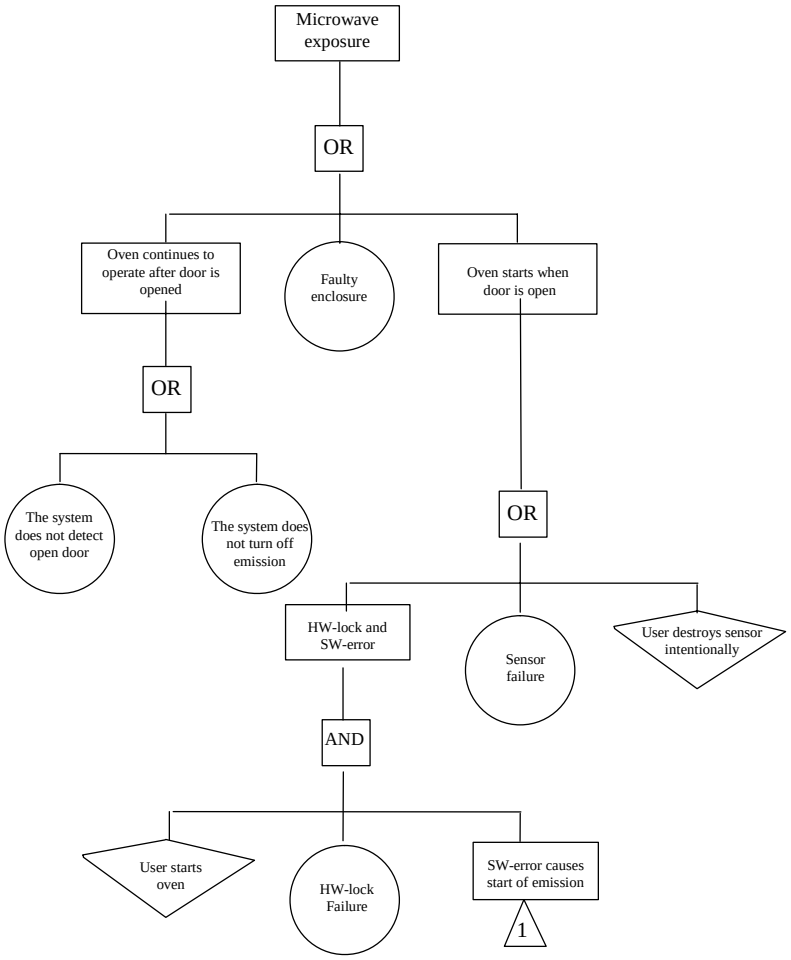


Figure 5.2 *A fault tree for the hazardous condition: microwave exposure [Gow94].*

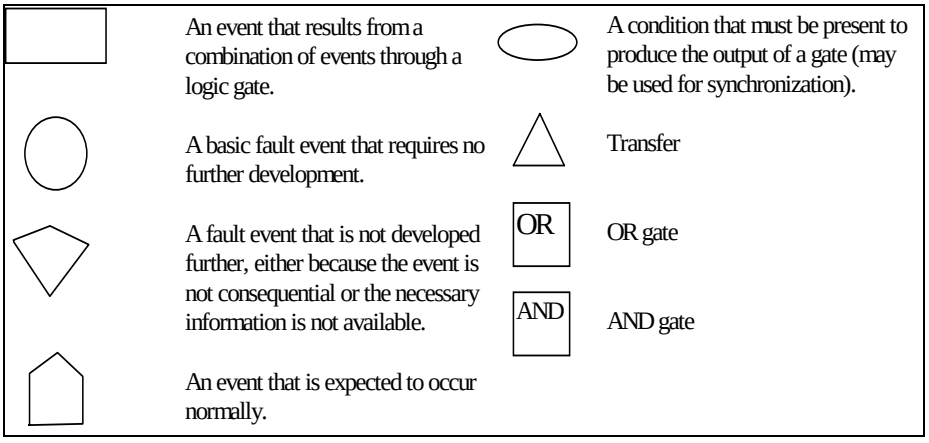


Figure 5.3 *Fault tree notation symbols.[Lev95, p. 319]*

The above figure shows an example of a fault tree (Figure 2). The hazardous state is assumed to be microwave exposure by a microwave oven. Figure 3 shows the notation used.

1.6.2 Formal analysis of faulty behaviour

Current multi-domain systems need to adapt FTA-like and FMEA-like techniques so that it is possible to pinpoint critical fault combinations and at the same time consider the multiplicity of faults that affect digital units (software and hardware). Recent years have seen steps in that direction by a combination of formal verification techniques and FTA/FMEA like reasoning [ÅNS99, BVÅ03, HN03]. These reports show applications of enhanced methods for safety and formal design verification to case studies taken from the secondary power system of the Eurofighter Typhoon aircraft, or hydraulic and air control subsystems of the JAS 39 Gripen aircraft. In what follows we briefly describe the ideas behind these enhancements.

Current industrial practice treats the design and development of systems from components, and the analysis of system safety as two separate processes, often performed in parallel and by two different teams of engineers. In the one process, if the state of the art methods are used, model-driven development is used for high-level design specification followed by code generation and verification/validation. In the other process, separate safety-related models of the same system are built in order to perform hazard and FTA/FMEA. The separation of these processes and the continuous update and rechecking of the analyses during an emerging product design, makes the overall development costly and the coherence of the various system models questionable.

Recent work allows combination of mathematical models for analysis of functional properties as well as determining whether the combinations of faults in a component and its environment can lead to major system level failures [HN03, BCC03]. In this way, the same design model can be reused both for analysis of functional properties and adherence to requirements, as well as FTA/FMEA-like analysis. All one needs to do is to extend the component models with fault modes for components, and make mathematical deductions as to the possibility of violation of a safety requirement, given potential combinations of fault modes. So far the notion of a component has been treated light-weight manner in these works. More accurately, components are expected to be white box and the verifier has access to the “source code“ of the design.

SAVE needs to continue in this track and extend the approach within the framework of (non-white) components. One way to reduce the combinatorial explosion in the faulty system state-space may be to differentiate between transient and permanent classes of faults, and their effects on multi-domain systems.

1.7 Fault containment and fault tolerance

The goal of a safety systems engineer is to eliminate the existence of hazards to the lowest reasonably practical level (the ALARP principle – As Low As Reasonably Practical). When the hazards cannot be eliminated, reduce the impact of their existence, or when that is not possible at least try to control the hazards. Techniques for reducing the impact of faults fall into two broad categories of *hazard elimination* and *hazard reduction*.

Hazard elimination can be achieved using a range of techniques including physical elimination of a hazard (encapsulating a digital hardware for avoiding radiation effects, removal of program bugs using bug-chasing with formal analysis tools), and decoupling. High safety requirements in a computer-based system might require safety-critical parts of the system to be separated from the non safety-critical parts. The safety-critical parts are then assembled and reduced in order to decrease the number of interfaces to other software parts. A significant benefit of isolating the safety-critical parts in a program is that the resources dedicated to verification of safety can be used more wisely, i.e. no resources need to be wasted on non safety-critical software.

When the safety-critical modules of a software system have been identified it is possible to protect them with so called firewalls. The firewalls can be physical or logical. Physical firewalls can be provided by, e.g. memory management units or separate computers.

One way for hazard reduction is by anticipating errors and building shields or incorporating redundancy into the design of the system. An example of the first type arises where electromagnetic interference or cosmic radiation can make programs do inadvertent jumps in execution. Unused memory should therefore be initialized to *illegal instruction* or something similar that takes the computer to a safe state if executed. Likewise, memory boundaries should be guarded against overwrites.

Using redundancy is an old trick for avoiding failures[2]. Redundancy in data appears in CRC coding of messages, error correcting codes, checksums and parity bits. Redundancy in time mandates that every task to be run by a system is characterized by mandatory and optional parts, and the optional parts are only executed if the available time is not needed for rerunning the mandatory parts of tasks that failed to complete at first attempt. Redundancy in state appears in software (or hardware) replication and can have the style of:

- Primary/back-up (warm or cold passive), where the state of some replica is updated to take over upon failure of the primary
- Active replication, where the state of all replicas are consistently updated at all times
- Rotating leader and consensus-based decision making, where the state of replicas are consistently updated except in finite periods of instability.

A review of techniques for fault tolerance can be found in a survey by Nadjm-Tehrani and Szentivanyi [NS03]. It begins by explaining basic notions in the area of fault-tolerance and presents some building blocks for achieving robustness in presence of certain types of faults. A more detailed exposure to questions related to fault-tolerance, and a particular solution to it, in the wider context of dependability can be found in a book edited by Powell [Pow01]. Support for availability in middleware and trade-offs between availability and performance in precense of the above three mechanisms are studied by Szentivanyi and Nadjm-Tehrani [SN03].

1.8 Challenges in component based development

The work on treatment of hardware faults (both static and dynamic) with the field of fault tolerance, has shown a considerable progress in the last decades. However, common mode failures (CMF) are difficult to treat by one single remedy [LH94]. Lala and Harper propose that avoidance, removal, and tolerance is needed for effective treatment of CMF in safety-critical systems. This leads to the need for combination of formal methods for avoidance and removal, run-time techniques (e.g. watch dogs) for detection and confirmation, and tolerance techniques that prescribe recovery to a safe state via exception handling. Such exception handling interfaces are essential for components to be used in a safety-critical context. The work in SAVE should build up on and extend the state of the art for specifying consequences of component failures in the context of a system, as well as effects of environment failures in the context of a component.

Redundancy in time needs advanced analysis techniques to ensure that the available resources of a system suffice for potential re-execution of mandatory parts. Such analyses are challenging to perform if components are re-configurable and their run-time properties are a function of their current configuration.

[2] An early reference to the topic goes back to 1824 and is cited by Shin and Krishna, in their book on real-time systems (1997).

Redundancy in space can of course be engineered into a system, and there are excellent examples for this in advanced applications of today [For03]. However, there are projects with the vision to move the basic support for fault tolerance into the infrastructure that acts as middleware. Whether this is a feasible undertaking, and the exploration of the middle ground in between application-based implementation of fault tolerance and middleware-based implementation of fault tolerance is an interesting area to study.

Re-configurable hardware poses a special challenge in the context of safety-critical systems. While there are well-known methods for estimating reliability of digital hardware (based on profiling and historical data), the advent of re-configurable hardware such as FPGAs opens up the issue of how should these units be treated in the safety assessments. In terms of logic and flexible reprogramming these units are very similar to software. However, the safety engineer's traditional view of how to treat digital hardware in safety evaluations is far more lenient compared to the software counterparts.

The need for reduction of hazardous conditions in a system has the implication of making Commercial Of-The-Shelf-Software (COTS) almost unusable in safety-critical applications. Most COTS are made in such a general fashion that they can be used in a wide array of applications. But, most applications will not make use of all the functionality provided; this is hazardous. However, the main point of using COTS is to increase reliability[3], so there is a trade-off here. On one hand high reliability is desirable and on the other hand safety. As previously discussed in this section, just because software is deemed reliable does not mean that it is safe. Another catch with COTS is that they are proprietary and are thus not likely to have all functionality documented. The documentation of a software module's behavior is imperative, in order to make a hazard analysis of a system. This documentation might very well have to cover the requirements specifications, designs, implementations and all the way down to the machine code level. Striking the "right" balance between (somewhat opposing) requirements pairs reliability and safety on the one hand, and availability and timeliness on the other, is central to the research agenda of dependability, and hence a valuable source of problems within SAVE.

Acknowledgements

The text on state-of practice in this section was written from inputs by Lars Holmlund and Rikard Johansson from Saab AB, whose cooperation is gratefully acknowledged.

[3] Another main point is of course "time to market" – the shorter "turn around time" the better. This "time" is however in reference to reliability since it takes a certain time (and costs money) to achieve sufficient reliability. Managers and engineers do therefore want to buy finished COTS that are well proven and save them the time and effort designing the software.

1.9 References

[ÅNS99] O. Åkerlund, S. Nadjm-Tehrani, and G. Stålmarck, Integrating Formal Methods into System Safety and Reliability Analysis, in 17th international System Safety Conference, p326-336, System Safety Society, 1999.

[BCC03] M. Bozzano, A. Cavallo, M. Cifaldi, et.al., Improving Safety Assessment of Complex Systems: An Industrial Case Study, To appear in proceedings for Formal Methods in Europe, FME'03, Springer Verlag.

[BVÅ03] M. Bozzano, A. Villafiorita, O. Åkerlund, et.al. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems, in European Safety and Reliability Conference, ESREL'03, Balkema Publisher, June 2003.

[Ekr03] J.-P. Ekros, Software Reliability, book chapter in *Safety and Reliability, Interactions between machines, software and people*, J. Eklund and J.-P. Ekros (Eds.), Linköping University, ISBN:91-7373-600-7, 2003.

[For03] K. Forsberg, Design Principles of Fly-by-Wire Architectures, PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Sweden, June 2003.

[Gow94] L. Gowen, Specifying and Verifying Safety-Critical Software Systems. 7th IEEE Symposium on Computer-based Medical Systems. June, 1994.

[Hat97] L. Hatton, Re-examining the Fault Density-Component Size Connection, IEEE Software, Volume 14, issue 2, pages 89-97, March 1997.

[HMW01]D. Hamlet, D. Mason, and D. Woit, Theory of Software Reliability based on Components, International Conference on Software Engineering, 2001.

[Hol03] E. Hollnager, How to Assess the Risks of Human Erroneous Actions, book chapter in *Safety and Reliability, Interactions between machines, software and people*, J. Eklund and J.-P. Ekros (Eds.), Linköping University, ISBN:91-7373-600-7, 2003.

[HN03] J. Hammarberg, S. Nadjm-Tehrani, Development of Safety-Critical Re-configurable Hardware with Esterel, in 8th International workshop on Formal Methods in Industrial-Critical systems, FMICS'03, Electronic Notes in Theoretical Computer Science 80, Elsevier, June 2003.

[HTC98] M. P. E. Heimdahl, J. M Thompson, and B. J. Czerny, Specification and Analysis of Intercomponent Communication, IEEE Computer, 31(4), April 1998.

[IFIP] A. Avezienis, J.-C. Laprie, B. Randell, Fundamental Concepts of Dependability, Technical Report CS-TR-739 at University of Newcastle, 2001 (www.cs.ncl.ak.uk/research/pubs/trs/papers/739.pdf, visited 7th November 2003).

[Jon95] C. Jones, Patterns of Software Systems: Failure and Success, IEEE Computer, Volume 28, issue 3, pages 86-87, March 1995.

[Lev95] N. Leveson, Safeware, System Safety & Computers, Addison Wesley, 1995.

[LH94] J. H. Lala, and R. E. Harper, Architectural Principles for Safety-Critical Real-time Applications, Proceedings of the IEEE, 82(1), IEEE, January 1994.

[Lyu95] M. L. Lyu, Handbook of Software Reliability Engineering, McGraw Hill, 1995.

[NS03] S. Nadjm-Tehrani, D. Szentivanyi, Fault tolerance in Distributed Computer Systems, book chapter in *Safety and Reliability, Interactions between machines, software and people*, J. Eklund and J.-P. Ekros (Eds.), Linköping University, ISBN:91-7373-600-7, 2003.

[Oco95] P. D. T. O'Connor, Practical Reliability Engineering, 4th Edition, John Wiley & Sons, 2001.

[Pow01] D. Powell (Ed.), A generic Fault-tolerant Architecture for Real-time Dependable Systems . Kluwer Academic Publishers, 2001.

[SN03] D. Szentivanyi and S. Nadjm-Tehrani, Middleware Support for Fault Tolerance, To appear as chapter 28 in Middleware for Communications, Q. Mamoud (Ed.), John Wiley & Sons, 2004.

Chapter 6

Modeling and Programming of Vehicular Embedded Control Systems; towards model based development

Martin Törngren, DeJiu Chen, Jad El-Khoury and Ola Larses

This chapter builds upon and extends chapters 1 and 2 by treating programming and modeling of embedded control systems. First, the evolution of the languages is studied and the characteristics and constraints that shaped this evolution are recapitulated (summarized from chapters 1 and 2). Secondly, basic terminology and issues related to programming and modeling are introduced. Thirdly, the chapter briefly describes the state of practice in programming and modeling of embedded control systems. Vehicular embedded systems include large amounts of functionality with quite varying requirements, characteristics and traditions; consequently, a wide variety of languages are used. The approach taken is to provide representative samples in order to get an overview of state of practice. Some of these approaches have a notion of reusable components; such component based approaches are further elaborated in more detail in chapter 4. Fourthly, we give an outlook on recent developments in the modeling of embedded control systems, and present model based development in more detail. We also qualitatively compare this with the component based approach. The chapter ends by summarizing related challenges for model based development.

6.1 Introduction; programming and modeling of embedded systems

As discussed in chapter 2, there is a variety of vehicular systems, where each vehicle in addition has a number of applications domains (compare Fig. 2.2, in section 2.1). Given this broad variety of systems, it is rather natural that a multitude of different *programming languages, modelling languages,* and *design methods,* are being used today. Figure 2A.1 illustrates this situation., which will be further discussed in the following.

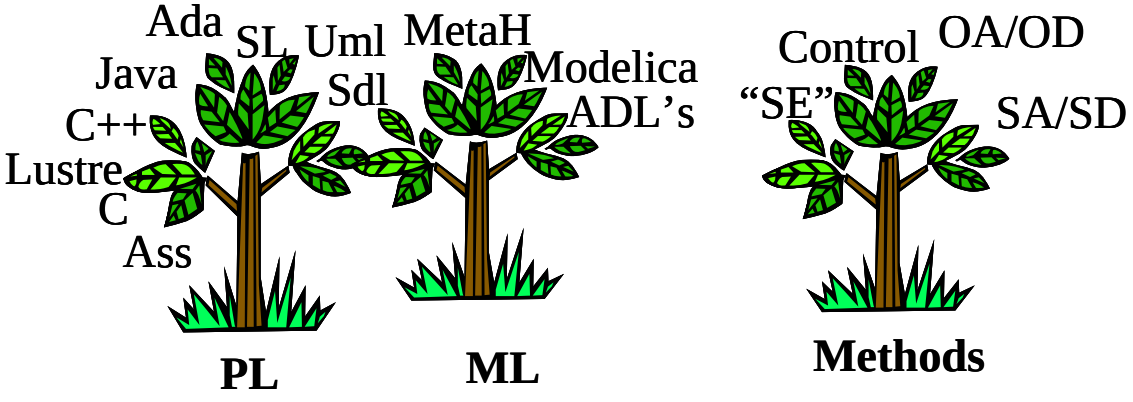


Figure 6.1 *The programming, modeling and design method trees, illustrating the overlap between programming and modeling languages. There is also a strong connection between different design methods and with specific languages (not illustrated in the figure). There are also languages used to program configurable hardware such as VHDL and SystemC that are not included.*

The usage of for example a particular programming and modelling approach is strongly related to traditions of engineering domains. For example, in process control, Programmable Logic Controllers (PLC's) are predominantly used, and along with them typically the IEC1311 programming standards. This approach is also being used in the train industry. If we take the automotive industry on the other hand, we find programming being carried out in for example C, UML or Simulink, depending on which department we are visiting. The same goes for modelling. It goes without saying that this plethora of languages and approaches does not come without problems; for embedded systems it is a sign of the relative immaturity of the area.

In the following we will attempt to define and relate different languages and modelling approaches. It is appropriate, however, as a starting point to briefly recall the characteristics and particular requirements of embedded control systems, since these have influenced the techniques that are used today, and since they also drive further developments.

6.1.1 Characteristics and constraints influencing languages

Essential characteristics and constraints of embedded control systems that need to be taken into account in programming and modelling include:

- *Different functionality and models of computation.* As apparent from chapter 2, embedded control systems are normally structured into a system platform and applications, each typically including several layers. This means that there will be activities that belong to the system platform for example dealing with initialisation, logging, basic communication services, drivers for sensor readings and outputs, and error detection. For the application there will be activities such as motion control, diagnostics related to the control, human machine communication, etc. These different activities correspond to different *models of computation* referring to the type of computations and interactions among software/hardware entities that take place. Examples of models of computation include difference equations, state machines and as part of them logical expressions, boolean operations, and task interactions.
- *Tight coupling to the environment.* Apart from driver routines there is a need to have models of the environment which also can include humans. Such models are important for a variety of reasons including analysing and designing interactions with the environment, including mechanisms for error detection. The real world is truly parallel, thus the need to express parallel activities (compare the first bullet).
- *Real-time constraints*, arising from the tight coupling with the environment. These constraints gives the need to define and describe different execution strategies in terms of both time and event triggering, (possibly several) scheduling policies, synchronization, and timing supervision.
- *Resource constrained implementations.* As discussed in chapter one, some vehicular systems are highly resource constrained (mainly due to strict cost constraints). In such applications, trade-offs between the system behaviour (quality of service) and the resources required (processing, memory and power) is essential.
- *Distributed systems and mapping,* that is the need to assign functions to different nodes of a distributed system, to define the tasks of the system, and their implementation in software and/or hardware.
- *Fault model, error detection, and error handling strategies.* This is essential with respect to several dependability attributes, such as safety and reliability. Such properties obviously need attention both during design time (e.g. through context

modelling and hazard analysis, formal verification and testing) and during run-time (in terms of error detection and handling mechanisms).

- *Product complexity management.* Because of the multitude of different technologies, competencies required and aspects addressed (e.g. performance, safety, reliability, and timing), complexity management is essential. It can be addressed in a number of ways including information hiding, separation of concerns, and documentation of design rationale. These techniques allow developers to focus on certain issues of the system at a certain development stage or for a certain purpose, while suppressing other irrelevant details or aspects. For example, views can be used to provide representations from particular perspectives, such as customers view, safety engineers view, and maintenance engineers view. Documenting design rationale is essential for reuse of components whose correct operation in a given context may well rely on environmental assumptions, on the existence of other components, on a particular timing, on a particular fault model and error handling for safe operation, etc. etc.
- *Flexibility with respect to future changes and product variants.* Vehicular systems have a long life time, so changes need to be envisioned and designed for to avoid costly redesign. Examples of this includes incorporating new hardware (because the old hardware can become obsolete), allowing customers to upgrade or even downgrade vehicles, and the handling of product variants, implying the need for configurability.
- *Architectural design to meet requirements.* Fulfilling different and partly conflicting quality attributes such as performance, safety and flexibility requires trade-offs to be made. Early design space exploration through model based design provides ne opportunities in this regard.

The above aspects to a large extent define what needs to be programmed and modelled when developing an embedded control system. This evolutionary aspect of languages is the topic of the following section.

6.1.2 Evolution of programming and modelling languages

The usage and developments of programming languages has been driven by the above characteristics and constraints. Consequently, we find “safe” languages, languages that have been extended to better support real-time constraints and parallellism, languages that are good at low level operations, and so on.

To cope with the system complexity efficiently there has over the years been a constant trend to raise the abstraction levels at which the systems are being programmed and modelled, see Figure 2A.2 which illustrates this for programming languages. Assembly languages were the main ones used for more than 20 years ago. Then came the shift to high-level programming languages with the idea to provide programmers with more powerful tools, that to some extent would relieve them of the burden of knowing the implementation (e.g. the CPU) by heart. Concerns were then raised whether the compilers would be able to produce efficient

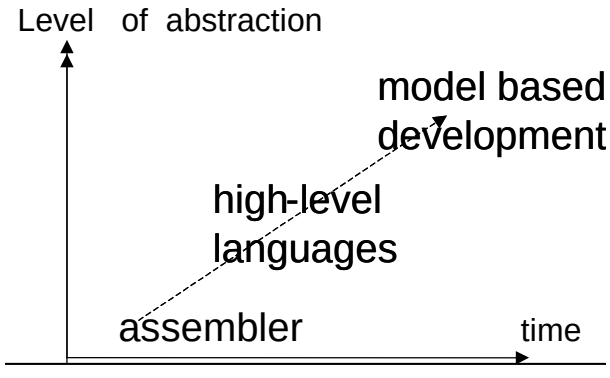


Figure 6A.2 *Increasing the abstraction levels*

and reliable code. The same concerns are now being raised with regards to code generation from models.

Along the way, a multitude of different design methods have been proposed (cmp. Fig. 2A.1). In software engineering, these include several variants of 'Structured Analysis and Design', and 'Object Oriented' approaches (see e.g. Cooling, 2003). There are also traditions from control engineering that implicate certain design procedures and come along with certain programming and modelling styles (e.g. block and data-flow diagrams). Yet other methods are found in engineering design and systems engineering (see e.g. Eppinger, Pahl & Beitz).

For programming languages the evolution can be illustrated in several ways. C++, Java and ADA are related to efforts in raising the abstraction levels of software programs. However, Java and C++ have shortcomings when it comes to efficient and predictable real-time implementations. This is also seen in the RT-Java development and related research (RT-Java ref, Nilsson).
We do find Java and C++ in vehicles today, but then mainly for non real-time and non safety related tasks. ADA is a high-level language that is "safe, and that supports real-time programming. However, it is today mainly used in military systems.

The C-programming language is highly flexible in that it allows most things to be programmed. It is thus useful for many purposes and in particular for low level platform programming, but it is highly unsafe (Ref: or elaboration). This has spurred efforts to standardization (ANSI-C), guidelines for safety critical systems (Misra), and definitions of safe subsets of C (Ref: Hatton). Because of its widespread use, a further development over the recent years is the use of the C-language as an intermediate format as indicated in Figure 2A.3.

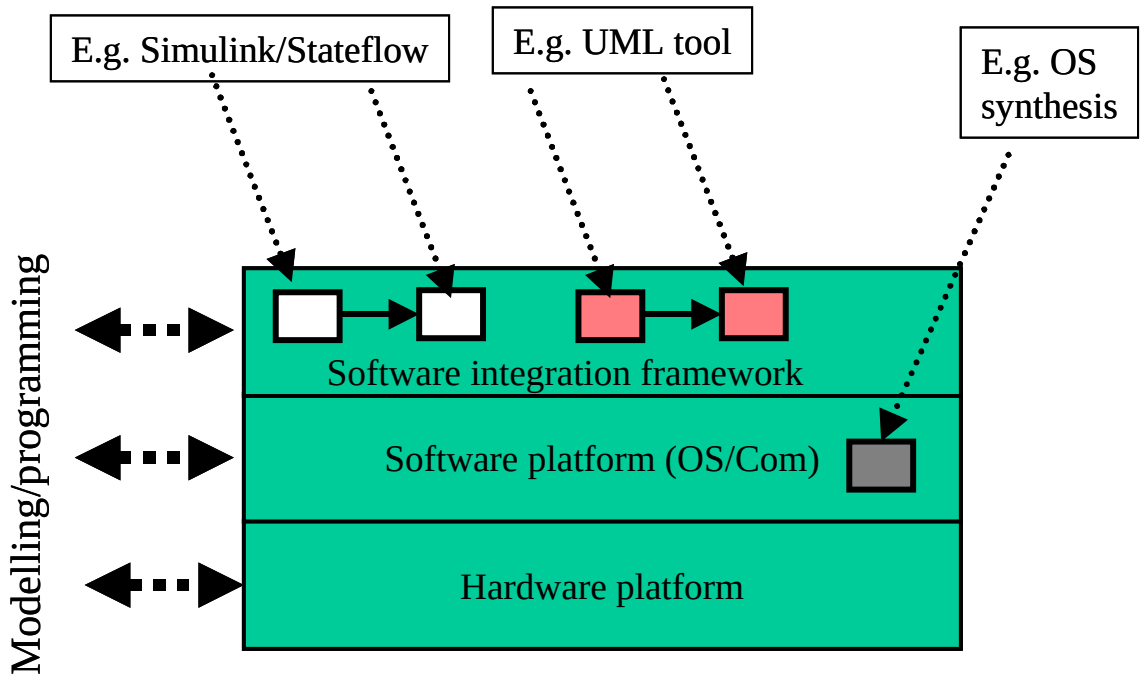


Figure 6.3 *Illustration of programming needs including the platform and the applications. The trend to synthesize programs/configurations from models is also indicated.*

By using higher level modeling languages such as UML and Simulink/Stateflow, it is in principle possible to develop the applications more efficiently since developers can focus on the application (a kind of separation of concerns). Then, either the resulting design model is manually translated into C-code, or code generation is used to integrate the resulting C-code with the platform. The use of code generation has increased significantly only over the last few years in the vehicular industry. One example is Aerospace, where the graphical representation of the Lustre language is used for application programming, from which C-code is generated (within the Scade tool) [ref].

Modelling languages (recall Fig. 2A.1) have been evolving mainly to serve the following purposes of improving design and maintenance. To better explain this it is relevant to define what is normally meant by a model.

Definition of a model: *A model constitutes a simplified representation of our knowledge about a system. To be useful, a model of a system should have a well defined purpose, balance between simplicity and complexity (detail), be expressed in a formal fashion such that mathematical tools can be applied, and finally be robust*, (Sellgren, 2003).

Models are central in mature engineering disciplines, and are used for several purposes. Models can provide descriptions of several relevant system levels such as requirements, designs, and implementations; and their properties such as safety, reliability and timing. Models with a graphical representation can facilitate communication between people and play an important role for documentation purposes; large amounts of time are wasted in trying to understand complex systems (ref). Serving, in particular, early design stages is essential and provides new opportunities by facilitating early analysis and better understanding in project teams (by use of descriptions of overall designs). A clear trend is that of separating specifications and design, from the implementation. This has the benefit of separating the WHATs from the HOWs, and allows the WHATs to be reused, designed and verified more easily.

The evolution of modelling languages has taken several directions. These and recent trends are discussed in section 6.3.

6.1.3 Relation between programming and modelling languages

From the above discussion it should be apparent that there are strong similarities, yet differences, between modelling and programming languages. It can be noted that there exists an overlap. For example, languages that can be used both for modelling and for programming include Simulink, SDL and subsets of the UML models.

A common denominator for programming and modelling languages are the desires to provide readable and flexible, yet descriptions that are formal enough to enable analyses and synthesis from programs/models.
The main differences comes from the fact that programming languages are mainly intended to support detailed design in terms of constructing programs. Modelling languages, on the other hand, are introduced to provide abstractions of complex systems for specific purposes, such as communication and/or as blueprint of system architecture. However, model based development makes the gap between the two smaller by providing synthesis (code generation) to bridge the gap between models and detailed design.

Definition of a modelling language: A modelling language is defined by an abstract syntax, a concrete syntax and its interpretation. The abstract syntax defines the concepts, relationships, integrity constraints and model composition principles available in the language, thus

determining all the syntactically correct models that can be built. The concrete syntax defines the form of visualization; graphical, textual or both. The interpretation defines the meaning of the entities of the language and the resulting models, i.e. its semantics, (Sztipanovits and Karsai, 2003).

6.2 State of practice: Programming of embedded control systems

Many parts of an embedded system needs to be programmed and there is a wide variety of approaches being utilized. We here provide representative examples to illustrate this variety in the state of practice. In addition, there are a multitude of development scenarios. For example, in some cases a vehicle builder may provide specifications of a subsystem and a supplier will design, implement and deliver the subsystem. Another example is where the vehicle builder develops parts of the software and does the integration on a control unit (hardware and some software) which is delivered by a supplier. Other scenarios include whether code generation is used from models, or whether the models are manually translated into code. For other examples see for example the Artist roadmaps (Artist, 2003) ++.

6.2.1 Programming trains at Bombardier

Bombardier uses the programming languages defined in the IEC 61131-3 standard, for programming their devices. The primary language used is the graphical language called function blocks, although the standard defines three graphical and two textual programming languages. Eventually, the application code which utilizes the underlying layers are situated on top in the figure.

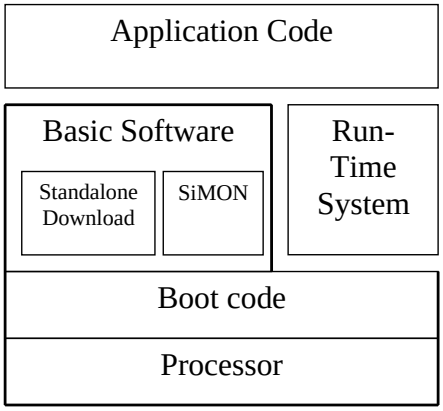


Figure 6.4. *A rough decomposition of a typical node*

The basics of the language are that an application is divided into a number of function blocks. Each function block has a set of in and out ports, and a hidden internal implementation. Each function block can contain both data and an implementation as a function. The intention with the function blocks is that they should be equivalent to integrated circuits, representing a specialized control function. The external data interface of a function block is ports and the function block implements one single function. Ports are either in or out ports, and have a type and a name.

When using the graphical FBD approach, composing a system is a matter of connecting function blocks, similar to electronic circuit diagram designs. Function blocks can also be composites of other function blocks, a picture derived from the standard visualizing a composed function block declaration with external interface and body is shown in figure x. The figure consists of two parts, the external interface definition in the upper half, and the

body with interconnected function blocks in the lower part. Notice that ports inside the function block in the upper part of the figure has the same name as the boundary ports in the lower part of the figure, when specifying the internal structure (body).

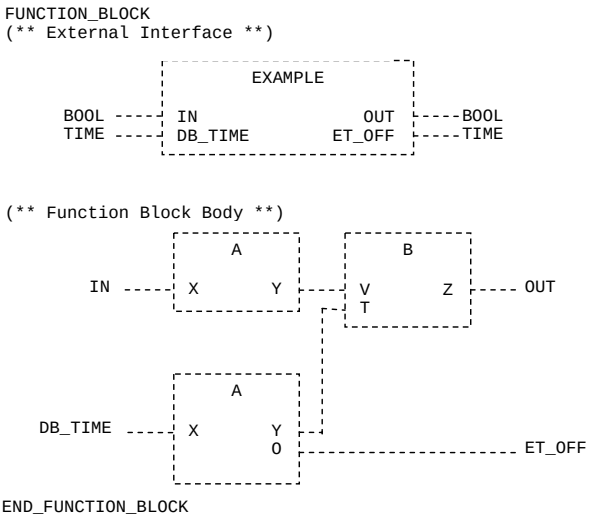


Figure 6.5, graphical function block declaration

6.3 Modeling of embedded control systems – trends

The basic needs for modeling arises due to the increasing complexity of the products.
Thus the subsequent needs to improve the cooperation among people, fault avoidance (correct by construction) and early fault removal (model verification and validation) measures.
Modelling constitutes and essential facet of engineering, and is natural ingredient in mature engineering disciplines. Models do not exist in isolation; they are developed within particular design contexts for various purposes. Successful models can be used throughout the life cycle of a product. Thus, model based development is characterized by a systematic usage of modelling throughout the life cycle to support product development and maintenance.

For embedded control systems there are a multitude of aspects to model for different purposes. Different aspects include

- Different levels of design, including requirements, environment, solutions, and the implementation.
- Constraints and properties related to desired and emerging behaviors such as functionality, performance and safety.

Different purposes include support of

- Design activities such as communicating, validating, modifying and analysing designs.
- Maintenance, by system documentation

The evolution of modelling languages has taken several directions (ElKhoury et al., 2003):

- Modelling techniques just above the programming language level. Examples of this category include MetaH and some ADLs such as Unicon [refs]
- Systems engineering approaches, such as UML/SysML, East-EAA, AADL and ACME [refs]
- Functional modelling techniques, such as Modellica, Ptolemy and SDL
- Function/Architecture codesign approaches, such as VCC and AIDA/Xilo (compare also the SETTA and Safe-Air projects)

- Quality or aspect specific approaches; e.g. modelling approaches dealing specifically with timing and/or with platform modelling [refs]

These modelling language efforts are briefly described in the following. As a general observation, the modelling in embedded systems has evolved from structural descriptions to also include abstract behavioral models.

- ACME is an architecture description language (ADL) and toolset, developed by Carnegie Mellon University since 1995. It is intended to be a common interchange format for other architecture design languages and tools. [7][8].
- Giotto is a programming language for embedded hard real-time control systems with periodic behaviour. It is developed at the University of California at Berkeley. Giotto separates the platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues. [9][10][11]
- Lustre is a formal approach that adopts the synchronous paradigm, providing a simple and verifiable way of designing the system functionality. Its data flow model is similar to conventional control design. [12][13][14][15][16][17][18]
- MAST is developed by the computer and real-time group, at the university of Cantabria, Spain. It is a modelling approach whose foundation evolved from the need to perform timing analysis of a system. [19][20][21]
- MetaH is a language and toolset for describing, analysing, and implementing realtime safety/mission critical computer systems for avionics control applications. It is being developed by Honeywell Technology Center since 1980's and is currently being extended into an Avionics Architecture Description Language (AADL). [22][23][24][25][26][27][28]
- Orccad is jointly developed by the BIP project and the Robotics Department of INRIA Rhône-Alpes. It is a software environment dedicated to the design and implementation of advanced robotics control systems. It also allows the specification and validation of missions to be achieved by the system. [29][30][31][32][33]
- Ptolemy is developed at the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley. Ptolemy facilitates functional description by providing various models of computations and hence allowing different domains to be described in the same language. [34][35][36][37][38][39]
- Rapide is a language and toolset developed by Stanford University since 1996, with the aim of supporting component-based development of distributed, time sensitive systems. It utilizes an architecture concept for defining the abstractions and a formal basis for specifying and analysing high-level behaviours and performance. [40][41][42]
- SDL originated in 1972 and has since been developed by CCITT (International Telegraph and Telephone Consultative Committee). It has been designed to specify and describe the functional behaviour of telecommunication systems, but has been used in various different applications. SDL focuses on a single descriptive mechanism for behaviour, namely communicating Extended State Machines, but can be used in combination with other languages such as MSC, ASN.1 and TTCN. [44][45][46][47].
- UniCon is a software ADL and toolset, developed by Carnegie Mellon University in 1995. The focus is on supporting the descriptions of architectural abstractions and styles found in various software systems and on constructing new systems from the architecture descriptions. [48][49]
- VCC is a commercial tool developed by Cadence. It is a system-level development environment for platform-based hardware/software co-design for automotive systems. It allows architectural decisions like hardware to software partitioning at the early stages of design. [50][51]

- Wright is a software ADL and toolset developed by Carnegie Mellon University in 1997. It supports the formalization of architectural styles, and model verification and validation based on a formal behaviour description in CSP. [52][53]
- Modellica
- SysML
- East-EAA
- Simulink/Stateflow
- Aida/Xilo
- Rosetta

ACME, Wright, UniCon and Rapide are focused on software architecture description. Lustre and Mast have a computer science origin with formal methods and scheduling theory background respectively, while VCC is an industrial approach. Orccad, Giotto and MetaH are domain-specific approaches that aim at control applications to be implemented on computer systems. Both Ptolemy and SDL focus on the high-level specification of the system, and less on implementation details.

The degree of support for the different design levels (functional, architectural, medium-level and detailed design) and for structuring and behaviour design at the different design levels. Figure 2A.XX and table ZZ present summaries of these factors for each of the studied approaches. Most approaches focus on the software development, except for VCC and MetaH that deal with the hardware architecture and middleware. Ptolemy and Lustre are dedicated for the behaviour design of the system, while ACME is dedicated for structure. All other approaches support different aspects at different levels during development.

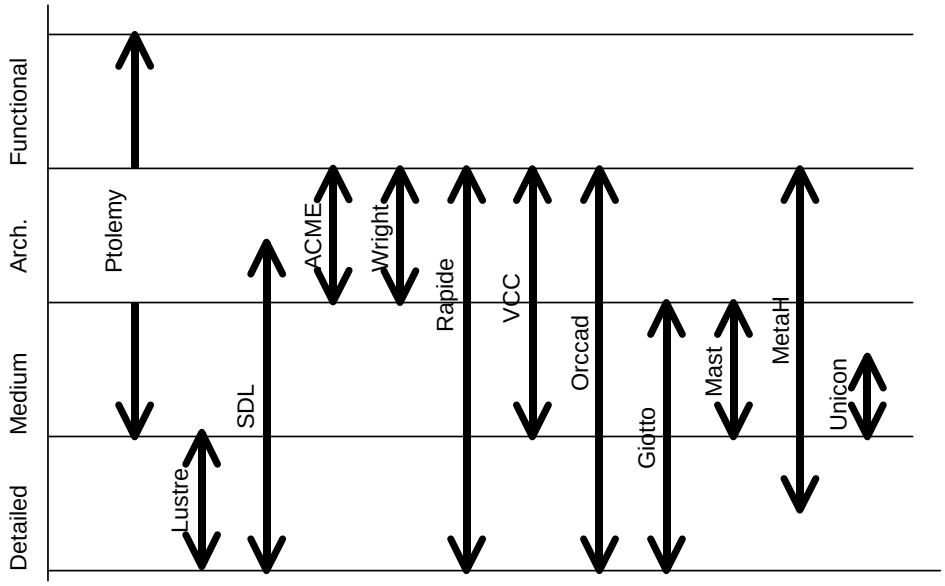


Figure 6.6 *Design levels focused on by the studied modelling approaches*

Table 6.1. A summary of design activities

Ptolemy	Behavioural design of functionality, with structural breakdown of the behaviour to support multiple models of computation.
Lustre	Behavioural design of reactive program, with some structuring through hierarchical *nodes*.
SDL	Structural breakdown of functionality, with behavioural design of *processes*.
Acme	Structural design (external languages & tools for behavioural design) of software architecture
Wright	Structural design of software architecture and behavioural design the parts

Rapide	Structural design of software architecture and behavioural design the parts
VCC	Structural design of hardware architecture, plus the allocation of software to hardware
Orccad	Structural design of mission architecture, and behavioural design of the parts.
Giotto	Structural design of system into modes and tasks, with behavioural design. The allocation of software to hardware is also supported.
MAST	Structural design of activities into tasks with behavioural design of real-time aspects. The allocation of software to hardware is also supported.
MetaH	Structural design of application into software and hardware architectures, with behavioural design focusing on real-time and reliability aspects.
Unicon	Structural design of software, with scheduling support.

6.4 Towards model based development

To be able to deliver vehicles that are cost-efficient, safe, of good quality, and with the desired features, considering ever decreasing time-to-market constraints, there is an overall need to provide solutions to

- *The need to support tighter levels of co-operation and integration between vehicle manufactures and supplier companies.* The efficient use of distributed architectures implies that functions do not necessarily correspond one-to-one with architecture components and that more sophisticated models of architectural components than are available today are needed to map "optimally" functions to components. This implies that a much tighter relationship among design teams of different companies of the supply chain must be established. Because designers that must collaborate are now distributed in different companies, it is essential that an overarching design methodology be devised with the adequate support of tools and modelling techniques.
- *The need to reduce development time* by avoiding unnecessary iteration cycles, entailing the need to make qualified architectural decisions early. Implied by this requirement, is the need to re-use features and components throughout all levels of the design hierarchy. This is also required in view of the ever increasing variety of vehicular variants. Standardization is essential.
- *The need to ensure high quality and optimization, despite the increasing system complexity.* System qualities of concern include performance, reliability, safety and cost. To be able to verify designs and to assess such crucial system properties at the different stages of development, there is a need for formal, or at least semiformal methods.

Obviously, all of the above bullets in some way or the other affect costs, and typically have in common that they attempt to move costs from integration testing, maintenance and recalls to the design phase. In common for the above needed solutions, is the requirement for **modeling languages** that enable rigorous analysis and design automation.

Figure 2A.XY outlines these basic needs. Foremost and first, competent people has to be there! Given this basic stage, however, we find companies with undefined structures, poor documentation and no systematic approach to development. The next step towards mature engineering is therefore a well defined process and methodology, where the latter for example could consist of documentation of proven partial solutions, methods for specific purposes such as testing, general principles and heuristics appropriate to the products being developed.

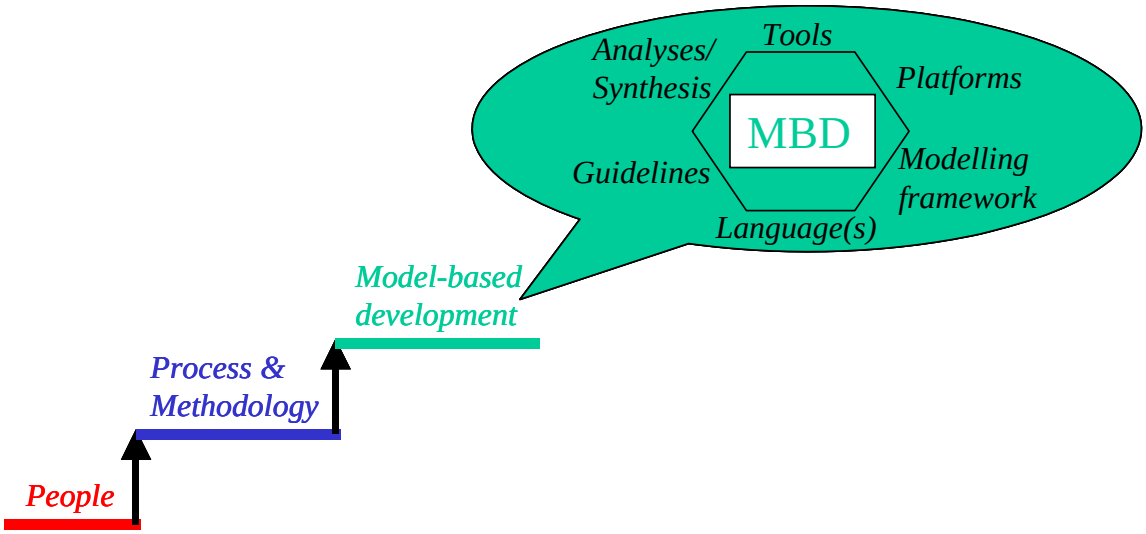


Figure 6.7 *Towards Model Based Development. The right hand part of the figure illustrates the ingredients considered an integral part of a mature model based approach.*

However, even these fundamental steps become insufficient as the system complexity is increasing and given tough market demands requiring cost-efficient, high quality products that are delivered on time; thus the final step: design automation.
We define **Model Based Development (MBD)**, as a development approach that includes all parts of Figure 2A.XY. MBD is the characteristic of mature engineering disciplines, where mechanical and electrical engineering are examples.

The following observations can be made regarding MBD:

- A mature MBD approach requires all parts of Fig. 2A.XY. For Design Automation this means that it is not sufficient with model based specifications since manual translation is error prone. For example, without analysis techniques there is no way to carry out automated design space exploration and verification; this in turn requires formal enough languages. In addition, without an appropriate process in place, MBD will be difficult to implement in an organization.
- MBD is today only carried out for subsystems or disciplinary parts of systems.
- Design automation is one essential aspect of MBD – i.e. the possibility to automate the evaluation (e.g. testing) and generation of different solutions.
- MBD provides opportunities beyond design automation since models should be reusable, both for new designs, but also as part of documentation for maintenance of existing products.
- To implement MBD requires not only a technological but also a cultural tradition. Thus, training and guidelines for developers will be required. Working with MBD also requires that the methodology has to be adapted accordingly. For example, testing and debugging needs to be supported at the model level (just as high-level language debugging at source code level is now possible).

MBD is strongly associated with the following ingredients (compare with right hand parts of Fig. 2A.XY):

- *Language(s).* MBD employs modelling languages providing appropriate levels of abstraction in system. The languages needs to be based on well defined modeling syntax and semantics, and support the specific system aspects or qualities such as timing and

reliability of interest in design. Associated with the language is also one or more suitable representations, typically including graphical visualization of models.

- *A modeling framework*. In MBD, models have many purposes and users – thus creating the needs for several views or submodels of a system. For example, for embedded control systems we need to model functionality both in terms of structure and behavior, and several aspects of the implementation such as physical layout and how functions are mapped to different electronic units. To avoid developing a multitute of completely separate models (for each level and aspect) there is a need for a modeling framework (or meta-model) that relates different models (e.g. through refinement) and aspects (e.g. functionality and timing behavior) to each other.
- *Tool.* To handle complex systems and for automation there is a need for computer aided engineering (CAE) tools.
- *Guidelines*. To develop systems according to an MBD approach, there is a need for guidelines that provides practical assistance in creating models, in designing using models and in structuring the models.
- *Analysis and synthesis techniques.* Such techniques are essential to assist designers and forms the basis for further design automation. Examples of analysis includes model based prediction of system level properties through simulation and/or analytical methods. Examples of synthesis includes generating suitable mappings of functions to processors/tasks, and code generation.
- *Platform.* For MBD, platforms are required to support design and implementation, and basically provide means for executing models in different environments. Examples of platforms include a simulation environment where models can be executed, platforms supporting rapid prototyping to execute models interacting with the controlled process, and implementation platforms.

6.5 Comparing model based and component based approaches

We claim here that CBD and MBD go hand in hand; they do however have different origins and emphasis. There are some misconceptions that sometimes make this less clear.

MDB misconception: MBD is just about synthesizing code; synthesize and you are done.
CBD misconception: Just select and connect your components, and you are done.

It is clear that given the constraints mentioned in section 6.1.1 make such claims invalid in the general case.

The main emphasis of MBD has been discussed above. MBD strives to cover several abstraction levels to support several design stages and issues. As such, MBD is more top-down oriented, although it does bring a holistic view to systems development, and for example, incorporates bottom up abstractions (Setta, Safe-Air, Aida). Within MBD there is a clear need to manage components at the different levels (reuse, encapsulation, managing complexity)

	CBD	***MBD***
Approach	Bottom-up, composition	Top-down, refinement, bottom-up abstraction
Reuse	SW components	At all covered levels
Requires	Run-time framework Component standard	Platforms and Tools Analysis & synthesis techniques Language(s), Modelling framework and Guidelines

Advantages (potential)	Reuse of SW Time to market reduced (if component composition works)	Reuse at the covered levels Time to market reduced Less design faults Reduced costs for systems integration and maintenance.
Problems/challenges	Composition/analysis techniques Lack of standards for embedded control systems Component models capturing non-functional properties	Modelling skills/people Initial costs (tools, training) Costs: Licenses and maintenance Methodology/guidelines Analysis & synthesis techniques Modelling framework

CBD on the other hand is characterized by packaging software into components for reuse. The CBS approach normally relies on an operational framework that can execute standardized components. The actual configuration may take place pre-run time (exemplified by some specific CBD approaches for real-time control systems) or during run-time (traditional SW CBD exemplified by Corba). CBD is by its nature bottom up oriented, but strives to increase the levels of abstraction, and to improve analysis and component documentation by adding models (compare CMU: analytical interface).

It should be noted that CBD is practiced in electronics and mechanical engineering; it can infact be argued that mechanical engineers are more used to CBD than software engineers since they since long when working in CAE environments with components and think in terms of components and interfaces.

6.6 Conclusion

Model based development (MBD) is seen as a necessary technology, to master both the product and the development complexity. However, while MBD is practiced for subsystems, there is no holistic MBD available yet, and there are plenty of research and industrial efforts trying to develop MBD further, see e.g. Artist (2003).

The challenge is how multiple models can be properly linked together at certain levels of abstraction for the purpose of analysis, trade-offs, and other development activities. For this reason, a higher-level model-of-models that specifies the relations of multiple models in regard to system structure decomposition and the dynamics of development may be necessary.

- Numerous efforts on tool interfacing and modeling standards
- Lack of handling non-functional aspects (true for control and SW)
- Concerns about efficiency and reliability of the tools continue to be raised as this process progresses.
- S*hortage of methodology*: Programming at higher levels of abstraction does not automatically lead to improvements. There is a need for methodologies and guidelines addressing issues including readability, reusability, tool-related constraints, and efficient and reliable implementation.
- Zoo of modeling languages.
- Synthesis for distributed systems

Even greater challenges face researchers and developers when it comes to the development of mechatronic systems such as vehicles. Mechatronic systems come with a need to represent information of requirements and constraints, functionality, and their realizations in mechanics,

electronics and software – a software perspective only is obviously not sufficient. Relations between entities (e.g. components) at the same, and between different levels and disciplines need to be maintained to ensure that products will fulfill requirements and be maintainable.
Another issue is the variety of modeling languages and tools, [8]. These languages and tools are not always inherently compatible with each other. Most of the tools and languages are developed for specific domains. There is a risk that the organizations get caught in a tool environment, as there are high costs involved in both procuring and adapting new tools.
It is clear that introducing MBD requires a fundamental change in that way that work is performed, and will require a significant initial development effort. This relates both to staff training, costs for tools, and to initial model development.

7 Real-Time Systems

In this chapter we will provide an introduction to issues, techniques, and trends in real-time systems. We will specifically discuss timing properties, timing analysis, real-time operating systems, real-time scheduling, real-time communication, as well as real-time issues in component-based design of real-time systems. For each of these areas, state-of-the-art tools and standards are presented.

7.1 Introduction

Consider the *airbag* in the steering-wheel of your car. It should after the detection of a crash (and only then) inflate just in time to softly catch your head to prevent it from hitting the steering-wheel; not too early - since this would make the airbag deflate before it can catch you; nor too late - since the exploding airbag then could injure you by blowing up in your face and/or catch you too late to prevent your head from banging into the steering wheel.

The computer controlled airbag system is an example of a real-time system (RTS). But RTSs come in many different flavours, including vehicles, telecommunication systems, industrial automation systems, household appliances, etc.

There is no commonly agreed upon definition of what a RTS is, but the following characterisation is (almost) universally accepted:

- RTSs are computer systems that physically interact with the real world.
- RTSs have requirements on the timing of these interactions.

Typically, the real-world interactions are via sensors and actuators, rather than the keyboard and screen of your standard PC.

Real-time requirements typically express that an interaction should occur within specified timing bound. It should be noted that this is quite different from requiring the interaction to be as fast as possible.

Essentially all RTSs are embedded in products, and the vast majority of *embedded computer systems* are RTSs. RTSs is the dominating application of computer technology, as more than 99% of the manufactured processors (more than 8 billions in 2000 [Hal00]) are used in embedded systems.

Returning to the airbag system, we note that it in addition to being a RTS it is a safety-critical system, i.e., a system which due to severe risks of damage have strict *Quality of Service* (QoS) requirements, including requirements on the functional behaviour, robustness, reliability, and timeliness.

A typical strict timing property could be that a certain response to an interaction always must occur within some prescribed time, e.g., that the charge in the airbag must detonate between 10 and 20 ms from the detection of a crash; violating this must be avoided at any cost, since it would lead to something unacceptable, like you having to spend a couple of months in hospital. A system that is designed to meet strict timing requirements is often referred to as a *hard real-time system*. In contrast, systems for which occasional timing failures are acceptable - possibly because this will not lead to something terrible - are termed *soft real-time system*.

An illustrative comparison between hard and soft RTSs that highlights the difference between the extremes is shown in table 1. A typical hard real-time system could in this context be an engine control system, which must operate with μs-precision, and which will severely damage the engine if timing requirements fail by more than a few ms. A typical soft real-time system could be a banking system, for which timing is important, but where there are no strict deadlines and some variations in timing are acceptable.

Table 1: Typical characteristics of Hard and Soft Real-Time Systems [Kop03]

Characteristic	**Hard Real-Time**	**Soft Real-Time**
Timing requirements	hard	soft
Pacing	environment	computer
Peak-load performance	predictable	degraded
Error detection	system	user
Safety	critical	non-critical
Redundancy	active	standby
Time granularity	millisecond	second
Data files	small	large
Data integrity	short term	long term

Unfortunately, it is impossible to build real systems that satisfy hard real-time requirements, since due to the imperfection of hardware (and designers) any system may break. The best that can be achieved is a system that with very high probability provides the intended behaviour during a finite interval of time.

However, on the conceptual level hard real-time makes sense, since it implies a certain amount of rigour in the way the system is designed, e.g., it implies an obligation to prove that the strict timing requirements are met, at least under some simplifying, but realistic, assumptions.

Since the early 1980ies a substantial research effort has provided a sound theoretical foundation (e.g. [RTS, Klu]) and many practically useful results for the design of hard real-time systems. Most notably, hard real-time system scheduling has evolved into a mature discipline, using abstract, but realistic, models of tasks executing on single CPU, multiprocessor, or distributed computer systems, together with associ-

ated methods for timing analysis. Such *schedulability analysis*, e.g., the well-known rate-monotonic analysis [LL73, KRP+98, ABD+95], have found significant use also in some industrial segments.

However, hard real-time scheduling is not the cure for all RTSs. Its main weakness is that it is based on analysis of the worst possible scenario. For safety-critical systems this is of course a must, but for other systems, where general customer satisfaction is the main criteria, it may be too costly to design the system for a worst-case scenario that may not occur during the system's lifetime.

If we look at the other end of the spectrum, we find the *best-effort approach*, which is still the dominating approach in industry. The essence of this approach is to implement the system using some best practice, and then use measurements, testing and tuning to make sure that the system is of sufficient quality. Such a system will hopefully satisfy some soft real-time requirement; the weakness being that we do not know which. On the other hand, compared to the hard real-time approach, the system can be better optimised for the available resources. A further difference is that hard real-time system methods essentially are applicable to static configurations only, whereas it is less problematic to handle dynamic task creation etc. in best-effort systems.

Having identified the weaknesses of the hard real-time and best-effort approaches major efforts are now put into more flexible techniques for soft real-time systems. These techniques provide analysability (like hard real-time), together with flexibility and resource efficiency (like best-effort). The basis for the flexible techniques are often quantified *Quality-of-Service* (QoS) characteristics. These are typically related to non-functional aspects, such as timeliness, robustness, dependability, and performance. To provide a specified QoS, some sort of resource management is needed. Such a QoS-management is either handled by the application, by the operating system, by some middleware, or by a mix of the above. The QoS-management is often a flexible on-line mechanism that dynamically adapts the resource allocation to balance between conflicting QoS-requirements.

7.2 Analysis of Real-Time Systems

Analysis techniques are techniques that are used to investigate what properties a component, or a system of components, exhibit. Of particular interest when dealing with RTSs is the issue of *temporal correctness*, i.e. that actions are performed at the right time (that is, we are not concerned with the actual values of an action, only the timing of the action).

Classical program-analysis techniques, that focus on functional and structural correctness are well developed. However, to validate the correctness of an RTS, analysis techniques to validate the temporal correctness must be employed *in addition to* other analysis techniques. In this section we focus on techniques to analyse the temporal

behaviour of RTSs.

7.2.1 Timing Properties

Timing analysis is a complex problem. Not only are the timing-analysis techniques sometimes complicated, but also the problem itself is elusive; for instance, what is the meaning of the term "program execution-time"? Is it the average time to execute the program, or the worst possible time, or does it mean some form of "normal" execution time? Under what conditions does a statement regarding program execution-times apply? Is the program delayed by interrupts or higher priority tasks? Does the time include waiting for shared resources? etc. etc.

To straighten out some of these question marks, and to be able to study some existing techniques for timing analysis, we structure timing properties into four major types. Each type has its own purpose, benefits and limitations. The types are listed below.

Execution time This refers to the execution time of a singe task (or program, or function, or any other unit of single threaded sequential code). The result of an execution-time analysis is the time (i.e. the number of clock cycles) the task takes to execute, when executing undisturbed on a single CPU, i.e. the result should not account for interrupts, preemption, background DMA transfers, DRAM refresh delays, or any other types of interfering background activities.

At a first glance, leaving out all types of interference from the execution-time analysis would give us unrealistic results. However, the purpose of the execution-time analysis is *not* to deliver estimates on "real-world" timing when executing the task. Instead, the role of execution-time analysis is to find out how much computing resources is needed to execute the task. (Hence, background activities that are not related to the task should not be accounted for.)

There are some different types of execution-times that can be of interest:

- Worst-Case Execution-Time (WCET) – This is the worst possible execution time a task could exhibit, or equivalently, the maximum amount of computing resources required to execute the task. The WCET should include any possible atypical task execution such as exception handling or clean up after abnormal task termination.
- Best-Case Execution-Time (BCET) – During some types of real-time analysis, not only the WCET is used, but as we will describe later, having knowledge about the BCET of tasks is useful.
- Average Execution-Time (AET) – The AET can be useful in calculating throughput estimates for a system. However, for most RTS analysis the AET is of

less importance, simply since a reasonable approximation of the average case is easy to obtain during testing (where typically, the average system behaviour is studied). Also, only knowing the average and not knowing any other statistical parameters such as standard deviation or distribution function makes statistical analysis difficult. For analysis purposes a more pessimistic metric such as the 95%-quartile would be more useful. However, analytical techniques using statistical metrics of execution time are scarce and not very well developed.

Response time The response time of a task is the time it takes from the *invocation* of the task to the *completion* of the task. In other words, the time from when the task first is placed in the operating-system's ready queue to the time when it is removed from the running state and placed in the idle or sleeping state.

Typically, for analysis purposes it is assumed that a task does not voluntarily suspend itself during its execution. That is, the task may not call primitives such as `sleep()` or `delay()`. However, involuntarily suspension, such as blocking on shared resources, is allowed. That is, primitives such as `get_semaphore()` and `lock_database_record()` are allowed. When a program voluntarily suspends itself, then that program should be broken down into two (or more) analysis tasks.

The response-time is typically a *system level property*, in that it includes interference from other, unrelated, tasks and parts of the system. The response-time also includes delays caused by contention on shared resources. Hence, the response-time is only meaningful when considering a complete system, or in distributed systems, a complete node.

End-to-end delay The above described "execution time" and "response time" are useful concepts since they are relatively easy to understand and have well defined scopes. However, when trying to establish the temporal correctness of a system, knowing the WCET and/or the response-times of tasks is often not enough. Typically, the correctness criteria is stated using *end-to-end* latency timing requirements, for instance an upper bound on the delay between the input of a signal and the output of a response.

In a given implementation there may be a chain of events taking place between the input of a signal and the output of a response. For instance, one task may be in charge of reading the input and another task for generating the response, and the two tasks may have to exchange messages on a communications link before the response can be generated. The end-to-end timing denotes timing of externally visible events.

Jitter The term *jitter* is used as a metric for variability in time. For instance, the jitter in execution time of a task is the difference between the task's BCET and WCET. Similarly, the response-time jitter of a task is the difference between its best-case response-time and its worst-case response-time. Often, control algorithms have requirements

that the jitter of the output should be limited. Hence, the end-to-end jitter is sometimes a metric equally important as the end-to-end delay.

Also input to the system can have jitter. For instance, an interrupt which is expected to be periodic may have a jitter (due to some imperfection in the process generating the interrupt). In this case the jitter-value is used as a bound on the maximum deviation from the ideal period of the interrupt. Figure 1 illustrates the relation between the period and the jitter for this example.

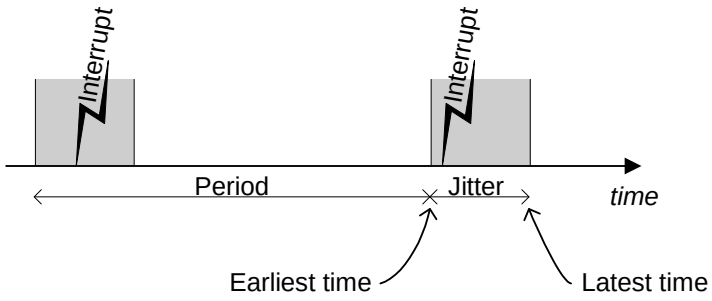


Figure 1: Jitter used as a bound on variability in periodicity

Note that jitter should not accumulate over time. For our example; even though two successive interrupts could arrive closer than one period, in the long-run, the average interrupt interarrival-time will be that of the period. (If jitter *does* accumulate, it can be modelled by using a shorter period.)

Other types of timing In the above list of types of timing properties, we only mentioned types related to tasks. However, in many RTSs, other timing properties may also exist. Most typical are delays on a communications network, but also other resources such as hard disk drives may be causing delays and need to be analysed. The above introduced times can all be mapped to different types of resources, for instance the WCET of a task corresponds the maximum size of a message to be transmitted, and the response time of message is defined analogous to the response time of a task.

7.2.2 Methods for timing analysis

When analysing hard RTSs it is essential that the estimates obtained during timing analysis are *safe*. An estimate is considered safe if it is guaranteed that it is not an underestimation of the actual worst-case time. It is also important that the estimate is *tight*, meaning that the estimated time is close to the actual worst-case time.

For the previously defined types of timings (section 7.2.1) there are different methods available:

Execution-time estimation For real-time tasks the WCET is the most important execution time measure to obtain. Sadly, however, it is also often the most difficult measure to obtain.

Methods to obtain the WCET of a task can be divided into two categories: (1) static analysis, and (2) dynamic analysis. Dynamic analysis is essentially equivalent to testing (i.e. executing the task on the target hardware) and has all the drawbacks/problems that testing exhibits (such as being tedious and error prone). One major problem with dynamic analysis is that it does not produce safe results. In fact, the result can never exceed the true WCET and it is very difficult to be sure that the estimated WCET is really the true WCET.

Static analysis, on the other hand, can give guaranteed safe results. Static analysis is performed by analysing the code (source and/or object code is used) and basically counting the number of clock cycles that the task may use to execute (in the worst possible case). Static analysis uses models of the hardware to predict execution times for each instruction. Hence, for modern hardware it may be very difficult to produce static analysers that give tight results. One source of pessimism in the analysis (i.e. overestimation) is hardware caches; whenever an instruction or data-item cannot be guaranteed to reside in the cache, a static analyser must assume a cache miss. And since modelling the exact state of caches (sometimes of multiple levels), branch predictors etc. is very difficult and time consuming, few tools that give adequate results for advanced architectures exists. Also, to perform a program flow and data analysis that exactly calculates, e.g., the number of times a loop iterates or the input parameters for procedures is difficult.

Methods for good hardware and software modelling do exists in the research community, however, combining these methods into high-quality tools has proven tedious.

Schedulability Analysis The goal of schedulability analysis is to deem whether or not a system is *schedulable*. A system is deemed schedulable if it is guaranteed that all task deadlines will always be met. For statically scheduled (table driven) systems, calculation of response-times are trivially given from the static schedule. However, for dynamically scheduled systems (such as fixed priority or deadline scheduling) more advanced techniques have to be used.

There are two main classes of schedulability analysis techniques: (1) response-time analysis, and (2) utilisation analysis. As the name suggest, a response-time analysis calculates a (safe) estimate of the worst-case response-time of a task. That estimate can then be compared to the deadline of the task and if it does not exceed the deadline then the task is schedulable. Utilisation analysis, in contrast, does not directly derive the response-times for tasks, rather they give a boolean result for each task telling whether or not the task is schedulable. This result is based on the fraction of utilisation of the CPU for a relevant subset of the tasks, hence the term utilisation analysis.

Both types of analysis are based on similar types of task models. However, typically the task models used for analysis are not the task models provided by commercial RTOSes. This problem can be resolved by mapping one or more OS-task to one or more analysis task. However, this mapping has to be performed manually and requires an understanding of the limitations of the analysis task model and the analysis technique used.

End-to-end delay estimation The typical way to obtain end-to-end delay estimations is to calculate the response time for each task/message in the end-to-end chain and to summarise these response times to obtain and end-to-end estimate. When using a utilisation based analysis technique (in which no response time is calculated) one have to resort to using the task/message deadlines as safe upper bounds on the response times.

However, when analysing distributed RTSs, it may not be possible to calculate all response times in one pass. The reason for this is that delays on one node will lead to jitter on another node, and that this jitter may in turn affect the response times on that node. Since jitter can propagate in several steps between nodes, in both directions, there may not exist a "right" order to analyse the nodes. (If A sends a message to B, and B sends a message to A; which node should be analysed first?) Solutions to this type of problems are called *holistic* schedulability analysis methods (since they consider the whole system). The standard method for holistic response-time analysis is to repeatedly calculate response-times for each node (and update jitter values in the nodes affected by the node just analysed) until response-times do not change (i.e. a fix-point is reached).

Jitter estimation To calculate the jitter one need not only perform a worst-case analysis (of for instance, response-time or end-to-end delay). It is also necessary to perform a best-case analysis.

However, even though best-case analysis techniques often are conceptually similar to worst-case analysis techniques, there has been little attention paid to best-case analysis in the research community. One reason for not spending too much time on best-case analysis is that it is quite easy to make a conservative estimate of the best-case: the best-case time is never less than zero (0). Hence, in many tools it is simply assumed that the BCET is zero, whereas great efforts can be spent analysing the WCET.

However, it is important to have tight estimates of the jitter, and to keep the jitter as low as possible. It has been shown that the number of *execution paths* a multitasking RTS can take dramatically increases if jitter increases [TH99]. Unless the number of possible execution paths is kept as low as possible it becomes very difficult to achieve good coverage during testing.

7.2.3 Trends and Tools

As pointed our earlier, there is a mismatch between the analytical task-models and the task-models provided by commonly used RTOSes. One of the basic problems is that there is no one-to-one mapping between analysis tasks and RTOS tasks. In fact, for many systems there is a N-to-N mapping between the task types. For instance, an interrupt handler may have to be modelled as several different analysis task (one analysis task for each type of interrupt it handles), and one OS task may have to be modelled as several analysis tasks (for instance, one analysis task per call to `sleep()` primitives).

Also, current schedulability analysis techniques cannot adequately model other types of task-synchronisation than locking/blocking on shared resources. Abstractions such as message queues are difficult to include in the schedulability analysis.[1] Furthermore, tools to estimate the WCET are also scarce. Currently only two tools that gives safe WCET estimates are commercially available [Abs, Bou].

These problems has lead to low penetration of schedulability analysis in industrial software-development processes. However, in isolated domains, such as in real-time networks, some commercial tools that are based on real-time analysis do exists. For instance, Volcano [CRTM98, Vol] provides tools for the CAN bus that allow system designers to specify signals on an abstract level (giving signal attributes such as size, period and deadline) and automatically derive a mapping of signals to CAN-messages where all deadlines are guaranteed to be meet.

On the software side tools provided by, for instance, TimeSys [Timb], Arcticus Systems [Arc], and TTTech [TTT] can provide system development environments with timing analysis as an integrated part of the tool suite. However, these tools all require that the software development processes is under complete control of the respective tool. This requirement has limited the use of these tools.

The widespread use of UML [OMG03b] in software design has lead to some specialised UML-products for real-time engineering [Rat, IL]. However, these products, as of today, do not support timing analysis of the designed systems. There is however recent work within the OMG that specifies a *profile* "Schedulability, Performance and Time" (SPT) [OMG03a], which allows specification of both timing properties and requirement in a standardised way. This will in turn lead to products that can analyse UML-models conforming to the SPT-profile.

The SPT-profile has however not been received without criticism. Critique has mainly come from researchers active in the timing analysis field, claiming both that the profile is not precise enough and that some important concepts are missing. For instance,

[1] Techniques to handle more advanced models include timed logic and model checking. However, the computational and conceptual complexity of these techniques has limited their industrial impact. There are however examples of commercial tools for this type of verification, e.g. [Tima].

the Universidad de Cantabria has instead developed the MAST-UML profile and an associated MAST-tool for analysing MAST-UML models [MHD01, MAS]. MAST allows modelling of advanced timing properties and requirement, and the tool also provides state-of-the-art timing analysis techniques.

7.3 Real-Time Operating-Systems

When building real-time computer-systems it is often beneficial to use a *Real-Time Operating System* (RTOS). An RTOS provides services for resource access and resource sharing, very much similar to a general-purpose operating-system. The main reasons not to use a general-purpose operating-system when developing RTSs are:

- High resource utilisation, e.g. large RAM and ROM footprints, and high internal CPU-demand.
- Difficult to access hardware and devices in a timely manner, e.g. no application level control over interrupts.
- Lack of services to allow timing sensitive interactions between different processes.

7.3.1 Typical Properties of RTOSes

The state of practice in real-time operating systems (RTOS) is reflected in [FAQ]. Not all operating systems are RTOSes. An RTOS is typically multi-threaded and preemptible, there has to be a notion of process priority, predictable process synchronisation has to be supported, priority inheritance should be supported, and the OS timing-behaviour should be known [Art03]. This means that the interrupt latency, worst-case execution time of system calls, and maximum time during which interrupts are masked must be known. A commercial RTOS is usually marketed as a runtime component of an embedded-systems development platform.

As a general rule of thumb one can say that RTOSes are:

- Being apt for the resource constrained environments where most RTSs operate.

  Most RTOSes can be configured pre runtime (e.g. at compile time) to only include a subset of the total functionality. Thus, the application developer can choose to leave out unused portions of the RTOS in order to save resources and/or get more predictable behaviour.

  RTOSes typically store much of their configuration in ROM. This is done for mainly two purposes: (1) minimise use of expensive RAM memory, and (2) minimise the risk that critical data is overwritten by an erroneous application.

- Giving the application programmer easy access to hardware features such as interrupts and devices.

  Most often the RTOSes give the application programmer means to install Interrupt Service Routines (ISRs) during compile time and/or during run time. This means that the RTOS leaves interrupt handing to the application programmer, allowing fast, efficient and predictable handling of interrupts.

  In general-purpose operating-systems, memory-mapped devices are usually protected from direct access using the MMU (Memory Management Unit) of the CPU. Hence forcing all device accesses to go through the operating system. RTOSes typically do not protect such devices and allow the application to directly manipulate the devices. This gives faster and more efficient access to the devices. (However, this efficiency comes at the price of an increased risk of erroneous use of the device.)

- Providing services that allow implementation of timing sensitive code.

  An RTOS typically has many mechanisms to control the relative timing between different processes in the system. Most notably the RTOS has a real-time process scheduler whose function is to make sure that the processes execute in the way the application programmer intended them to. We will elaborate more on the issues of scheduling in section 7.4.

  An RTOS also provides mechanisms to control the processes relative performance when accessing shared resources. This can for instance be done by priority queues, instead of plain FIFO-queues as is used in general-purpose operating-systems. An RTOS should provide resource-locking protocols with predictable real-time behaviour, such as the Priority Ceiling Protocol (PCP) or the Immediate ceiling Inheritance Protocol (IIP) ([But97, BW96] provides more details about these protocols).

- Being tailored to fit the development process that is typical when developing embedded systems (recall; most RTSs are also embedded systems).

  RTSs are usually constructed in a *host environment* that is different from the *target environment*, so called *cross platform* development. Also, it is typical that the whole memory image, including both RTOS and one or more applications, is created at the host platform and downloaded to the target platform. Hence, most RTOSes are delivered as source code modules or pre-compiled libraries that are statically linked with the applications at compile time.

7.3.2 Commercial RTOSes

There are an abundance of commercial RTOSes. Most of them provides adequate mechanisms to enable development of RTSs. Some examples are Tornado/VxWorks

[Win], LYNX [LYN], OSE [Sys], QNX [QNX], RT-Linux [RTL], and ThreadX [Log]. However, the major problem with these tools is the rich set of primitives provided. These system provides both primitives that are suitable for RTSs and primitives that are unfit for RTSs (or that should be used with great care). For instance, they usually provide multiple resource locking protocols; some of which are suitable and some of which are not suitable for real-time.

This richness becomes a problem when these tools are used by inexperienced engineers and/or when projects are large and project management does not provide clear design guidelines/rules. In these situations, it is very easy to use primitives that will contribute to timing unpredictability of the developed system. In our view, an RTOS should help the engineers and project managers by providing only mechanisms that helps designing predictable systems. However, there is an obvious conflict between the desire/need of RTOS manufacturers to provide rich interfaces and stringency needed by designers of RTSs.

There is a smaller set of RTOSes that have been designed resolve these problems, and at the same time also allow extreme lightweight implementations of predictable RTSs. The driving idea is to provide a small set of primitives that guides the engineers towards good design of their system. Typical examples are the research RTOS Asterix [Ast] and the commercial RTOS SSX5 [Liv99]. These systems provide a simplified task model, in which tasks cannot suspend themselves (e.g. no `sleep()` primitive) and tasks are restarted from their entry point on each invocation. The only resource locking protocol that is supported is IIP, and the scheduling policy is fixed priority scheduling. These limitations makes it possible to build an RTOS that is able to run, e.g., 10 tasks using less that 200 bytes of RAM, and at the same time giving predictable timing behaviour [App98]. Other commercial systems that follow a similar principle of reducing the degrees of freedom and hence promote stringent design of predictable RTSs include Arcticus Systems' Rubus OS [Arc].

Many of the commercial RTOSes provide standard APIs. One of the most important RTOS-standard is RT-POSIX [IEE98]. The POSIX standard is based on Unix, and its goal is portability of applications at the source code level. The basic POSIX services include task and thread management, file system management, input and output, and event notification via signals. The POSIX real-time interface defines services facilitating concurrent programming and providing predictable timing behaviour. Concurrent programming is supported by synchronisation and communication mechanisms that allow predictability. Predictable timing behaviour is supported by preemptive fixed priority scheduling, time management with high resolution, and virtual memory management. Several restricted subsets of the standard intended for different types of systems have been defined, as well as specific language bindings, e.g. for Ada [ISO95].

Within the avionics domain the APEX standard [Air96] has been developed. The goal of APEX is to allow analysable safety critical real-time applications to be implemented, certified and executed. APEX is not an RTOS per se, rather it is to be

seen as a system layer providing a mapping between application and OS services. The scope of the APEX layer is intended to provide the minimum functionality required by an embedded avionics application. APEX RTOSes are mandated to use static cyclic scheduling.

Another domain specific standard is the OSEK standard [Gro]. The objective of the standard is to describe an environment which supports efficient utilisation of resources for automotive control applications. The standard contains a set of APIs with support for traditional RTOS functions and network management. The typical applications that use OSEK have tight real-time constraints and an high criticality. In addition, these applications are usually produced in high volumes. Therefore, in order to save on production costs, there is a strong push towards the optimisation of the application, by reducing the memory footprint to a minimum. Besides standardised APIs OSEK also provides a language for off-line system configuration, OIL. The off-line configuration ability facilitates the production of small footprint systems.

7.4 Real-Time Scheduling

In this section we will give an overview of the more common types of schedulers used in real-time systems.

Traditionally, real-time schedulers are divided into *offline* and *online* schedulers. Offline schedulers are making all scheduling decisions before the system is executed. At runtime a simple dispatcher is used to activate tasks according to an off-line generated schedule. Online schedulers, on the other hand, decide during execution, based on various parameters, which task should execute at any given time.

As there are loads of different schedulers developed in the research community, we have in this section focused on highlighting the main categories of schedulers that are readily available in existing RTOSes.

7.4.1 Introduction to Scheduling

A real-time system consists of a set of real-time programs, which in turn consists of a set of *tasks*. These tasks are sequential pieces of code, executing on a platform with limited resources. The tasks have different timing properties, e.g., *execution times*, *periods*, and *deadlines*. Several tasks can be allocated to a single processor. The scheduler decides, at each moment, which task to execute.

A real-time system can be *preemptive* or *non preemptive*. In a preemptive system, tasks can preempt each other, letting the task with the highest priority execute. In a non preemptive system a task that has been allowed to start will execute until its completion. The difference between pre-emptive and non pre-emptive execution is shown in Figure 2. Here, two tasks, task A and task B, are executing on a node. Task

A has higher priority than task B. Task B arrives before task A. Scenarios for both non pre-emptive execution (a) and pre-emptive execution (b) are shown in the figure.

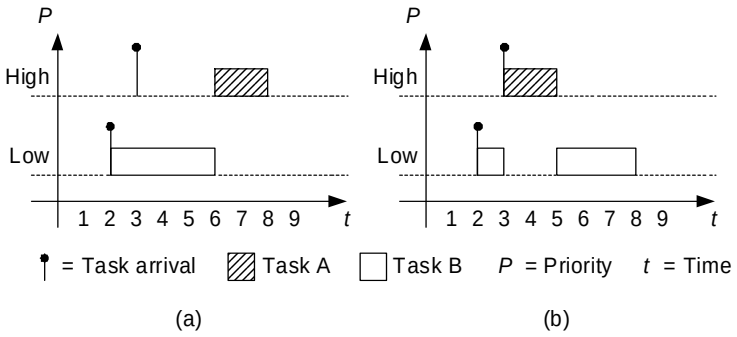


Figure 2: Task execution in a non pre-emptive (a) and a pre-emptive (b) system

Tasks can be categorised into either being *periodic*, *sporadic*, or *aperiodic*. Periodic tasks are executing with a specified time (period) between task releases. Aperiodic tasks have no information saying when the task is to released. Usually aperiodics are triggered by interrupts. Similarly, sporadic tasks have no period, but in contrast with aperiodics, sporadic tasks have a known minimum time between releases. Typically, tasks that perform measurements are periodic, collecting some value(s) every *n*th time unit. A sporadic task is typically reacting to an event/interrupt that we know has a minimum inter-arrival time, e.g., an alarm or the emergency shut down of a production robot. The minimum inter-arrival time can be constrained by physical laws, or it can be enforced by some hardware mechanism. If we do not know the minimum time between two consecutive events, we must classify the event-handling task to be aperiodic. Hence, the difference between sporadic and aperiodic tasks is that we have a known minimum inter-arrival time for the sporadic tasks, whereas the aperiodic tasks have no known inter-arrival time. In Figure 3 the periodic task has a period equal to 2, i.e., inter-arrival time is 2, the sporadic task has a minimum inter-arrival time of 1, and the aperiodic task has no known inter-arrival time.

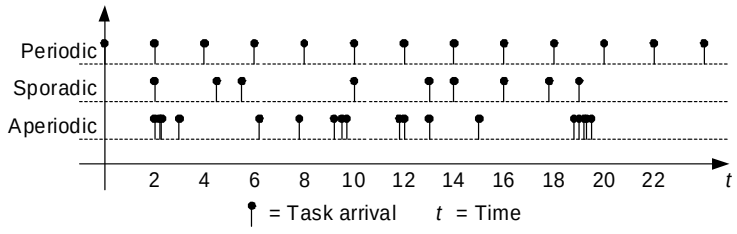


Figure 3: Periodic, sporadic, and aperiodic task arrival

A real-time *scheduler* schedules the real-time tasks sharing the same resource (e.g. a

CPU or a network link). The goal of the scheduler is to make sure that the timing requirements of these tasks are satisfied. The scheduler decides, based on the task timing properties, which task is to execute or to use the resource.

7.4.2 Offline Schedulers

Offline schedulers, or table-driven schedulers, works the following way: The schedulers create a schedule (the table) before the system is started (offline). At runtime, a *dispatcher* follows the schedule, and makes sure that tasks are only executing at their predetermined time slots (according to the schedule). Offline schedules are commonly used to implement the time-triggered execution paradigm.

By creating a schedule offline, complex timing constraints can be handled in a way that would be difficult to do online. The schedule that is created is the schedule that will be used at runtime. Therefore, the online behaviour of a table-driven scheduler is very deterministic. Because of this determinism, table-driven schedulers are the more commonly used schedulers in applications that have very high safety-critical demands, e.g. avionics. However, since the schedule is created offline, the flexibility is very limited, in the sense that as soon as the system will change (due to, e.g. adding of functionality or change of hardware), a new schedule has to be created and given to the dispatcher. To create new schedules is non-trivial and sometimes very time consuming.

There also exist combinations of the predictable table-driven scheduling and the more flexible priority-based schedulers, and there exists methods to convert one policy to another [FLD03, Arc, MTS02].

7.4.3 Online Schedulers

Scheduling policies that make their scheduling decisions during runtime are classified as *online schedulers*. These schedulers make their scheduling decisions based on some task properties, e.g. task priority. Schedulers that base their scheduling decisions based on task priorities are called *priority-based schedulers*.

Priority-based Schedulers Using priority-based schedulers the flexibility is increased (compared to table-driven schedulers), since the schedule is created online, based on the currently active task's constraints. Hence, priority-based schedulers can cope with changes in work-load and added functions, as long as the schedulability of the task-set is not violated. However, the exact behaviour of priority-based schedulers is harder to predict compared to the behaviour of table-driven schedulers. Therefore, these schedulers are not used as often in the most safety-critical applications.

Two common priority-based scheduling policies are *Fixed Priority Scheduling (FPS)* and *Earliest Deadline First (EDF)*. The difference between these scheduling policies

is whether the priorities of the real-time tasks are fixed or if they can change during execution (i.e. they are dynamic).

In FPS, priorities are assigned to the tasks before execution (offline). When executing the system, the task with the highest priority among all tasks that are available for execution is scheduled for execution. It can be proven that some priority assignments are better than others. For instance, for a simple task model with strictly periodic non-interfering tasks with deadlines equal to the period of the task, a *Rate Monotonic (RM)* priority assignment has been shown by Liu and Layland [LL73] to be optimal. In RM, the priority is assigned based on the period of the task. The shorter the period is, the higher priority will be assigned to the task.

Using EDF, the task with the nearest (earliest) deadline among all available tasks is selected for execution. Therefore the priority is not fixed, it changes with time. It has been shown that EDF is an optimal dynamic priority scheme [LL73].

Scheduling with Aperiodics In order for the priority-based schedulers to cope with aperiodic tasks, different service methods have been presented. The objective of these service methods is to give a good average response-time for aperiodic requests, while preserving the timing properties of periodic and sporadic tasks. These services are implemented using special *server* tasks. In the scheduling literature many types of servers are described. Using FPS, for instance, the *Sporadic Server (SS)* is presented by Sprunt *et al.* [SSL89]. SS has a fixed priority chosen according to the RM policy. Using EDF, *Dynamic Sporadic Server (DSS)* [SB94, SB96] extends SS. Other EDF-based schedulers are the *Constant Bandwidth Server (CBS)*, presented by Abeni [AB98], and the *Total Bandwidth Server (TBS)* by Spuri and Buttazzo [SB94, SBS95]. Each server is characterised partly by its unique mechanism for assigning deadlines, and partly by a set of variables used to configure the server. Examples of such variables are bandwidth, period, and capacity.

In section 7.2 we give examples of how timing properties of FPS can be calculated.

7.5 Real-Time Communications

As real-time systems become distributed, data needs to be transmitted between the system nodes. In many cases, this transmission must be timely and deterministic. Hence, there is a demand for real-time communication networks providing real-time guarantees. There are real-time communication networks of different types, ranging from small fieldbus-based control systems to large Ethernet/Internet distributed applications. There is also a growing interest for wireless solutions.

In this section we give an introduction to common techniques in communications in general and real-time communications in particular. We also provide an overview of the most popular real-time communication systems and protocols used today, both in

industry and academia.

7.5.1 Communications Techniques

Common access mechanisms used in communication networks are *CSMA/CD (Carrier Sense Multiple Access/Collision Detection)*, *CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance)*, *TDMA (Time Division Multiple Access)*, *Tokens*, *Central Master*, and *Mini Slotting*. These techniques are all used both in real-time and non real-time communication, and each of the techniques have different timing characteristics.

In CSMA/CD, collisions between messages are detected, causing the messages involved in the collision to be retransmitted, often in a very non deterministic way. CSMA/CD is used e.g. in Ethernet. CSMA/CA, on the other hand, is avoiding collisions and is therefore more deterministic in its behaviour compared to CSMA/CD. Hence, CSMA/CA is more suitable for hard real-time guarantees, whereas CSMA/CD can provide soft real-time guarantees. Examples of networks that implement CSMA/CA are CAN and ARINC 629.

TDMA is using time to achieve exclusive usage of the network. Messages are sent at predetermined instances in time. Hence, the behaviour of TDMA-based networks is very deterministic, i.e., very suitable to provide real-time guarantees. One example of a TDMA-based real-time network is TTP.

An alternative way of eliminating collisions on the network is to use tokens. In token based networks only the owner of the (unique within the network) token is allowed to send messages on the network. Once the token holder is done or has used its allotted time the token is passed to another node. Tokens are used in e.g. Profibus.

It is also possible to eliminate collisions by letting one node in the network be the master node. The master node is controlling the traffic on the network, and it decides how and when messages are allowed to be sent. This approach is used in e.g. LIN and TTP/A.

Finally, mini slotting can be used to eliminate collisions. When using mini slotting, as soon as the network is idle and some node would like to transmit a message, the node has to wait for a unique (for each node) time before sending any messages. If there are several competing nodes wanting to send messages, the node with the longer waiting time will see that there is another node that already has started its transmission of a message. In such a situation the node has to wait until the next time the network will become idle. Hence, collisions are avoided. Mini slotting can be found in e.g. FlexRay and ARINC 629.

7.5.2 Fieldbuses

Fieldbuses are a family of factory communication networks that have evolved as a response to the demand to reduce cabling costs in factory automation systems. By moving from a situation in which every controller has its own cables connecting the sensors to the controller (parallel interface), to a system with a set of controllers sharing a bus (serial interface), costs could be cut and flexibility could be increased. Pushing for this evolution of technology was both the fact that the number of cables in the system increased as the number of sensors and actuators grew, together with controllers moving from being specialized with their own microchip, to sharing a microprocessor with other controllers. Fieldbuses were soon ready to handle the most demanding applications on the factory floor.

Several fieldbus technologies, usually very specialized, were developed by different companies to meet the demands of their applications. Fieldbuses used in the automotive industry are, e.g. CAN, TT-CAN, TTP, LIN, and FlexRay. In avionics, ARINC 629 is one of the more used communication standards. Profibus is widely used in automation and robotics, while in trains TCN and WorldFIP are very popular communication technologies. We will now present each of these fieldbuses in some more detail, outlining key features and specific properties.

Controller Area Network (CAN) The Controller Area Network (CAN) [CAN02] was standardized by the International Standardisation Organisation (ISO) [CAN92] in 1993. Today CAN is a widely used fieldbus, mainly in automotive systems but also in other real-time applications, e.g. medical equipment. CAN is an event-triggered broadcast bus designed to operate at speeds of up to 1 Mbps. CAN is using a fixed-priority based arbitration mechanism that can provide timing guarantees using Fixed-Priority Scheduling (FPS) type of analysis (Tindell *et al.* [TBW95, THW94]).

CAN is a collision-avoidance broadcast bus, using deterministic collision resolution to control access to the bus (so called CSMA/CA). The basis for the access mechanism is the electrical characteristics of a CAN bus, allowing sending nodes to detect collisions in a non-destructive way. By monitoring the resulting bus value during message arbitration, a node detects if there are higher priority messages competing for the access to the bus. If this is the case, the node will stop the message transmission, and try to retransmit the message as soon as the bus becomes idle again. Hence, the bus is behaving like a priority-based queue.

Time-Triggered CAN (TT-CAN) Time-Triggered communication on CAN (TT-CAN) [TTC] is a standardised session layer extension to the original CAN. In TT-CAN, the exchange of messages is controlled by the temporal progression of time, and all nodes are following a predefined static schedule. It is also possible to support original event-triggered CAN traffic together with the time-triggered traffic. This traffic is

sent in dedicated arbitration windows, using the same arbitration mechanism as native CAN.

The static schedule is based on a time division (TDMA) scheme, where message exchanges may only occur during specific time slots or in time windows. Synchronization of the nodes is either done using a clock synchronization algorithm, or by periodic messages from a master node. In the latter case, all nodes in the system are synchronizing with this message, which gives a reference point in the temporal domain for the static schedule of the message transactions, i.e., the master's view of time is referred to as the network's global time.

TT-CAN appends a set of new features to the original CAN, and being standardised, several semiconductor vendors are manufacturing TT-CAN compliant devices.

Flexible Time-Triggered CAN (FTT-CAN) Flexible Time Triggered communication on CAN (FTT-CAN) [AFF98, AFF99] provides a way to schedule CAN in a time-triggered fashion with support for event-triggered traffic as well. In FTT-CAN, time is partitioned into Elementary Cycles (ECs) which are initiated by a special message, the Trigger Message (TM). This message triggers the start of the EC and contains the schedule for the time-triggered traffic that shall be sent within this EC. The schedule is calculated and sent by a master node. FTT-CAN supports both periodic and aperiodic traffic by dividing the EC into two parts. In the first part, the asynchronous window, the aperiodic messages are sent, and in the second part, the synchronous window, traffic is sent in a time-triggered fashion according to the schedule delivered by the TM. FTT-CAN is still mainly an academic communication protocol.

Time-Triggered Protocol (TTP) TTP/C [TTT, TTT99], or the Time Triggered Protocol Class C, is a TDMA based communication network intended for truly hard real-time communication. TTP/C is available for network speeds of up to 25 Mbps. TTP/C is part of the Time Triggered Architecture (TTA) by Kopetz *et al.* [TTT, Kop98], which is designed for safety-critical applications. TTP/C has support for fault tolerance, clock synchronization, membership services, fast error detection, and consistency checks. Several major automotive companies are supporting this protocol, e.g. Audi, VW and Renault.

For the less hard real-time systems (e.g. soft real-time systems), there exists a scaled-down version of TTP/C called TTP/A [TTT].

Local Interconnect Network (LIN) LIN [LIN], or the Local Interconnect Network, was developed by the LIN Consortium (including Audi, BMW, Daimler Chrysler, Motorola, Volvo, and VW) as a low cost alternative for small networks. LIN is cheaper than e.g. CAN. LIN is using the UART/SCI interface hardware, and transmission speeds are possible up to 20 Kbps. Among the nodes in the network, one node is

the master node, responsible for synchronization of the bus. The traffic is sent in a time-triggered fashion.

FlexRay FlexRay [BBE+02] was proposed in 1999 by several major automotive manufacturers, e.g. Daimler-Chrystler and BMW, as a competitive next generation fieldbus replacing CAN. FlexRay is a real-time communication network that provides both synchronous and asynchronous transmissions with network speeds up to 10 Mbps. For the synchronous traffic FlexRay is using TDMA, providing deterministic data transmissions with a bounded delay. For the asynchronous traffic mini-slotting is used. Compared with CAN, FlexRay is more suitable for the dependable application domain, by including support for redundant transmission channels, bus guardians, and fast error detection and signalling.

ARINC 629 For avionic and aerospace communication systems, the ARINC 429 [ARI99] standard and its newer ARINC 629 [ARI99] successor are the most commonly used communication systems used today. ARINC 629 supports both periodic and sporadic communication. The bus is scheduled in bus cycles, which in turn are divided in two parts. In the first part periodic traffic is sent, and in the second part the sporadic traffic is sent. The arbitration of messages is based on collision avoidance (i.e. CSMA/CA) using mini-slotting. Network speeds are as high as 2 Mbps.

Profibus Profibus [PRO] is used in process automation and robotics. There are three different versions of Profibus: (1) Profibus - DP is optimised for speed and low cost. (2) Profibus - PA is designed for process automation. (3) Profibus - FMS is a general purpose version of Profibus. Profibus provides master-slave communication together with token mechanisms. Profibus is available with data rates up to 12 Mbps.

Train Communication Network (TCN) The Train Communication Network (TCN) [KZ01] is widely used in trains, and it is implementing the IEC 61275 standard as well as the IEEE 1473 standard. TCN is composed of two networks: the Wire Train Bus (WTB) and the Multifunction Vehicle BUS (MVB). The WTB is the network used to connect the whole train, i.e. all vehicles of the train. Network data rate is up to 1 Mbps. The MVB is the network used within one vehicle. Here the maximum data rate is 1.5 Mbps.

Both the WTB and the MVB are scheduled in cycles called basic periods. Each basic period consists of a periodic phase and a sporadic phase. Hence, there is a support for both periodic and sporadic type of traffic. The difference between the WTB and the MVB (apart from the data rate) is the length of the basic periods (1 or 2 ms for the MVB and 25 ms for the WTB).

WorldFIP The WorldFIP [Wor] is a popular communication network in train control systems. WorldFIP is based on the producer-distributor-consumers (PDC) communication model. Currently, network speeds are as high as 5 Mbps. The WorldFIP protocol defines an application layer that includes PDC- and messaging-services.

7.5.3 Ethernet for Real-Time Communication

In parallel with the search for the holy grail of real-time communication, Ethernet has established itself as the de facto standard for non real-time communication. Comparing networking solutions for automation networks and office networks, fieldbuses was the choice for the former. At the same time, Ethernet developed as the standard for office automation, and due to its popularity, prices on networking solutions dropped. Ethernet is not originally developed for real-time communication since the original intention with Ethernet is to maximize throughput (bandwidth). However, nowadays, due to its popularity, a big effort is being made in order to provide real-time communication using Ethernet. The biggest challenge is to provide real-time guarantees using standard Ethernet components.

The reason why Ethernet is not very suitable for real-time communication is its handling of collisions on the network. Several proposals to minimise or eliminate the occurrence of collisions on Ethernet have been proposed. Below we present some of these proposals.

TDMA A simple solution would be to eliminate the occurrence of collisions on the network. This has been explored by, e.g. Kopetz *et al.* [KDKM89], using a TDMA protocol on top of Ethernet.

Usage of Tokens Another solution to eliminate the occurrence of collisions is the usage of tokens. Token-based solutions [VC94, PMBL95] on the Ethernet also eliminates collisions, but is not compatible with standard hardware.

A token based communication protocol is a way to provide real-time guarantees on most types of networks. This since they are deterministic in their behaviour, although a dedicated network is required. That is, all nodes sharing the network must obey the token protocol. Examples of token-based protocols are the Timed Token Protocol (TTP) [MZ94] and the IEEE 802.5 Token Ring Protocol.

Modified Collision Resolution Algorithm A different approach is to modify the collision resolution algorithm [RY94, Mol94]. Using standard Ethernet controllers, the modified collision resolution algorithm is non deterministic. In order to make a deterministic modified collision resolution algorithm, a major modification of the Ethernet controllers is required [LR93].

Virtual Time and Window Protocols Another solution to real-time communication using Ethernet is the usage of the Virtual Time CSMA (VTCSMA) [MK85, ZR86, EDES90] protocol, where packets are delayed in a deterministic way in order to eliminate the occurrence of collisions. Moreover, Window Protocols [ZSR90] are using a global window (synchronized time interval) that also remove collisions. The window protocol is more dynamic and somewhat more efficient in its behaviour compared to the VTCSMA approach.

Master/Slave A fairly straight-forward way of providing real-time traffic on Ethernet is by using a master/slave approach. As part of the Flexible Time-Triggered (FTT) framework [APF02], FTT-Ethernet [PAG02] is proposed as a master/multi-slave protocol. At the cost of some computational overhead at each node in the system, timely delivery of messages on Ethernet is provided.

Traffic Smoothing The most recent work, without modifications to the hardware or networking topology (infrastructure), is the usage of traffic smoothing. Traffic smoothing can be used to eliminate bursts of traffic [KSW00, CCLM02] which have severe impact on the timely delivery of message packets on the Ethernet. By keeping the network load below a given threshold, a probabilistic guarantee of message delivery can be provided. Hence, traffic smoothing could be a solution for soft real-time systems.

Black Bursts Black Burst [SK98] is using the fact that if a transmitting station is detecting a collision on the bus, the station will wait some time before it will re-transmit its message again (based on the collision resolution algorithm). However, suppose a station is jamming the network, causing stations to wait for re-transmission. What the jamming station does is that it will send its message right after it has stopped the jamming. If all stations are using unique length jamming signals, then there will always be a unique winner. This is what the black burst approach is using.

Switches Finally, a completely different approach to achieve real-time communication using Ethernet is by changing the infrastructure. One way of doing this is to construct the Ethernet using switches to separate collision domains. By using these switches, a collision free network is provided. However, this requires new hardware supporting the IEEE 802.1p standard. Therefore it is not an as attractive solution for existing networks as, e.g., traffic smoothing.

7.5.4 Wireless Communication

There are no commercially available wireless communication protocols providing real-time guarantees[2]. Two of the more common wireless protocols used today are the IEEE 802.11 (WLAN) and Bluetooth. However, these protocols are not providing the temporal guarantees needed for hard real-time communication. Today, a big effort is being made (as with Ethernet) to provide real-time guarantees for wireless communication, possibly by using either WLAN or Bluetooth.

7.6 Component-Based Design of RTS

Component-Based Design (CBD) of RTS is the main topic of this book. however, in this section we will limit our discussion to consider only the relation of real-time (as described in this chapter) and CBD. A more thorough discussion of the general issues of CBD is provided in Chapter **??** [Chapter 4 CD SW Development].

As stated before, the main challenge of designing real-time systems is the need to consider issues that do not typically apply to general-purpose computing systems. These issues includes:

- constraints on extra-functional properties, such as timing, QoS, and dependability,
- the need to statically predict (and verify) these extra-functional properties,
- scarce resources, including processing power, memory, and communication bandwidth.

In the commercially available component technologies today, there is little or no support for these issues. Also on the academic scene, there are no readily available solutions to satisfactory handle all these issues.

In the remainder of this chapter we will discuss how these issues can be addressed in the context of CBD. In doing so, we also highlight the challenges in designing a CBD-process and component technology for development of RTS.

7.6.1 Timing-Properties and CBD

In general, for systems where timing is crucial there will necessarily be at least some global timing requirements that have to be met. If the system is built from components, this will imply the need for timing parameters/properties of the components and some proof that the global timing requirements are met.

[2] Bluetooth provides real-time guarantees limited to streaming voice traffic.

In section 7.2 we introduced the following four types of timing properties:

- execution time,
- response time,
- end-to-end delay, and
- jitter.

So, how are these related to the use of a component-based design methodology?

Execution Time For a component used in a real-time context, an execution time measure will have to be derived. This is, as discussed in section 7.2, not an easy or satisfactory solved problem. Furthermore, since execution time is inherently dependent on the target hardware, and since reuse is the primary motivation for CBD, it is highly desirable if the execution time for several targets would be available. (Alternatively, that the execution time for new hardware platforms is automatically derivable.)

The nature of the applied component-model may also make execution-time estimation more or less complex. Consider, for instance, a client-server oriented component-model, with a server-component that provides services of different types, as illustrated in figure 4(a).* What does "execution-time" mean for such a component? Clearly, a single execution-time is not appropriate, rather the analysis will require a set of execution times related to servicing different requests. On the other hand, for a simple port-based object component-model [SVK97] in which components are connected in sequence to form periodically executing transactions (illustrated in figure 4(b)), it could be possible to use a single execution time measure, corresponding to the execution time required for reading the values at the input ports, performing the computation, and writing values to the output ports.

Rita om bild 4

Hence, it is not always relevant to reason about the execution-times of components, rather one need to identify the relevant parameter of the timing analysis model used, and relate these to specific component executions. A component model with relatively direct correspondence between components and the model used for timing analysis is from this perspective desirable.

Response Time Response times denote the time from invocation to completion of tasks, and response-time analysis is the activity to statically derive response-time estimates.

The first question to ask from a CBD perspective is: What is the relation between a "task" and a "component"?

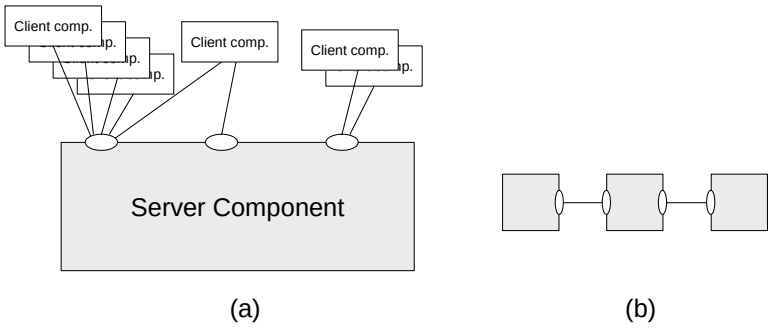


Figure 4: (a) A complex server component, providing multiple services to multiple users, and (b) a simple chain of components implementing a single thread of control.

This is obviously highly related to the component model used. As illustrated in figure 5(a), there could be a 1-to-1 mapping between components and tasks, but in general, several components could be implemented in one task (figure 5(b)) or one component could be implemented by several tasks (figure 5(c)), hence there is a many-to-many relation between components and tasks. In principle, there could even be more irregular correspondence between components and tasks, as illustrated in figure 5(d). Furthermore, in a distributed system there could be a many-to-many relation between components and processing nodes, making the situation even more complicated.

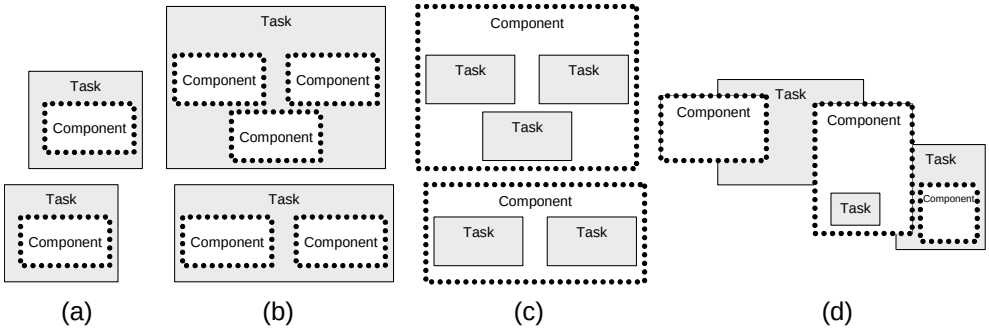


Figure 5: Tasks and components: (a) 1-to-1 correspondence, (b) 1-to-many correspondence, (c) many-to-1 correspondence, (b+c) many-to-many correspondence, and (d) irregular correspondence.

Having sorted out the relation between tasks and components, we can calculate the response times of tasks, given that we have an appropriate analysis method for the used execution paradigm, and that relevant execution time measures are available. However, how to relate these response times to components and the application level timing requirements may not be straightforward, but this is an issue for the subsequent end-to-end analysis.

Another issue with respect to response times is how to handle communication delays in distributed systems. In essence there are two ways to model the communication, as depicted in figure 6. In figure 6(a) the network is abstracted away and the inter-component communication is handled by the framework. In this case, response-time analysis is made more complicated since it must account for different delays in inter-component communication, depending on the physical location of components. In figure 6(b), on the other hand, the network is modelled as a component itself, and network delays can be modelled as delays in any other component (and inter-component communication can be considered instantaneous).

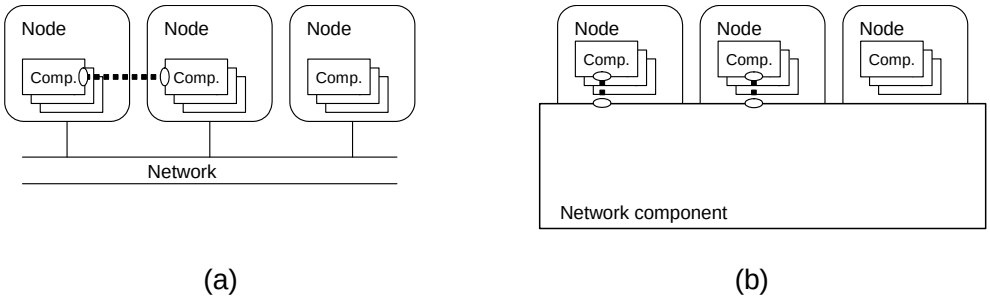


Figure 6: Components and communication delays: (a) communication delays can be part of the inter component communication properties, and (b) communication delays can be timing properties of components.

However, the choice of how to model network delays also has impact on the software engineering aspects of the component model. In figure 6(a), the communication is completely hidden from the components (and the software engineers), hence giving optimising tools many degrees of freedom with respect to component allocation, signal mapping, and scheduling parameter selection. Whereas, in figure 6(b) the communication is explicitly visible to the components (and the software engineers), hence putting a larger burden on the software engineers to manually optimise the system.

End-to-End Delay End-to-end delays are application level timing requirements relating the occurence in time of one event to the occurence of another event. As pointed out above, how to relate such requirements to the lower level timing properties of components discussed above is highly dependent on both the component model and the timing analysis model.

When designing RTS using CBD the component structure gives excellent information about the points of interaction between the RTS and its enviroment. Since, end-to-end delays is about timing estimates and timing requirements on such interactions, CBD gives a natural way of stating timing requirements in terms of signals received or generated. (In traditional RTS development, the reception and generation of signals is

embedded into the code of tasks and are not externally visible, hence making it difficult to relate response-times of tasks to end-to-end requirements.)

Jitter Jitter is an important timing parameter that is related to execution time, and that will affect response-times and end-to-end delays. There may also be specific jitter requirements. Jitter has the same relation to CBD as does end-to-end delay.

Summary of Timing and CBD As described above, there is no single solution for how to apply CBD for RTS. In some cases, timing analysis is made more complicated when using CBD, e.g., when using client-server oriented component-models, whereas in other cases, CBD actually helps timing analysis, e.g., identifying interfaces/events associated with end-to-end requirements is facilitated when using CBD.

Further, the characteristics of the component-model has great impact on the analysability of CBDed RTS. For instance, interaction patterns like client-server does not map well to established analysis methods and makes analysis difficult, whereas pipes-and-filter based patters (such as the port based objects component-model [SVK97]) maps very well to existing analysis methods and allow for tight analysis of timing behaviour. Also, the execution semantics of the component-model has impact on the analysability. The execution semantics gives restrictions on how to map components to tasks, e.g., in the Corba Component-Model [OMG02] each component is assumed to have its own thread of execution, making it difficult to map multiple components to a single thread. On the other hand, the simple execution semantics of pipes-and-filter based models allow for automatic mapping of multiple components to a single task, simplifying timing analysis and making better use of system resources.

7.6.2 Real-Time Operating Systems

There are two important aspects regarding CBD and Real-Time Operating Systems (RTOSes): (1) the RTOS may itself be component based, and (2) the RTOS may support or provide a framework for CBD.

Component Based RTOSes Most RTOSes allow for off-line configuration where the engineer can choose to include or exclude large parts of functionality. For instance, which communications protocols to include is typically configurable. However, this type of configurability is not the same as the RTOS being component based (even though the unit of configuration is often referred to as components in marketing material). For an RTOS to be component based it is required that the components conform to a component model, which is typically not the case in most configurable RTOSes.

There has been some research on component based RTOSes, for instance, the research RTOS VEST [Sta01]. In VEST, schedulers, queue managers and memory management is built up out of components. Furthermore, special emphasis has been put on predictability and analysability. However, VEST is currently still on the research stage and has not been released to the public. Publicly available is, however, the eCos RTOS [Mas02, eCo] which provides a component based configuration tool. Using eCos components the RTOS can be configured by the user, and third party extension can be provided.

RTOSs that Support CBD Looking at component models in general and those intended for embedded systems in particular, we observe that they are all supported by some run-time executive or simple RTOS. Many component technologies provieds frameworks that are independent of the underlying RTOS, and hence, RTOS can be used to support CBD using such an RTOS-independent framework. Examples include Corba's ORB [OMG] and the framework for PECOS [MSZ02, PEC].

Other component technologies have a tighter coupling between the RTOS and component framework, in that the RTOS explicitly supports the component-model by providing the framework (or part of the framework). Such technologies include:

- Koala [**?**] is a component model and architectural description language from Philips, introduced in Section [4.4.2]. As mentioned in that section, Koala provides high-level APIs to the computing and audio/video hardware. The computing layer provides a simple proprietary real-time kernel with priority-driven preemptive scheduling. Special techniques for thread-sharing is used to limit the number of concurrent threads.

- The Chimera RTOS (introduced in Section [4.4.4]) provides an execution framework for the Port-Based-Object component model [**?**], intended for development of sensor-based control systems, specifically reconfigurable robotics applications. Chimera has multiprocessor support, and handles both static and dynamic scheduling, the latter EDF-based.

- Rubus is a RTOS introduced in Section [4.4.3]. Rubus supports a component model in which behaviours are defined by sequeces of port-based objects. The Rubus kernel supports predictable execution of statically scheduled periodic tasks (termed red tasks in Rubus) and dynamically fixed-priority preemptive scheduled tasks (termed Blue). In addition, support for handling interrupts is provided. In Rubus, support is provided for transforming sets of components into sequential chains of executable code. Each such chain is implemented as a single tasks. Support is also provided for analysis of response-times and end-to-end deadlines, based on execution-time measures that have to be provided, i.e., execution time analysis is not provided by the framework.

- The Time-Triggered Operating System (TTOS) is an adapted and extended version of the MARS OS [KDKM89]. Task scheduling in TTOS is based on an off-line generated scheduling table, and relies on the global time base provided by the TTP/C communication system. All synchronization is handled by the off-line scheduling. TTOS, and in general the entire Time Triggered Architecture (TTA) is (just as IEC61131-3) well suited for the synchronous execution paradigm.

  In a synchronous execution the system is considered sequential; computing in each step (or cycle) a global output based on a global input. The effect of each step is defined by a set of transformation rules. Scheduling is done statically by compiling the set of rules into a sequential program implementing these rules and executing them in some statically defined order. A uniform timing bound for the execution of global steps is assumed. In this context, a component is a design level entity.

  TTA defines a protocol for extending the synchronous language paradigm to distributed platforms, allowing distributed components to interoperate, as long as they conform to imposed timing requirements.

- IEC 61131-3 is a standard for Programmable Logic Controllers, introduced in Section [4.4.6].

  Tools and component frameworks for this paradigm...

  <Only proprietary frameworks available. Erik G. will provide example.>

7.6.3 Real-Time Scheduling

Ideally, from a CBD perspective, the response time of a component should be independent of the environment in which it is executing (since this would facilitate reuse of the component). However, this is in most cases highly unrealistic, since

1. the execution time of the task will be different in different target environments, and

2. the response time is additionally dependent on the other tasks competing for the same resources (CPU etc.) and the scheduling method used to resolve the resource contention.

Rather than aiming for the non-achievable ideal, a realistic ambition could be to have a component model and framework which allows for analysis of response times based on abstract models of components and their compositions. Time-triggered systems goes one step towards the ideal solution, in that components can be timely isolated from each other. While not having a major impact on the component model, time-triggered

systems simplify implementation of the component framework since all synchronisation between components is resolved off-line. Also, from a safety perspective, the time-triggered paradigm gives benefits in that it reduces the number of possible execution scenarious (due to the static order of execution of components and due to the lack of preemtion).

Also, in time-triggered component models it is possible to use the structure given by the component composition to synthesise scheduling parameters. For instance, in Rubus [Arc] and TTA [KB03] this is already done today, by generating the static schedule using the components as schedulable entities.

In theory, a similar approach could be used also for dynamically scheduled systems; using a scheduler/task configuration-tool to automatically derive mappings of components to tasks and scheduling parameters (such as priorities or deadlines) for the tasks. However, this approach is still on the research stage.

7.7 Research challenges

<Here we should summarize the main research challenges>

The main challenges for CBD for RTS is mapping component models to target environments, framework, and analysis models.

<Need the highlights here>

References

[AB98] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS'98)*, pages 4–13, Madrid, Spain, December 1998. IEEE Computer Society.

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[Abs] AbsInt. http://www.absint.com.

[AFF98] L. Almeida, J. A. Fonseca, and P. Fonseca. Flexible Time-Triggered Communication on a Controller Area Network. In *Proceedings of the Work-In-Progress Session of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, Spain, December 1998. IEEE Computer Society.

[AFF99] L. Almeida, J. A. Fonseca, and P. Fonseca. A Flexible Time-Triggered Communication System Based on the Controller Area Network: Experimental Results. In *Proc. IFAC Int. Conf. on Filedbus Technology (FeT)*, 1999.

[Air96] Airlines Electronic Engineering Committee (AEEC). ARINC 653: Avionics Application Software Standard Interface (Draft 15), June 1996.

[APF02] L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transaction on Industrial Electronics*, 49(6), December 2002.

[App98] Northern Real-Time Applications. Total time predictability, 1998. Whitepaper on SSX5.

[Arc] Arcticus Systems. The Rubus Operating System. http://www.arcticus.se.

[ARI99] ARINC/RTCA-SC-182/EUROCAE-WG-48. Minimal Operational Performance Standard for Avionics Computer Resources, 1999.

[Art03] Roadmap - Adaptive Real-Time Systems for Quality of Service Management. ARTIST - Project IST-2001-34820, May 2003. http://www.artist-embedded.org/Roadmaps/.

[Ast] The Asterix Real-Time Kernel. http://www.mrtc.mdh.se/projects/asterix/.

[BBE+02] R. Belschner, J. Berwanger, C. Ebner, H. Eisele, S. Fluhrer, T. Forest, T. Führer, F. Hartwich, B. Hedenetz, R. Hugel, A. Knapp, J. Krammer, A. Millsap, B. Müller, M. Peller, and A. Schedl. FlexRay – Requirements Specification, April 2002. http://www.flexray-group.com.

[Bou] Bound-T Execution Time Analyzer. http://www.bound-t.com.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[CAN92] Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communications, February 1992. ISO/DIS 11898.

[CAN02] CAN Specification 2.0, Part-A and Part-B. CAN in Automation (CiA), Am Weichselgarten 26, D-91058 Erlangen, 2002. http://www.can-cia.de.

[CCLM02] A. Carpenzano, R. Caponetto, L. LoBello, and O. Mirabella. Fuzzy Traffic Smoothing: an Approach for Real-Time Communication over Ethernet Networks. In *Proceedings of the 4^{th} IEEE International Workshop on Factory Communication Systems (WFCS'02)*, pages 241–248, Västerås, Sweden, August 2002. IEEE Industrial Electronics Society.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. *Volvo Technology Report*, 1:9–19, 1998.

[eCo] eCos Home Page. http://sources.redhat.com/ecos.

[EDES90] M. El-Derini and M. El-Sakka. A Novel Protocol Under A Priority Time Constraint For Real-Time Communication Systems. In *Proceedings of* 2^{nd} *IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'90)*, pages 128–134, Cairo, Egypt, September 1990. IEEE Computer Society.

[FAQ] Comp.realtime FAQ. available at http://www.faqs.org/faqs/realtime-computing/-faq/.

[FLD03] G. Fohler, T. Lennvall, and R. Dobrin. A Component Based Real-Time Scheduling Architecture. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume LNCS-2677. Springer-Verlag, 2003.

[Gro] OSEK Group. OSEK/VDX Operating System Specification 2.2.1. http://www.-osek-vdx.org/.

[Hal00] Tom R. Halfhill. Embedded Markets Breaks New Ground. *Microprocessor Report*, 17, January 2000.

[IEE98] IEEE. Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP), 1998. IEEE Standard P1003.13-1998.

[IL] I-Logix. Rhapsody. http://www.ilogix.com/products/rhapsody.

[ISO95] ISO. Ada95 Reference Manual, 1995. ISO/IEC 8652:1995(E).

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):112–126, January 2003.

[KDKM89] H. Kopets, A. Damm, C. Koza, and M. Mullozzani. Distributed Fault Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1), 1989.

[Klu] Kluwer. Real-Time Systems (Journal). http://www.wkap.nl/kapis/CGI-BIN/-WORLD/journalhome.htm?0922-6443.

[Kop98] H. Kopetz. The Time-Triggered Model of Computation. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS'98)*, pages 168–177, Madrid, Spain, December 1998. IEEE Computer Society.

[Kop03] H. Kopetz. Introduction In Real-Time Systems: Introduction and Overview. Part XVIII of Lectures Notes from ESSES 2003 - European Summer School on Embedded Systems, Västerås, Sweden, September 2003.

[KRP+98] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M.G. Harbour. *A Practitioners Handbook for Rate-Monotonic Analysis*. Kluwer, 1998.

[KSW00] S. K. Kweon, K. G. Shin, and G. Workman. Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing. In *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, pages 90–100, Washington DC, USA, June 2000. IEEE Computer Society.

[KZ01] H. Kirrmann and P. A. Zuber. The IEC/IEEE Train Communication Network. *IEEE Micro*, 21(2):81–92, March/April 2001.

[LIN] LIN. Local Interconnect Network. http://www.lin-subbus.de.

[Liv99] LiveDevices. Realogy Real-Time Architect, SSX5 Operating System, 1999. http://www.livedevices.com/realtime.shtml.

[LL73] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[Log] Express Logic. Threadx. http://www.expresslogic.com.

[LR93] G. Lann and N. Riviere. Real-Time Communications over Broadcast Networks: the CSMA/DCR and the DOD-CSMA/CD Protocols. Technical report, TR 1863, INRIA, 1993.

[LYN] Lynuxworks. http://www.lynuxworks.com.

[MAS] MAST home-page. http://mast.unican.es/.

[Mas02] A. Massa. *Embedded Software Development with eCos*. Number ISBN: 0130354732. Prentice Hall, November 2002.

[MHD01] J.L. Medina, M. González Harbour, and J.M. Drake. MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems. In *Proc. 22^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 2001.

[MK85] M. Molle and L. Kleinrock. Virtual Time CSMA: Why Two Clocks are Better than One. *IEEE Transactions on Communications*, 33(9):919–933, 1985.

[Mol94] M. Molle. A New Binary Logarithmic Arbitration Method for Ethernet. Technical report, TR CSRI-298, CRI, University of Toronto, Canada, 1994.

[MSZ02] P. O. Müller, C. M. Stich, and C. Zeidler. *Building Reliable Component-Based Software Systems*, chapter Component Based Embedded Systems, pages 303–323. Artech House publisher, 2002. ISBN 1-58053-327-2.

[MTS02] J. Mäki-Turja and M. Sjödin. Combining Dynamic and Static Scheduling in Hard Real-Time Systems. Technical Report MRTC no. 71, Mälardalen Real-Time Research Centre (MRTC), October 2002.

[MZ94] N. Malcolm and W. Zhao. The Timed Token Protocol for Real-Time Communication. *IEEE Computer*, 27(1):35–41, January 1994.

[OMG] OMG. CORBA Home Page. http://www.omg.org/corba/.

[OMG02] OMG. CORBA Component Model 3.0, June 2002. http://www.omg.org/-technology/documents/formal/components.htm.

[OMG03a] OMG. UML Profile for Schedulability, Performance and Time Specification, September 2003. OMG document formal/2003-09-01.

[OMG03b] OMG. Unified Modeling Language (UML), Version 1.5, 2003. http://www.omg.-org/technology/documents/formal/uml.htm.

[PAG02] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency. In *Proceedings of the* 14^{th} *Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 152–160, Vienna, Austria, June 2002. IEEE Computer Society.

[PEC] PECOS Project Web Site. http://www.pecos-project.org.

[PMBL95] D. W. Pritty, J. R. Malone, S. K. Banerjee, and N. L. Lawrie. A Real-Time Upgrade for Ethernet Based Factory Networking. In *Proceedings of IECON'95*, pages 1631–1637, 1995.

[PRO] PROFIBUS. PROFIBUS International. http://www.profibus.com.

[QNX] QNX Software Systems. QNX realtime OS. http://www.qnx.com.

[Rat] Rational. Rational Rose RealTime. http://www.rational.com/products/rosert.

[RTL] List of real-time Linux variants. http://www.realtimelinuxfoundation.org/-variants/variants.html.

[RTS] IEEE Computer Society, Technical Committee on Real-Time Systems Home Page. http://www.cs.bu.edu/pub/ieee-rts/.

[RY94] K. K. Ramakrishnan and H. Yang. The Ethernet Capture Effect: Analysis and Solution. In *Proceedings of* 19^{th} *IEEE Local Computer Networks Conference (LCNC'94)*, pages 228–240, October 1994.

[SB94] M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium (RTSS'94)*, pages 2–11, San Juan, Puerto Rico, December 1994. IEEE Computer Society.

[SB96] M. Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10(2):179–210, March 1996.

[SBS95] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust Aperiodic Scheduling under Dynamic Priority Systems. In *Proceedings of the* 16^{th} *IEEE Real-Time Systems Symposium (RTSS'95)*, pages 210–219, Pisa, Italy, December 1995. IEEE Computer Society.

[SK98] J. L. Sobrinho and A. S. Krishnakumar. EQuB-Ethernet Quality of Service using Black Bursts. In *Proceedings of the* 23^{rd} *IEEE Annual Conference on Local Computer Networks (LCN'98)*, pages 286–296, Lowell, MA, USA, October 1998. IEEE Computer Society.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[Sta01] John A. Stankovic. VEST — A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2211:390–??, 2001.

[SVK97] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12), 1997.

[Sys] Enea OSE Systems. Ose. http://www.ose.com.

[TBW95] K. W. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[TH99] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proc. 20^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 360–369, December 1999.

[THW94] K. Tindell, H. Hansson, and A. Wellings. Analysing Real-Time Communications: Controller Area Network (CAN). In *Proc. 15^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 259–263. IEEE Computer Society Press, December 1994.

[Tima] The Times Tool. http://www.docs.uu.se/docs/rtmv/times.

[Timb] TimeSys. Timewiz - a modeling and simulation tool. http://www.timesys.com/.

[TTC] Road vehicles – Controller area network (CAN) – Part 4: Time triggered comunication. ISO/CD 11898-4.

[TTT] Time Triggered Technologies. http://www.tttech.com.

[TTT99] TTTech Computertechnik AG. Specification of the TTP/C Protocol v0.5, July 1999.

[VC94] C. Venkatramani and T. Chiueh. Supporting Real-Time Traffic on Ethernet. In *Proceedings of 15^{th} IEEE Real-Time Systems Symposium (RTSS'94)*, pages 282–286, San Juan, Puerto Rico, December 1994. IEEE Computer Society.

[Vol] Volcano automotive group. http://www.volcanoautomotive.com.

[Win] Wind River Systems Inc. VxWorks Programmer's Guide. http://www.windriver.-com/.

[Wor] WorldFIP. WorldFIP Fieldbus. http://www.worldfip.org.

[ZR86] W. Zhao and K. Ramamritham. A Virtual Time CSMA/CD Protocol for Hard Real-Time Communication. In *Proceedings of 7^{th} IEEE Real-Time Systems Symposium (RTSS'86)*, pages 120–127, New Orleans, Louisiana, USA, December 1986. IEEE Computer Society.

[ZSR90] W. Zhao, J. A. Stankovic, and K. Ramamritham. A Window Protocol for Transmission of Time-Constrained Messages. *IEEE Transactions on Computers*, 39(9):1186–1203, September 1990.

Verification and Testing Methods

Joel Huselius
Mälardalen Real-Time Research Centre
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
joel.huselius@mdh.se

In this chapter, we discuss issues with testing and debugging performed *implementations* of embedded systems. These issues that we present must be considered and known when performing testing and debugging and when building tools to facilitate them.

1 Outline

The outline of the chapter is as follows: Section 2 will discuss the problems, within the context of testing and debugging an implementation, that are inferred by parallelism and other sources of non-determinism in the implementation. Also solutions that use recording to solve these will be outlined. Thereafter, Section 3 will introduce the issues that must be respected when recording the execution of an implementation. Section 4 will discuss the issue of testing an implementation using recording of the execution, and Section 5 will discuss debugging. Current research challenges in this context are presented in Section 8, whereafter the paper is concluded in Section 9.

2 Testing and debugging implementations that have reproducible executions

According to Thane [20]: An implementation is *deterministic* when a known input always will lead to that the implementation delivers the same output. This requires knowledge of what constitutes the input, which leads to that the implementation must be *observable*. A deterministic implementation is also *reproducible* if the input required to make the execution deterministic can be supplied by the operator of the implementation. In this sense, also interactions with the environment and use of shared resources (scheduling etc.) are to be regarded as input.

Techniques for *regression testing*, debugging, and calculation of *test coverage* are often depending on that the implementation is deterministic or even reproducible.

Before an implementation can be subjected to testing, a bound must be stated for that activity - otherwise the effort would go on endlessly. One feasible way to realize the bound is to state the required *test coverage* of the activity; the required test coverage states how large percentage of the possible executions that must be tested, the actual test coverage is a measurement of how large percentage of the possible executions that have been tested. Of course, in order to calculate test coverage, the total number of executions must be known, wherefore determinism is required.

After that a bug has been sensed in the implementation, the implementation should be debugged. The process of debugging using a debugger (gdb etc.) is an iterative one that restarts the implementation over and over again (with the same input) in order to pinpoint the bug. Hence, a requirement for successful debugging is a reproducible execution.

When the bug has been removed, or an attempt to remove it has been made, the effort of testing must be resumed. As the process of removing the bug may fail to remove the bug or may even introduce new bugs, also the performed testing must be verified - this process is called *regression testing* and requires a reproducible execution.

2.1 Artificial determinisim and reproduciblity

In our view, the functionality of the individual building blocks that constitute the hardware of an implementation are deterministic; if state and all inputs are known, the resulting execution is deterministic.

The problem is that the contents of some input (external input, interrupts, distributed interactions, etc.), or at least some properties of the inputs (timing etc.), are unknown; thus, testing, debugging, and regression testing are infeasible for implementations that rely on them. Examples of irreproducible input are input from the environment and online scheduling: Reading a sensor is a source of non-determinism as the process that controls the value of the sensor is out of control of the software developer. As the processor clock is influencing online scheduling, the same goes for scheduling decisions.

The concept of proposed solutions to the above presented problems is to make more inputs known. By monitoring and then logging unknown input to the execution, the execution becomes deterministic. The execution even becomes reproducible as, in analogy with the record and play functions of a VCR, logged data can be used to replay an execution in a model of the platform.

The remainder of this section will discuss the issues relevant to this record/replay technology.

3 Recording the execution

Monitoring and logging, together labeled *recording* of an execution is performed by introducing *probes* into the implementation. Probes can be implemented in: software (SW), hardware (HW), or some hybrid (HY) of those two; the three implementation strategies can for example be compared with respect to perturbation (SW - high, HW -low), economical-cost (SW - low, HW - high), portability (SW - high, HW - low), and granularity (SW - low, HW - high). Properties of HY-probes are implementation dependent.

Whereas software probes are integrated to run on the same nodes as the part of the implementation that it is monitoring, hardware probes are often designed to snoop busses, networks, etc. Thus, hardware probes have the potential to monitor the interactions of the system without perturbing the same.

3.1 Overhead of recording

Many implementations of probes (even hardware implementations [25]) will debit the implementation with an overhead. This perturbation can be measured in the form of increased execution time, and in the form of increased memory demand.

Whereas an increase in memory demand can be remedied by assigning more memory resources to the implementation, the overhead in execution time is more difficult to cope with. Further, altering the execution time overhead will lead to that presumptions for scheduling decisions are altered, resulting in changes to the behavior of the implementation that cannot be quantified. This effect on the execution, referred to as the *probe effect* [5], has the same consequence on the implementation as removing a bug - the performed test cases must be attested using regression testing.

3.2 Control- and data-flow

As first described by Platter [14], an execution can be modeled as two flows: *data*- and *control-flow*. The data-flow describes the flow of data between architectural components on some level, the control-flow describes the inter-correlated sequences of instructions executed on all processors relevant to the implementation.

The data-flow is normally very voluminous, it must cover state of, and modifications to, items such as stack-, heap-, and global- data, operating system-, and processor- state. Representing the control-flow can also result is large volumes of data, but abstractions can often be used to reduce this amount. Examples of control-flow abstractions can for example be the ordering, result, and timing of *events* such as interrupts, exceptions, preemptions, etc. The data-flow can often be described by *events* such as communication, reading of sensors, parameterization, etc. However, both control- and data-flow must be monitored with a sufficient degree of detail to satisfy the use of the resulting logs, which at least includes the objective to make the execution deterministic.

3.3 The local correlation problem

Occurencies of events are monitored, and *entries* that describe them are logged into *records*. To understand the context of an individual entry, events must be interrelated to each other. It must therefore be possible to state how much execution resources a task has consumed between two records in the log. We term this the *local correlation problem*. For *synchronous events*, those which occurrence is dictated by the executed code, this can be determined by matching the control-flow with the code of the implementation. This is true for message sending and receiving etc. For *asynchronous events*, such as exceptions, interrupts, etc., other solutions are required. Potentially, the program counter (PC) can be used, but this does not provide

exclusive marking for instructions in loops, recursive functions etc. There are at least two approaches for how to solve it: One is to use a hardware platform which supports instruction counting, cycle counting or similar, the other is to use some software implementation.

Mellor-Crummey and LeBlanc [11], present a software implementation to instrument assembler-code with counters, thus enabling the counting of executed instructions, the method is called Software Instruction Counter (SIC). The authors note that the code of a program consists of short sequences of sequential code, called basic blocks, and conditional, or unconditional, connections between some of the basic blocks (by branches, jumps, or function calls). These one-way connections can either connect a basic block with a later (with higher address-value than the present), a forward branch, or with a prior block, a backward branch. The authors state that a combination of the program counter value and the number of backward-branches performed by the execution to reach the instruction from a known starting point is sufficient to uniquely mark each instruction instance that is executed. They can therefore construct a software-based instruction counter which only resource requirements except a small computation overhead is a reserved data-register which is used solely for performance reasons.

The main drawback of the software instruction counter is the temporal overhead; to instrument each branch in the assembler code will lead to large perturbation. A method described by Thane et al. [23] tries to reduce the overhead of software implementations by using checksums of dynamically modified data (e.g. stack and/or processor registers) to relate events.

3.4 The global correlation problem

Fidge [4] describe the problem of obtaining a truthful view of the events in an observed implementation. For example, as a distributed system is being observed, if the observer cannot be tightly coupled with the implementation it is observing, problems related to the observers apprehension of the ordering of events on different nodes may occur. Depending on variations in the propagation time of observer notifications, the ordering of events may be confused. We term this the *global correlation problem*.

According to Fidge, the problem can be divided into at least four sub-problems [4]: (1.) multiple observers may see different event orderings, (2.) observers may see incorrect orderings of events, (3.) different executions may yield different event orderings, and (4.) events may have arbitrary event orderings. All are more or less results of the absence of an exact global time-base, and/or the fact that network propagation times are not constant. Because of the lack of a exact global time, we cannot rely on any time-stamp taken at the node where the event occurred, if the observer is situated on another node.

1. In an implementation where many observers are used, different observers may see different event orderings, because the propagation of the event notification requires different time to different destinations.

2. As the propagation through a network may differ between two network packages, a package that is sent after another may arrive earlier. Thus, if two events occur on different nodes at different times, the notification of the last event may arrive at the observer before the first notification has arrived, thus erroneously implying that the last event occurred before the first.

3. Because the clock-rate of each node will diverge slightly from the ideal clock and clocks on other nodes, and the rate of that deviation partly depends on environmental aspects, even different invocations of a distributed system will differ.

4. Some of the events are unrelated, and may therefore be allowed to occur in arbitrary orderings. The problem with this is that an observer must know and recognize that, as different tests are run, it is allowed to have differing orderings between some of the events.

Some of these variations of the problem of observing can be solved by establishing a global timebase. Solutions can be grouped into *clock synchronization* [7] and *logical clocks* [10].

3.5 Failure detection

Important issues when implementing primitives to record an execution are *failure detection* and *log extraction*.

As the log is to be used after that the execution has been completed, there must be an infrastructure for transporting the log from the platform onto persistent storage. In order to maintain the testability, as most mediums require the intrusive operation of interrupts etc., this should not be performed during normal operation of the implementation. Only at the end of

the execution, when the system has experienced a failure or intentional system stop, the extraction of the log can be performed without worrying about testability.

That discussion leads to the issue of failure detection; the system must be aware of that it is dysfunctional and must be aware to act on that knowledge.

4 Testing the implementation

There are several techniques for verifying that the specification is correct (e.g. Times [1]) and that the intermediate steps from specification to implementation are performed correctly (e.g. the Rational Unified Process [9]). However, none of these ensure that the actual implementation is performing the task intended and described by the specification. There is a need for useful tools that take the implementation as input when verifying behavior, this section will elaborate on this subject.

4.1 Testability

Thane and Hansson [21] has determined that jitter, or differences in execution time, is bad for testability: An executed instance of a multi-tasking implementation can be seen as a sequential program, an *execution scenario* is a serialization (compare to database transactions etc.) of a multi-tasking implementation into a single-tasking one. Thus, a multi-tasking implementation can be seen as a set of single-tasking implementation of which all members must be tested as thorough as any sequential system - a larger set requires more testing. The size of the set is increasing with the jitter, wherefore jitter will increase the required testing effort. Thus, the jitter of the system should be reduced.

Previously, Puschner [15, 16] has argued for WCET-oriented programming, i.e. for algorithms in real-time systems to be optimized with regard to reduced jitter rather then reduced average execution time. According to Puschner, the main motivation for WCET-oriented programming is to make WCET-estimations more accurate and even automated, thus making scheduling easier and more efficient, but he also argues that reduced jitter will make control-algorithms function better and increase both predictability and maintainability of the system.

Due to the effects of cashes, pipelines, etc., a constant execution time should be supported by a suitable hardware platform such as that presented by Delvai et al. [3]. However, the negative effect of jitter on testability has exponential characteristics [22], which leads to that even small reductions in jitter will have significant impact on testability.

4.2 The completeness problem

Testing the complete set of possible combinations of known input data and all execution orderings is normally referred to as *exhaustive testing*. As described in the Section 4.1 above, the number of test cases is very large even in small multi tasking implementations, and it increases drastically as the implementation grows. Therefore, as it would require too long time to perform, exhaustive testing is normally not an option; it is only feasible to test a subset of the possible input. The problem of incomplete testing is referred to as the *completeness problem*, which states that only a certain level of confidence can be placed in the correct functionality of the system. The level of confidence relates directly to how well the system was tested. It is true that small parts of the system that are considered as especially important, could be selected for exhaustive testing. This would of course increase the confidence in the system, but is directly comparable to testing only a small subset of the possible input combinations to the system.

4.3 Testing the implementation

Respecting the issues with testability and completeness described above, Thane and Hansson [21] present a method that describes how all possible orderings in a system can be identified, how all sequences of interleaving due to interrupts, blocking by semaphores, or scheduling decisions can be listed. They can then group a particular recorded execution with one of the identified execution orderings. By running a sufficient number of tests and relating each test to its ordering, it is then possible to increase the confidence in the orderings that become subjected to testing.

However, this approach would either cause some of the less probable execution orderings to be insufficiently tested, or excessive testing due to the improbability or probability of experiencing those orderings. Therefore, appropriate distribution in the testing should be ensured by enforcing execution orderings during testing. By performing a sufficient amount of tests of a sufficient number of orderings, the confidence in the system can then be calculated based on the confidence in each ordering. Thane and Hansson state that the number of execution orderings, and therefore also the testability of the system,

is directly proportional to the number of preemption points and the jitter present in the system. Note that the confidence in a system can be a 2-dimensional property, a confidence in each execution ordering, and a confidence in covered execution orderings.

5 Debuging the execution

Once the testing effort has reviled the presence of a bug in the implementation, the bug must be located, quantified, and removed. This section discuss the issue of debugging.

5.1 Categories of failures

Clarke and McDermid provides a classification of different bugs that may occur in sequential implementations [2]:

Control bugs are those that force the task through another path than intended in an if-else statement.

Value bugs may be the assignment of incorrect values to the correct variable.

Addressing bugs assign values to incorrect variables.

Termination bugs concern failure to terminate a loop.

Input bugs could be unintended input values from sensors, or erroneous parameterization.

But also others are possible, for instance, memory leakage may have many causes: One is a control bug that leads to failure to execute the `free()` function when intended. Another is the absence of code; the call to the `free()` function may be absent in the code.

In addition to bugs present also in sequential implementations, the nature of parallel, distributed, and/or multitasking systems give rise to classes of errors that are not visible in sequential systems. Kranzlmüller summarizes in [8] that deadlocks and livelocks are common classes of bugs in these systems. In addition, also problems related to race conditions in the system are potentially present bugs [12]. Thane [20] adds precedence violations to the list. Finally, in real-time systems, also overdraft of timing budgets are potential bugs.

5.2 Replay

Deterministic reproduction of an execution through *record/replay* has previously been presented as plausible means for to remedy the irreproducibility problem [17, 20, 24, 26]: The log from a recorded *reference execution* is used to create a *replay execution* that is intended to be identical to the reference execution. Non-deterministic choices encountered during the replay execution are resolved according to the log.

The main drawback of the method lies in the overhead of the recording; the memory requirements for keeping the log intact can be substantial if the reference execution is long. The solution to which we adhere is to make a *short replay* that starts from a state other then the starting state of the reference execution. The *starting point* of replay is a globally consistent state gathered from checkpoints. A recent publication [18] has shown the applicability of short replay in industrial state-of-practice real-time systems.

As mentioned earlier, debugging is an iterative process that often requires several passes of the execution. As refraining from using short replay would force developers to go through the entire execution at every pass, apart from reducing the memory overhead, using short replay will also facilitate a faster development process [24].

However, there are bounds for how short a short replay can be: Theoretically, the replay must cover the period of time from the infection of the system (the execution of a bug) to the failure of the system (when the presence of the executed bug is sensed by the system-environment) [20]. This period is the *incubation period* of the system. In practice however, it is the amount of memory assigned to the recording effort, and the rate with which this memory is effectively used, that limits the length of the replay.

5.3 Reduced overhead for checkpointing

Taking checkpoints of the system state is, as is all recording, a very resource demanding operation. There are two methods known to us that can be used to reduce the overhead of taking checkpoints, these are not mutually exclusive:

In distributed environments, one way to deal with this problem is to use *uncoordinated checkpointing* that allow checkpointing of distributed entities (task, node, etc.) without coordination. As there is no coordination between entities concerning the timing of checkpoint acquisition, there are no guarantees for the existence of a consistent state. When trying to obtain a starting point for replay by selecting a set of checkpoints, one from each entity, there is a (substantial) risk that a pair of checkpoints in the set are inconsistent. There are two different erroneous scenarios; One scenario is that the checkpoint at the receiving entity represents a state when a particular message cannot not yet have been received, but the checkpoint at the sending entity represent a state when the message must have been sent - i.e. the message is *in transit*. The other scenario is that the checkpoint at the receiving entity represents the state when a message must have been received, but the checkpoint at the sending entity represent a state where the message cannot have been sent - in this case, the message is referred to as an *orphan* message.

The other way to reduce the overhead of taking checkpoints is to reduce the size of the checkpoints by using *memory excluding checkpoints* [13]. That is, to use in-complete checkpoints where deterministic parts of the data have been left out as it can be derived offline. The concept of memory excluding checkpoints is as follows: as the goal of checkpointing is recreation of a previous system state at a later point in time, a checkpointing algorithm is only required to log the data that cannot be deduced by other means.

Plank et al. [13] state that there are two distinct approaches to exclude memory from checkpoints: To omit dead memory, or to omit read-only-memory. The goal of the first approach is to identify the memory that is no longer needed by the application (compare to garbage collection), and exclude this memory from the checkpoint. The goal of the second approach is to exclude the data which has not changed since some known system state (for example another checkpoint, or the initial state of the system).

The challenge that we face is to minimize the size of checkpoints without inferring a jitter into the system. There are currently no transparent methods known to us that can perform the exclusion of memory addresses from checkpoints while keeping a constant execution time. The only work performed in this area [6, 18] has derived memory to exclude by performing a suite of testruns.

6 Regression testing

Regression testing is the process of attest that an attempt to remove a bug from an implementation has not caused unforeseen side effects. As the process of removing the bug may fail to remove the bug or may even introduce new bugs, the performed testing must be verified.

Essentially, regression testing can be performed by re-executing a subset of the performed tests. We are not concerned with the method of selecting the subset of testcases, we limit the scope to the possibility of performing regression tesing - a process that require reproducibility in the implementation; a potentially feasible solution to this problem is to use the old logs collected with the intent to replay an execution for debugging purposes, however, this requires that the new implementation and the logs from the old implementation are compatible. The solutions to this problem must vary slightly depending on how execution resources are measured (PC-value, SIC's, checksums, etc.).

Essentially, some preprocessing must be performed to translate the log from the representation of the old to the new implementation. Once the log has been successfully translated, regression testing can be performed as with reproducible implementations.

7 State of the practice

Today, the industry often use brute force of tackle the lack of reproducible behavior in implementations; test coverage is often measured in hours of testing, and by spending large amounts of time and effort, most observed bugs can be found and removed or circumvented. Whether this approach is favored due to lack of knowledge or as a consequence of a calculated risk-estimation is unclear. When discussing the issue with a leading manufacturer of industrial robots, it seems that the overhead of the recording effort is the single most important reason for this.

There are however some commercial alternatives in use, these are based on special hardware implementations in the form

of logic analyzers etc. (e.g. Lauterbach). Due to the lack of interfaces, these solutions do not allow monitoring of on-chip events, wherefore on-chip caches etc. complicate debugging using tools like these. Other integrated hardware solutions that provide more interfaces, such as JTAG (www.jtag.com) and BDM (www.motorola.com), introduce an execution time overhead during use. A promising family of hardware standards on the rise are clustered in Nexus (www.nexus5001.org), it seems that some implementations of Nexus has the potential to leave the execution of the implementation undisturbed. Further evaluation is however required to ensure this.

8 Challenges in component based systems

In this section, we will outline a set of intresting topics for furture work in the area of testing and debugging component based implementations.

Architectural support for recording It should be possible to reuse the probing technology inserted into components. Is it possible to construct a probe-component that can be a part of every other component?

Observability as a design feature In the context of this chapter, a key requirement for components in the system is the ability to be observable; if it is not possible to record the data- and control-flow of the implementation, it cannot be tested or debugged. However, in an effort to make complex architectures more understandable, current design methodology strives for to hide as much of the intricate details as possible. In a component based system, it may therefore be difficult for the user of a component to determine what to record - the manufacturers of the components must provide these methods.

Thus, we argue that the component model must support recording.

The effect of jitter on testability As described by Thane and Hansson, the number of execution orderings increase with jitter. However, as they assume that the execution time of each task can vary within a continuos interval described by the Best-Case Execution Time (BCET) and the Worst-Case Execution Time, their method to estimate the number of execution orderings in a system will overestimate the number of execution orderings possible.

We need to improve the method to estimate the number of execution orderings in a multi-tasking implementation.

Coverage in execution orderings We need to improve the method for testing multi tasking real-time system proposed by Thane and Hansson [21] by enforcing execution orderings during testing. The appropriate distribution in the testing of execution orderings could be ensured by some modified replay approach.

Reusable replay-technology We need reusable replay solutions that have reduced platform dependence. Also the replay technology should be reusable, most solutions currently known to us have limitations in assumptions about hardware [25], operating system [18], language [19] etc. Solutions are more restrictive in the types of bugs that can be found [26].

Transparent memory exclusion Current implementations for selecting memory to checkpoint are not constructed with testability in mind. The only way known to the authors to compose memory excluding checkpoints without compomising testability is to manually identify the members of the checkpoint offline - a time-consuming process that requires resonable knowledge of the system functionality. Thus, in order to facilitate memory excluding checkpoints in systems with reuse, we need transparent methods for performing memory excluding checkpoints without compromising testability.

9 Conclusions

In this chapter, we have discussed testing and debugging of non-deterministic implementations. It has been made clear that the assumptions (e.g. reproducibility) and tools (e.g. gdb) used to test and debug deterministic implementations are insufficient.

We have explained and provided references to some of the more important issues (probe effect, overhead, correlation, testability, etc.) that must be respected when performing testing and debugging and when building tools to facilitate these activities. To some of these issues, the available solutions have been referenced.

Further, we have elaborated on furure work that should be performed within the scope of the section. It is our intent to persue some of these items of future work within the scope of SAVE.

References

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of the* 8^{th} *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 460–464, April 2002.

[2] S. Clarke and J. McDermid. Software fault trees and weakest preconditions: A comparison and analysis. *Software Engineering Journal*, 8(4):225–236, July 1993.

[3] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR example. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 169–176, July 2003.

[4] C. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

[5] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[6] J. Huselius, D. Sundmark, and H. Thane. Starting conditions for post-mortem debugging using deterministic replay of real-time systems. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 177–184, July 2003.

[7] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *Transactions on Computers*, 36(8):933–940, August 1987.

[8] D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler University of Linz, Austria, September 2000.

[9] P. Kruchten. Tutorial: Introduction to the rational unified process. In *Proceedings of the* 24^{rd} *International Conference on Software Engineering*, page 703, May 2002.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[11] J. Mellor-Crummey and T. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86. ACM, April 1989.

[12] R. Netzer and B. Miller. What are race conditions? - some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[13] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, 1999.

[14] B. Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

[15] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proceedings of the* 8^{th} *IEEE International Workshop on Obj ect-Oriented Real-Time Dependable Systems*, January 2003.

[16] P. Puschner. Hard Real-Time Programming is Different. In *Proceedings of the* 11^{th} *International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.

[17] D. Stewart and M. Gentleman. Non-stop monitoring and debugging on shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263–269. IEEE Computer Society, May 1997.

[18] D. Sundmark, H. Thane, J. Huselius, A. Pettersson, R. Mellander, I. Reiyer, and M. Kallvi. Replay debugging of complex real-time systems: Experiences from two industrial case studies. In *Proceedings of the* 5^{th} *International Workshop on Automated Debugging*, pages 211–222, September 2003.

[19] K.-C. Tai, R. Carver, and E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):280–287, Januari 1991.

[20] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[21] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[22] H. Thane, A. Pettersson, and H. Hansson. Integration testing of fixed priority scheduled real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, December 2001.

[23] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 288–295, April 2003. Presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD03).

[24] N. Thoai, D. Kranzlmüller, and J. Volkert. Shortcut replay: A replay technique for debugging long-running parallel programs. *Lecture Notes in Computer Science*, 2550:34–46, January 2002.

[25] J. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[26] F. Zambonelli and R. Netzer. An efficient logging algorithm for incremental replay of message-passing applications. In *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392–398. IEEE, April 1999.

Chapter 9

Volvo Construction Equipment Components - a Case-Study

Joakim Fröberg

9.1 Introduction

In this chapter, 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.

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. 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 external subsystem vendor and VCE must develop control systems and software at competitive prices as any other vendor.

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.

9.2 Functionality overview

A useful platform for product lines must support the variety of functional requirements that the different products impose. Ideally, the platform should, be able to support a seamless integration of 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 sensors 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 chapter. 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 such as 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, e.g., 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 such as streaming video or Internet access.

9.3 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 designed 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 [Bro96]. In VCE, the architecture was designed with the intent of meeting the identified quality requirements. 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 most important quality requirements on the platform are classified in operational quality requirements and development quality requirements as follows.

Operational quality requirements:

- Reliability and robustness– 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.
- 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 the same 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.

9.4 Application characteristics

Depending on the vehicle, the number of nodes and tasks vary from 2-5 nodes and 80-xyz tasks. The total amount of code for a vehicle is about 800 k lines.

The system hardware for a wheel loader consists of two nodes connected via redundant buses. The application consists of 80 tasks running with period times from 10 ms to 1 s and with execution times ranging from 10 μs to 1 millisecond. These tasks have hard real-time requirements, are scheduled off-line and execute according to a dispatch table on-line. Each node is very I/O intensive, the complete system incorporating 150 I/O channels.

The application is interrupt intensive and the worst case utilization of the processors, for the critical part, is approximately 80%, divided into 35% for interrupts and 45% for hard real-time tasks. The remaining spare capacity is used by on-line scheduled soft real-time tasks. During run-time, the spare capacity will exceed the remaining 20% if the load generated by hard tasks and interrupts is less than the worst case.

9.5 Component model

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

The component model allows hierarchical decomposition, and at the top level 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.

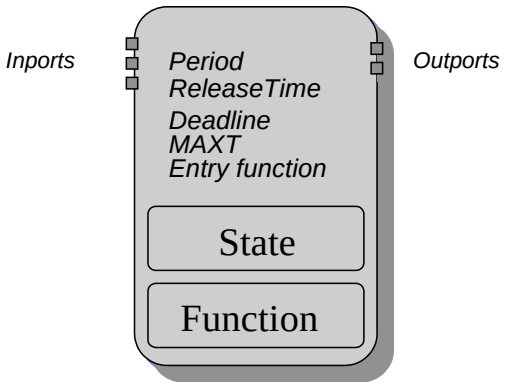


Figure 9.1 *Task model*

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.

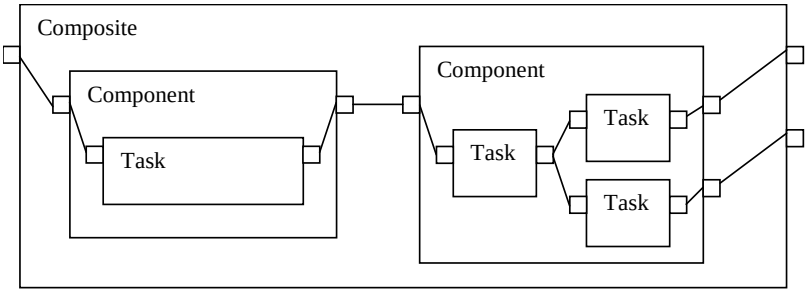


Figure X.3 *Component decomposition*

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:

- Early prediction of timing behavior
- Easier component reuse

9.6 Architectural views

In order to describe the VCE electronic architecture we use several 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.

9.6.1 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.

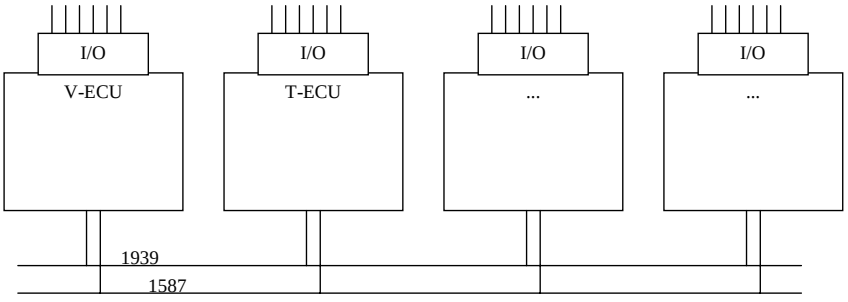


Figure X.5 *Hardware setting*

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.

9.6.2 ECU software view

As depicted in Figure 6, each ECU consists of hardware and two different software layers **Error! Reference source not found.**. The top layer is an application layer with specific functionality for different ECU's and products. The middle layers are the interfaces between the application and the hardware. These layers are the communication layer, which handles communication between different nodes, the I/O layer that handles the in and output on the affected node and the real-time operating system Rubus, which handles task scheduling and synchronization.

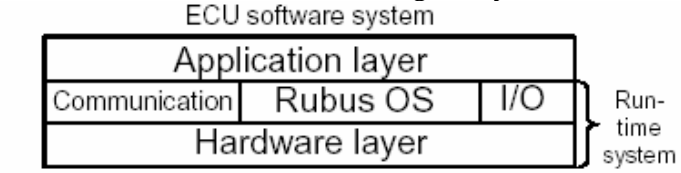


Figure 6. *The Node (ECU) Architecture*

A view of the Rubus RTOS used by Volvo Construction is shown in figure 7. Rubus is a real-time operating system with support for both static scheduling and pre-emptive fixed priority scheduling. The static and pre-emptive fixed priority schedulers are referred to as the *red* and *blue* parts of Rubus respectively. The red part only handles hard, static real-time, while the blue part only handles soft real-time. Finally, there is also a green part of Rubus, which is an interrupt handler kernel. It has the highest priority, and distributes and routes the interrupts.

The red part of Rubus always has higher priority than the blue part. Therefore, the red part is used for time critical operations (firm and hard real-time), since it is easier to verify timing properties of the red part. The blue part of Rubus is usually used for more dynamic properties, and soft real-time. The green part always handles all interrupts, and has the highest priority in the system.

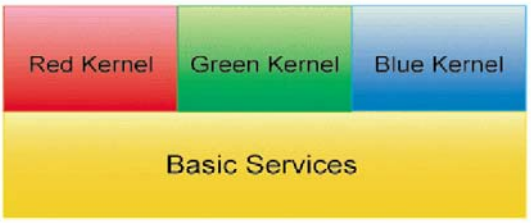


Figure 7. Rubus RTOS Architecture

9.6.3 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.)

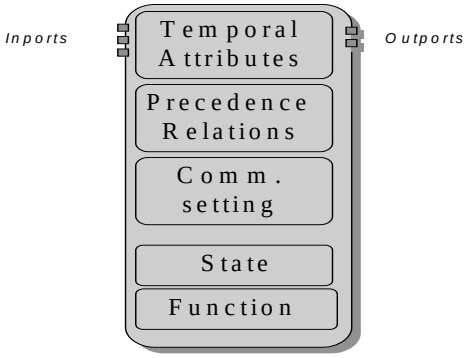


Figure X.6 *Task configuration*

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.

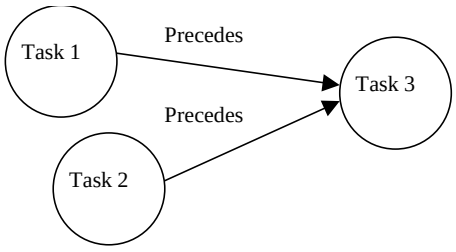


Figure X.7 *Precedence relations*

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 timeliness kept intact [San98].

These are the significant interrupts currently in use in the VCE applications:

1. Interrupt per 1939 (CAN) link
2. Interrupts for the 1587 link
3. Frequency counter interrupts
4. Watchdog timer interrupt

The VCE ECU uses hardware that process the CAN messages and therefore the 1939-interrupt handler does not use a large amount of CPU capacity. The interrupt handler performs only copying of the messages. The 1587-interrupt handlers have to perform more operations and occupy the major part of the total interrupt processing. (Total is 14 % of the CPU capacity) A frequency counter interrupt is extremely light in terms of processing. The interrupt handler only increase a counter and an application task then uses this value together with the task period time to calculate a frequency. The watchdog timer interrupt also uses only a minor fraction of the required interrupt processing capacity.

In order to schedule the task-set for the system, the interrupts must be modeled as red tasks with a period time and an execution time. This provides enough information for the scheduler to generate the system execution schedule. The problems that are related to the scheduling issues are investigated in [San98].

9.6.4 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.

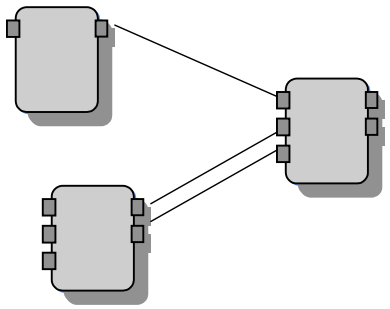


Figure X.8 *Communication flow*

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 [Iso00]. 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.

9.7 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.

Currently VCE uses parameterization and variables as well as optional nodes as techniques for achiving variability in the electronic systems

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.

9.7.1 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, E2. E2 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.

9.7.2 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.

9.7.3 Optional nodes, sensors, actuators, and resources

Parameterization via datasets provides a convenient way to vary 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.

References

[Bro96] Brownsword, L., Clements, P., *A Case Study in Successful Product Line Development*, Technical Report CMU/SEI-96-TR-016, October 1996.

[Dou98] Douglas, B.P., *Real-Time UML - Developing efficient objects for embedded systems*, Addison Wesley Longman, Inc, 1998

[Wall00] Wall, A., *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*, IT Licentiate theses, 2000-004, MRTC Report 00/21.

[San98] Sandström, K., Eriksson, C., and Fohler, G., *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*, In Proc. of the fifth International Conference on Real-Time Computing Systems and Applications, October 1998.

[Iso00] Isovic, D., Lindgren, M., Crnkovic, I., *System Development with Real-Time Components*, In Proc. of ECOOP2000 Workshop 22 - Pervasive Component-based systems, June 2000.