A Requirements Analysis Method for the Evaluation and Selection of Concurrency Constructs

Master Thesis

September 2002, Breght R. Boschker

symbian

RuG
Rijksuniversiteit Groningen

A Requirements Analysis Method for the Evaluation and Selection of Concurrency Constructs

Master Thesis

September 2002, Breght R. Boschker

symbian

RuG
Rijksuniversiteit Groningen

Master Thesis in Software Engineering for the Department of Computing Science, University of Groningen, The Netherlands

September 2002.

Breght Roderick Boschker (b.r.boschker@wing.rug.nl)

Abstract

Throughout the years, the research community has developed a number of different concurrency constructs. All constructs have shown to be able to provide at least logical concurrency, but the constructs have fundamental differences. Selecting a solution that is tailored to a specific problem involves analyzing the domain, requirements and constraints for a given problem. This thesis uses a subset of requirements engineering to define a requirements analysis method that is used to assess the constructs with respect to the problem to obtain the solutions and their suitability for the given problem. In this thesis, the method is used to assess three concurrency solutions: cyclic scheduling, threads, processes and an event-handling framework.
The second part of this thesis uses the method as described to show a way of implementing a telecommunications protocol that is tailored to the nature of protocols, namely by using an event-handling framework.

Acknowledgements

My thanks go to my supervisors, Peter Molin at Symbian AB in Ronneby, Sweden and Jan Bosch at the University of Groningen, The Netherlands. I also thank Peter Karlsson at Telelogic Sverige AB for supplying the *Telelogic Tau SDL and TTCN* software and documentation.

Furthermore, I would like to thank my girlfriend Astrid, family and friends for their everlasting support during my stay in Sweden and during my work for this thesis.

Preface

This thesis was written as partial fulfillment for the requirements for the Master's Degree in Software Engineering at the University of Groningen, The Netherlands. Symbian AB (now UIQ Technology AB) provided the initial task description, technical, practical and human resources and a welcome breakfast.

Table of contents

1 Introduction

1.1 Problem description

Throughout the history of modern computer systems, the development of new technology has lead to ever-increased capabilities of these systems. With this came a change in demand and use: systems would be required to serve multiple users or multiple programs at the same time. This, in addition to the observation that computers that are waiting for input/output operations can be used to do other tasks lead to the development of a number of techniques known as *concurrency*. Throughout the years, a number of methods have been designed for, at least logically, executing multiple programs at the same time.
Although most of these methods are suitable for most general problems, some might be better suited than others. This is especially the case when the chosen concurrency solution plays a key role in the system to be designed.

The aim of this thesis is to look at the assessment of requirements and the analysis of the problem domain for helping in taking (low-level) architectural design decisions or, more specifically, on helping in selecting a concurrency solution that is (better) suited for that specific problem.

As the choice for a concurrency construct can essentially be seen as a design pattern issue, the impact is, as [Hofmeister 99] describes it, sometimes relevant to architecture and sometimes only relevant to detailed design. However small detailed design issues may be, they can sometimes have great impact on the quality requirements for the product (e.g. performance).

The following section describes a method that can help in selecting, or in evaluating an already chosen solution.

1.2 Description of method

Determining a solution for a given problem begins with understanding that problem. If the problem is designing a software product, understanding the problem means looking at the requirements and characteristics for that product.

When building a product, it is important to know what the product should do, how it should do that and under what constraints. The factors that specify what a product should be are called *requirements*, which are collected in a document called a *requirements specification document* (RSD). The particulars on how to construct such a document are described extensively in Requirements Engineering literature (e.g. [Sommerville 01], [Kotonya 97]).

In short, the method described here comprises three steps. The first being the collecting of information (requirements and concurrency constructs), the second being the analysis of information and the third being the assessment of the information (see Figure 1).

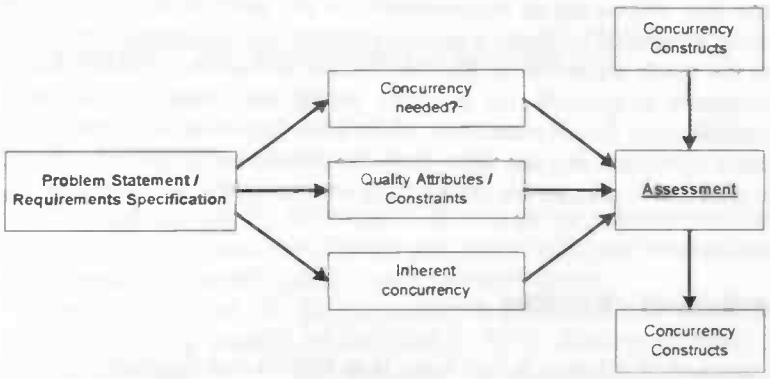


Figure 1: A requirements analysis method

In more detail, the method works as follows:

I. Collecting of Information – the first step in the method described here, is to collect information. Sources for information are e.g. literature, the engineers, stakeholders, domain analysis etc. The two sorts of information used in this method are:

Requirements – the basis for every product. Requirements specify what the product should do, how it should do that and under what constraints. As a source for requirements, the engineer can take an existing requirements specification document (RSD), or, when no such document is present, derive the information needed through a Requirements Engineering (sub-) process. Requirements can be derived by consulting stakeholders, from literature, or by analyzing the problem domain.

Concurrency Constructs – in the case of designing a product which incorporates some form of concurrency, there will be a number of candidate methods that provide concurrency. This method uses concurrency constructs together with their characteristics, which follow either from literature, from the experience of the engineer(s) or other sources.

II. Analysis of Requirements – the second step in this method is the analysis of requirements and constraints. This step is used to narrow down the amount information so that in the following step, choices can be made more easily. In this step, three 'questions' are defined which possibly give clues for taking certain constructs and thus narrowing down the number of suitable constructs:

Concurrency needed – a logical question when designing a system is to ask whether a truly concurrent solution is needed. An engineer should carefully consider the alternatives, especially when the product to be designed is subject to strict constraints.

Quality Attributes / Constraints – in addition to functional requirements for a product, there are a number of non-functional requirements, or Quality Attributes (QAs). These, together with constraints, describe e.g. what the performance of a product should be or how efficient it should be (e.g. concerning usage of resources). Different concurrency constructs have different characteristics for quality attributes and constraints, and should therefore be taken into consideration.

Inherent Concurrency – some problems are inherently concurrent in nature. Inspecting the problem domain might give clues for using a certain concurrency construct or for not using others. Chapter 8 gives an example of a problem where domain analysis gives hints for taking a certain construct.

III. Assessment of Information – when the information needed is present, the final step is to assess that information. Assessment is done by listing the constructs and their properties

together with a list of requirements. In the assessment, all requirements are classified into being supported (or being a general property of), unsupported by a construct or being unknown or too much dependent on the situation for that construct. Results of the classification are then displayed graphically (in a table) listing only those requirements that have a different classification for all constructs. If the table becomes too large, a sub-set can be taken which takes constructs that are alike. Both the textual and graphical notation can then be used in the evaluation or selection of concurrency constructs for the given problem.

1.3 Structure of chapters

The structure of chapters in this thesis is as follows (see Figure 2):

Chapter 1 (this chapter) introduces the material and describes the problem and the requirements analysis method that will be addressed in this thesis.

Chapter 2 uses a sub-process of requirements engineering (domain analysis) to deliver a list of characteristics for a synthesized mobile device.

Chapter 3, 4 and 5 describe the concurrency solutions processes, threads, cyclic scheduling, active objects and concurrent OO.

Chapter 6 incorporates chapters 2 to 5 to assess the different constructs with respect to the characteristics as defined in chapter 2.

Chapter 7 uses the results from chapter 6 to assess concurrency constructs in a 'real-world example'.

Chapter 8 describes a method to implement a formal protocol specification in a way that takes into account the nature of the problem.

Chapter 9 contains the conclusions and recommendations.

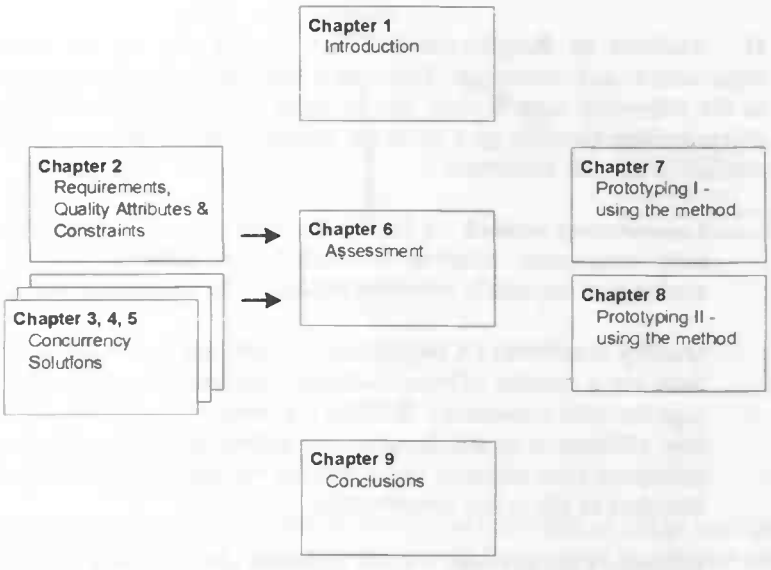


Figure 2: Structure of chapters

2 Case Study – Designing PDAs/Smart Phones

2.1 Introduction

When designing a software product, the first thing to do is to get the requirements for that product. Generally, the sources for requirements are diverse and involve different stakeholders. The process of getting the requirements is generally called requirements elicitation.
This chapter focuses on a subset of the requirements elicitation process, namely deriving requirements by analyzing the context of the product to be developed, rather than eliciting requirements from stakeholders.

The aim of this chapter is to produce a list of characteristics based on a fictional, but realistic future mobile device through a requirements engineering process, together with known concurrency-related characteristics from literature. The results of this chapter are used in chapter 6, where the different characteristics are compared to the different methods for providing concurrency.

2.2 Problem description

Where there were separate devices for calling, office applications and games, each with their own specialized domain, there are now more and more devices that tend to integrate different functional domains into one. The term for these generally communication oriented integrated devices I will adhere to is 'personal digital assistant' (PDAs, this includes Smart Phones, Communicators, Mobile Phones, and 'genuine' Personal Digital Assistants).
The operating system that a PDA runs has to cope with tough conditions. For example, a PDA running out of memory or storage space is not unthinkable. Similarly, when equipped with communication facilities, no assumption on availability of the communication channel should be made. When an operating system is designed to run on a wide range of devices, it will need to be able to cope with the different capabilities of the different devices.
These, and others, are factors that directly influence the design of an operating system for PDAs. Apart from technical issues that an operating system has to cope with, there are number of customer satisfaction related issues, such as availability and usability of a device, but also efficient network usage, to keep usage cost within limits. A device that fails to live up to all these requirements and user expectations will be very likely to fail on today's and tomorrow's market.

Designing an architecture that can cope with the posed requirements and restrictions is non-trivial. There are currently a number of systems in existence that meet the requirements, some being able to do it better than others: often, the approach would be to take solutions from the world of traditional mainframe and PC programming. Although sometimes successful, the solutions taken for these platforms do not necessarily meet the requirements for the PDA context.

The first step when making an operating system for a PDA is to identify the requirements that that system has to meet. Operating systems and their properties have been described extensively by the research community, but mostly these descriptions are concerned with the more traditional mainframe and PC systems. Nonetheless, there are a number of more or less general properties that follow from the literature, which can be used when designing an operating system for a PDA.

In order to define the problem area, in this case concurrent solutions and mobile devices, it is important to derive the properties that are specific for portable devices. To do so, it is wise to first have a look at what the context and intended use of PDAs is [Rakotonirainy 00]. After that, a selection of properties concerning concurrency will be made, which will be used in the assessment. The assessment will be done by comparing a number of concurrent solutions, taken from both the 'traditional' and mobile operating system world, with respect to the selected properties.

2.3 Deriving Requirements: Context Analysis

An operating system for a PDA operates in a varying context (both regarding the device itself and the context the device operates in, see Figure 1). In addition to this, PDAs mostly have limited resources. Nonetheless, high demands are posed on reliability, availability and, to certain extent, performance. As PDAs become more and more widespread, the target audience diversifies: the PDA in the 'old days' was just a gadget for managers, providing basic agenda and address book functionality, dropping prices and improving technology bring these devices within reach of a much broader audience (as happened with mobile phones in most countries in Western Europe).
A wider audience requires different capabilities from a PDA: just office applications may not satisfy younger customers, an audience that probably wants games, audiovisual facilities as well as messaging capabilities. New uses lead to new streams of data to and from the device, e.g. audio and video, e-mail etc. Such new use logically leads to increased demands on memory and CPU, part of which could be solved by remote computation. However, this approach may require a high(er) bandwidth connection, which could lead to increased cost for the owner.

A key point for a system to have success and gain widespread acceptance, is the ease of use: most users will probably not want to be bothered by the internals of the system, they just want to use it. Therefore, most actions, such as, for example, installation on demand should be transparent to the user. In short: as technology becomes more affordable for a wider public, changing requirements for that technology will be unavoidable.
Standards are available for inter-device communication like Bluetooth, and new standards will arise, thus ever increasing the potential number of devices that can be connected to a PDA.

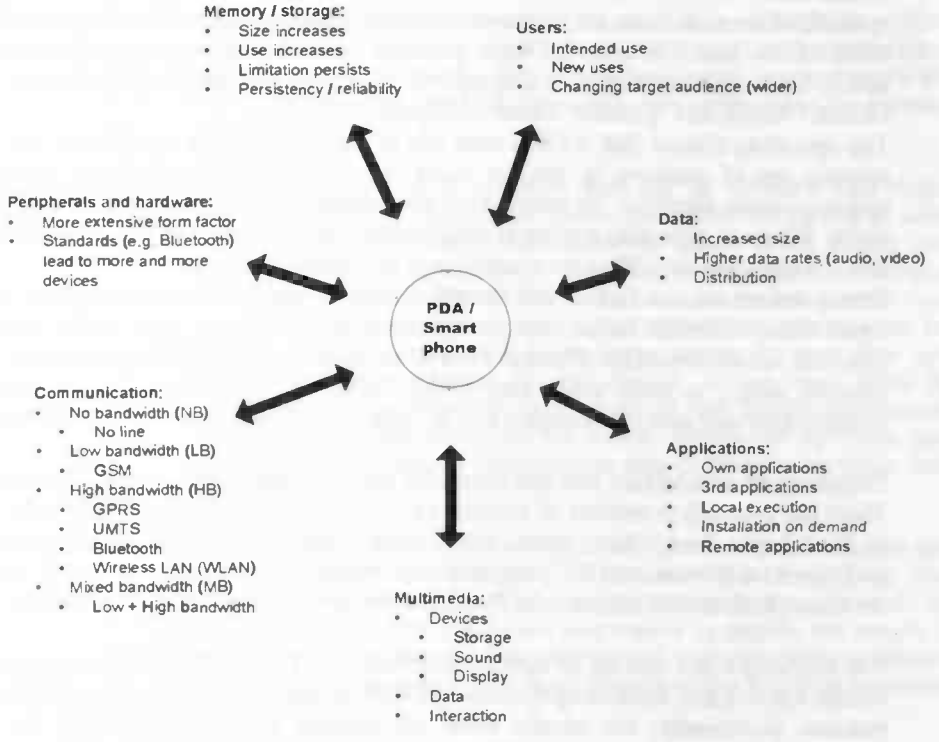


Figure 3: Context description of a PDA

In the following sections, I will try to capture factors that influence, or are at least concerned with, the design and use of PDAs. From these, a selection will be made of factors that are related to or influence concurrency issues.

2.3.1 Data

New uses of PDAs lead to new kinds of data that the device will have to be able to handle. For example, the upcoming higher bandwidth networks like GPRS and UMTS could bring streaming video functionality within reach. Clearly, the size of the data the device will have to handle will increase. Also, the rate at which the data has to be available will increase with such uses.

Factor	Description
Size	With the upcoming of higher bandwidth communication channels, larger amounts of data can be transferred to and from devices. This poses special requirements on a device concerning the limited storage space, reliability (of data and the system) and processing power.
Speed / transfer rate	Higher bandwidth means a larger amount of data in a smaller amount of time.
Distribution	Given the limited storage capacity of a PDA, a selection has to be made of what to store. Furthermore, a user might want to access his or her data from whatever device he or she can. This includes not only PDAs, but also (public) computers, etc. When a device has the possibility to store its data in a remote location, some factors must be taken into account, for example, the possibility that the data is not reachable (the device has no communication channel at that moment). Users will expect their device to function normally even if there is no communication possible, so accordingly, measures will have to be taken. Furthermore, users want their data to be secure, reliable and consistent.

2.3.2 Communication

Having a communication channel available offers new possibilities as well as new problems. Communication includes (in this case) access to remote data, remote processing etc. A device should always try to provide the best level of service it can, which is in this case, the highest and most reliable bandwidth channel.

Factor	Description
No bandwidth	A scenario in which there is no communication possible, leads to a clear need for independence. That is, all critical and/or high priority data must always be available on the device. What the scope of this data exactly is, is to be determined. Examples: user credentials, schedules, addresses, critical applications, and important messages.
Low bandwidth	In the case, there is a connection available, but with limited bandwidth, the device should adapt its behavior accordingly. That is, if the user wishes to have some data updated regularly, there should be some limitations to what data (size, priority) and with what interval. At any time, the tasks with higher priority (incoming calls) should prevail over lower priority tasks.
High bandwidth	High bandwidth channels provide new possibilities for devices. For example, transferring a larger amount of data (e.g. sound/stills/video) at a higher rate. When using a remote computing approach, screen updates could occur more often and at a higher quality.
Mixed bandwidth	Portable devices generally support a number of standards for communicating. Mobile telephones currently support mostly only the GSM standard and/or infra red communication, whereas PDAs support for example Bluetooth, WLAN (IEEE 802.11), HSCSD, fixed (serial) lines etc. As the functionality of multiple devices gets integrated into one, so do methods for communication. This poses different possibilities for such devices, for example, a PDA with GSM functionality and an Ethernet connection could use its GSM functionality solely for calls and the Ethernet connection for its (other) data communication purposes.

2.3.3 Applications

Applications form, together with the operating system, the essence of the device: without operating system no applications, and without applications no use for the device. Although most PC users have gotten used to rebooting their machine occasionally, such behavior will not be tolerated by users of PDAs or mobile phones: they will lose trust in their device and probably resort to other devices. Stability of a device is influenced by both the operating system and (third

party) applications. Although a good operating system can prevent a user process from crashing the system, a badly written system application (driver etc.) could compromise system stability.

Factor	Description
Own	Often, a PDA comes with some essential software like office applications and messing facilities. Although not guaranteed, the software provided by the manufacturer is usually well tested and stable. These applications are mostly optimized for use with the particular operating system that is supplied with the device. In the case of PDAs, this is important, as resources are limited. Manufacturer-supplied applications for PDAs nowadays mostly include office tools, an address book, a calendar, a web browsers (html, wap etc.), games, messaging applications and, sometimes, multimedia applications.
Third-party	If the operating system of the device is open to third-party developers (that is, developers can write both system and application software), which is likely to be more and more in future devices, this means that, even though the manufacturer has done its utmost best to provide a good, stable system, a system application provided by another supplier might compromise system stability. There is not only a threat in system applications, but also in third-party supplied user applications: when a badly written application consumes most system resources, device stability could be compromised. Apart from possible system stability issues, there is the issue of the ease with which third-party software can be installed, and how well the system behaves under software conflicts.
Local execution	In the normal case, all software that is needed, is available on a PDA and is executed there. The applications are all subject to the limitations that go for the specified device. If a device runs out of space, stability should not be endangered. Most devices come with applications for messaging, gaming, office tasks etc.
Installation on demand	In future devices, it is thinkable that not all software that might be required, resides on the device. One argument for not having all software that could be needed pre-installed on the device is the limited storage space of a device. An approach that could be adopted is an installation-on-demand or just-in-time (JIT) installation: whenever a specific function or application is needed, (part of) the software is downloaded to the device. Special precautions have to be taken so that high priority applications always reside on the device, in the case there is no communication possible. The exact scope of high priority applications will have to be defined either by the users' wishes, or by some system requirement. This way of installation is also suitable for dynamic updates of software.
Remote execution	In, for example, UNIX-environments, it is common practice to be able to work on remote machines and only transfer the results (e.g. display output) to the device. Most of the processing is then done on some remote system, while the device only needs to display the results. This allows for a flexible solution concerning the size and complexity of applications, but poses higher requirements on the available bandwidth.

2.3.4 Memory / storage

In the years to come, prices of memory modules will probably drop. This allows PDA manufacturers to supply their devices with more memory. However, when compared to desktop systems, the amount of memory available to the software in a PDA is much less.

Factor	Description
Size	In the years to come, the amount of memory that is installed in a PDA will increase. However, the amount of memory will be far from the amounts that are installed in normal PCs.
Use	There is a constant race between available memory and the use thereof.

Factor	Description
	If the pool of memory available increases, so does its use. However, if the amount of memory does not increase, this does not mean that the use of memory will be tempered. In addition, the use of page-files on PDAs is mostly unthinkable. Therefore, it is important that applications for limited-memory systems are designed with such restrictions in mind.
Persistency / Reliability	Both users and applications will depend on their device not to lose data whenever possible. This means that data must be stored in a persistent way, even in case of power failure.

2.3.5 Peripherals

Peripherals are devices that extend or change the capabilities of a device, of which there are a vast number. The connection between two devices can be by means of a cable/direct connection or wireless transmission. Throughout the years, different (open) standards and proprietary formats for inter-device communication have been developed. Currently, communication standards that are used widely for inter-device communication are USB, Ethernet, IrDA and Bluetooth.
There are factors that must be taken into account when a PDA is to support different peripheral devices, as it is not known what device is to be connected to a standardized connection. Resource usage issues might arise for certain peripheral devices. In addition, some devices require a device driver installed on the host system, which is a potential problem for system stability.

Factor	Description
Standards	The existence of standards allows for a large number of devices to be connected to a PDA. A standardized communication channel, however, does not say anything about the device that will be connected to the other end. Problems (device conflicts, resource conflicts etc.) may arise when devices are connected to a PDA.
Types of devices	There are different groups of devices that can be connected to a system: for example storage, input/output, communication, processing etc. While some devices may extend system functionality, others will change or even limit existing functionality. Also, a situation is thinkable where one peripheral device requires one or more others to be present to function properly. Also, some devices may be required for the system to function correctly, whereas others may be optional.

2.3.6 Users

Devices that are newly introduced on the market, mostly find their way through a group of so-called 'early adopters', a group that has shown to consist mainly of business people. However, as devices are on the market for some time, the group that uses them becomes bigger and broader. This has impact on the way the device is used. Also, there might be a difference in intended use between the manufacturer and the end user (in fact, as the device is on the market for a while, this can change).

Factor	Description
Intended use	There are two kinds of intended use: the use intended by the manufacturer and the use as intended by the end user. End users tend to be creative and find new uses that the manufacturer did not intend. Such use will be categorized as 'new use'. Manufacturers can make some assumptions on devices and their software, provided they are used as they intended. This, however, does not mean that they do not have to consider the new uses.
New uses	When users use their devices in a way that was not meant or not foreseen by the manufacturer, system stability could be compromised, unless the manufacturer has considered this 'new use'. This use of the device could be intentionally or by accident, but this should not matter.
Changing audience	As technology is accepted and used by a larger and broader group of people, the functionality that is requested by users might shift.

2.3.7 Multimedia

Multimedia is the term for a combination of (interactive) audio/video applications. A typical multimedia system consists of hardware and software components as well as data upon which they act. Hardware devices include sound and/or video input/output as well as some kind of storage medium. The data used is often compressed. Therefore, an encoder/decoder is needed for playback or recording. Processing the data can be done in hardware or in software.

Factor	Description
Devices	There are largely five types of devices involved in a typical multimedia system: 1. The CPU, 2. Audio-devices, 3. Video-devices, 4. User-interaction devices, and 5. Storage media
Data	Multimedia systems generally work with large data (audio/video). Often, this data is compressed to reduce the total data size. The compression and decompression of data are CPU-intensive tasks that might not work on systems with limited CPU power.
Interaction	Another point in multimedia is the ability to 'interact' with the data. This could be, for example, a computer game. To this purpose, there need to be some user-interaction devices such as a mouse, keyboard, joystick or pen device.

2.4 System characteristics for mobile systems

Having made an inventory of context factors for mobile systems, this list can be used to look at what impact those factors have on the design of (operating) systems for such devices. Doing so leads to a number of characteristics or *requirements*. Some of these requirements are clearly of functional nature, others are quality-related. In the following sections, a number of requirements are 'derived'. Identified requirements are written in italics.

2.4.1 Data

All devices work with data, but the type of data handled varies per device and per application. Increased data sizes could lead to storage shortage. In order for the system to be able to continue functioning properly, some kind of 'warning mechanism' is desirable, for which notifications must be able delivered as fast as possible. Such a notification is generally called an *event*. For example, when the amount of free storage space drops below a certain threshold, an event is fired and applications could act accordingly.

When data is used in a distributed fashion, the system must take into account that data might not be available instantly or not available at all. Applications depending on this data should be able to be notified of (temporarily) non-available data.

When high-priority data is required, the system must make sure that this data is available. Lower-priority data can be fetched later on. One solution for handling the possible delays in systems with distributed data is the *pre-fetching* of required data. Pre-fetching is generally done as a background task.

With specific types of data that require preprocessing (e.g. transferal, unpacking) before they can be used, a *background process* (i.e. a process with a low *priority*) could be used to do the work in advance, transparently to applications.

2.4.2 Communication

Having a communication channel offers new possibilities. Concerning concurrency, there are a number of observations, which have certain consequences: of these the observation that the availability and amount of bandwidth can and will vary is probably the most important. Together with the notion that the device should always provide its users with the best of its capabilities whenever possible, this leads to the observation that the system and its applications should be

notified of bandwidth changes, so that the device can adapt its behavior accordingly. The same goes when a device supports remote computing and the communication channel is closed.
In addition to this. the device should be able to distinguish between different types of communication tasks. For example, calls should be handled with a higher *priority* than the background downloading of some new software. Therefore, there must be some mechanism that allows tasks to be interrupted for higher priority tasks. This observation leads to the need for *preemption* of tasks.
Telephony facilities need support for *real-time* (RT) tasks that always have higher priority than other tasks the device is handling in order to guarantee continuity of service.

2.4.3 Applications

Applications on a PDA need to be designed with the limitations the device poses in mind: this means, whenever possible, small memory usage and efficient coding. However, being primarily a user-interaction device, applications need to be responsive.
Sometimes, applications can be written more efficiently when parts of an application can run in a more or less independent fashion of the main program. In traditional systems, this is generally done by moving the part to a separate unit of execution (e.g. a *thread* or a *process*).
A *remote execution* approach should preferably be *transparent* to the user. This could be done by, for example, a client-server approach, where the server handles (remote) execution (see section 11). When a device has Installation-On-Demand (IOD)/Just-In-Time (JIT)-functionality, this should be *transparent* and at runtime.
Applications that crash or have other errors, should not compromise system stability (e.g. by leaving memory allocated). Therefore, *system-wide error handling* is essential. A clean way of installing device drivers is by adhering to a microkernel/system server approach. *Clients* request services of the *servers* by some privileged call. Calls can be both synchronous and asynchronous. (e.g. [Burns 01], sporadic servers or [Tasker 00], EPOC system servers). Extending the system can be done at runtime.

2.4.4 Memory / storage

The amount of memory in a PDA is limited, and extending the amount is not done as easily as it is done in PC systems. Instead, system software and applications should be written with the limited amount memory in mind. In terms of concurrency, the *overhead* involved in introducing and using concurrency in an application should be as *small* as possible. This includes CPU-overhead, as well as memory overhead.
Another way to save on memory is reuse of resources. Resources can be run-time bound, or storage bound, such as the size an application takes in memory. Concerning the former, the system needs to keep track of what resources are in use by what application so that in case of a crash or an unclean shutdown, these resources can be reclaimed. E.g., disk space usage, as meant by the latter resource can be reduced by e.g. the use of shared function libraries.
When the system threatens to run out of memory, the operating system must be able to close down applications, in a –whenever possible– 'safe' way, so that memory can be freed. In addition, applications can *adapt* their behavior to the amount of memory available, for example, a web browser could switch off downloading of pictures or 'active content' to save memory. On the other hand, when more memory becomes available, the device should act accordingly and provide the best service it can.

2.4.5 Peripherals

Most devices have a number of general interfaces to which peripherals can be connected. Some devices may have support for (a number of) peripherals built-in in the operating system, others might need the installation of a device driver. Users might not want to be bothered by the installation of a device driver, they just want Plug-n-Play (PnP) installation of their peripherals, and whenever possible, without rebooting their device. This functionality is called *hot (un)plugging*. Using this approach requires the system to be *flexible* and support run-time extension. In addition, when peripherals are connected (provided the required drivers are installed), they extend or change system functionality: applications should be notified of this change in context. Device drivers handle signals coming from devices or the system. Applications should be able to make use of the devices connected, but should not bothered with the details of its internals. This notion leads to the requirement of standard interfaces.

2.4.6 Users

PDAs and mobile phones are mass-market devices. Users should always be able to rely on their device, but on the other hand not be bothered by its internals. Users or operators should be able to change settings or install new functionality over the air (OTA) transparently. Furthermore, the system should be both *adaptable* and *adaptive*, so that the different needs of different users can be satisfied.

2.4.7 Quality attributes

Apart from several functionality-related factors, each system needs to satisfy a number of quality attributes. Failing to do so, generally leads to customer dissatisfaction that can lead to project failure. With PDAs, this is not different. The quality requirements for these devices, among others, are *reliability* and *availability* (the user must be able to rely on the device), *flexibility* (installation of new applications), *efficiency* (memory, CPU and power consumption), and *transparency* ('don't bother the user with technical details').
Assuring that these quality requirements are met, can be done by, among others, implementing them as functionality. A number of important quality attributes, and how they could be converted into functionality is given below:

Reliability – reliability of a PDA is important: no data should be lost or corrupted. Therefore, measures must be taken that prevent unreliability of a device. Device reliability can be compromised when certain resources (power, memory, storage etc.) run out or become unavailable. Mechanisms that cope with such situations could include tight and/or centralized *resource control*, enforcing or encouraging *efficient resource utilization* in applications, (centralized) *error handling* and *recovery mechanisms*, and, possibly, redundant hardware (e.g. a backup battery).

Flexibility – PDAs are mass-market devices and thus cover a wide audience. Different groups in the audience have different requirements on and expectations of the device. It is not economical for a manufacturer to develop several different systems for several groups. Therefore, the (system) of the device needs to be flexible and allow for customization. Another reason why a system should be flexible, is the differing conditions the devices operates in, and therefore, a device might need to dynamically adapt its behavior to reach the best overall performance possible. A flexible system can be achieved by partitioning the code into functional units (modules, objects), which can, to certain extent, be used to customize the system. In addition, a partitioned system could allow for a client/server approach, in which part of the functionality of the system or application could even be made external to that system or application.

Verifiability – in systems where resources are limited, and debugging is hard or impossible, and recalling systems for maintenance is no real option, like with PDAs, verifying that the system behaves as requested is of great importance. Verifiability of a system can be achieved by partitioning the system into more or less independent modules or objects that can be tested and verified separately. Also, avoiding constructs that make verification hard, such as concurrency, can help in verifying.

Efficiency – one obvious point in resource-limited systems is efficient use of the available resources (whether they be CPU time and, –related to this– power consumption, memory or storage space etcetera). However, when a system is open to third party applications, those applications cannot always be trusted to be as resource-efficient as desired. Therefore, in such limited systems, it is wise to control the awarding of resources wherever possible, reusing resources wherever possible. Examples of resource reuse are dynamically linked and loaded libraries that are only loaded when needed and are shared by possibly multiple applications. When no longer needed, they are unloaded to free up memory. A similar scenario can be adhered with such 'read-only' resources. In addition, a device that mostly relies on batteries for its power supply should be designed with this in mind. Therefore, whenever possible, the system should take power-conserving measures. This includes a policy for unloading or disabling parts of the system that are not in use.
An example of an architectural pattern that allows for power conservation is an *event-based* system that only is active upon and after the reception of an event.

Transparency – mobile mass-market devices target a much wider audience than only computer users. The majority of users will therefore not want to be bothered by the system's behavior. When a mobile operator decides to change settings, the users should not be required to reboot his or her device. In addition, when a device allows for remote computation, this should proceed transparently, and applications should continue working whether or not there is a communication channel available. Therefore, it is important for such a mobile device to adapt its behavior transparently depending on the context the device is in at any time. To do so, the device needs to be aware of its current state and context. Also, it must be able to detect and handle any errors that result from changing state and/or context. Dynamically loading and using modules and/or a client-server approach might be a solution for the transparent remote computing.

Availability – one more point that counts more heavily in the PDA-market than it seems to count in the PC-market, is the point of availability of a system: users that have to take their devices through an extensive boot-up session before being able to use it, are likely to get irritated by the device and resort to other devices or methods. In short: a device should be as readily available as a normal agenda. This requires a device to retain its state when turned off, but even better: the device should never have to be turned off or rebooted at all. This also has implications for applications: they need to be written with great care to prevent memory leaks. Naturally, there will always be parts of the system that can fail or need to be updated. A modular system can be of assistance in this case: only the failed or parts to be updated can be replaced and (re-)started. Preferably, system intrusiveness should be as low as possible, which should also be enforced by the operating system.

2.4.8 Power management

Mobile devices must handle the amount of power that is available efficiently, and use, wherever possible, power conserving mechanisms. This could include unloading or disabling parts of the system that are not in use. Although this is a 'defensive' way of power conservation, there is also an 'offensive' way of power conservation: designing applications with power efficiency in mind. Efficiency can be achieved by optimizing software at code level, as well as at architectural level.
PDAs are typical user-interaction driven devices that mainly wait for user input and perform some action upon reception of such an event. This observation leads to the idea of event-based systems of which the idle time can be used to conserve power. Suggested reading in [Golding 95], [Tasker 00].

2.4.9 Context-awareness

In order to provide the best possible performance to its user, a mobile device needs to have information on its context [Chen 00], both internal, and external. One example of this is for example the selection of the communication channel with the highest bandwidth: without knowing what the system's environment is (e.g. what method of communication is available), the system will not be able to provide efficient and transparent switching to the best available communication channel.
Continuous polling of the different interfaces is possible but is inflexible and generally inefficient. Therefore, there must be system-wide facilities that can inform the system of its context, so that the system and its applications can adapt their behavior accordingly when the system's context changes, to provide the best level of service it can. A system that exhibits such behavior is said to work in a 'Best-Effort'-way.
There can be conflicts in Best-Effort requirements like, for example, using the highest available bandwidth and cost of usage. In such cases, letting the user of the device choose might bring the solution.

2.5 Characteristics and Requirements Analysis

From the analysis follow a number of characteristics for mobile systems that will be assessed, which are listed in the table below. The table shows the requirements that are specific for the mobile system in the upper half; the lower half comprises characteristics for 'traditional' systems. The latter of these are taken from literature, e.g. [Stallings 00] or [Tanenbaum 92]. Some of the

characteristics mentioned are valid for both types of systems, and are therefore mentioned only once. The table can be read as '*row* is affected by/requires *column*'.

Characteristics		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	Runtime binary code reuse			+			+	–	+	–	+	–	–		+	+	+			+	–	+		+		R	
2	Effective event handling			+			+											+			–			–	–	+	
3	Fault tolerance		+		+	+		+		+				+	+	+	+			+	+	–	+	+		+	
4	Flexibility / adaptivity	–	+											+						x	+	–		+			
5	Highly available	+		R	+					+	+			+		+	+		–	+	+	+	+	+		+	R
6	Memory efficiency	+	+	–						–		+				+			–		–	–	–				–
7	Openness (extensibility)	–	+	+	+		+		+			+	+		+	+	+	+	+			–	+	–	+	+	
8	Power efficiency	+	+		+		–			–	–					+			–		–	–		–			–
9	(Preemptive) multitasking											–		+					+	+	–	+	R		R	R	+
10	Real-Timeliness	–			–			–		+			–	+		+	+	–	–	+	–	+	R			R	
11	Resource reuse	+					+		+								+				–	–	–	–		R	
12	Scalability			+	+					+	–			+	+	+	+	+				–			+	+	+
13	Task prioritization									R								+					R		R	+	
14	Transparency		+	+	+			+		+				+		+	+				–	–	+	+	+	+	+
15	Verifiability	+	+	–	–	+		+		–	+			+			+	–		+	–	–		–	–	R	
16	QoS support		+	+	+	+	+	+	+	+	+	+	+	+	+	+		+				+	+	+	+	+	+
17	Fairness of scheduling		–				–			+	–					+			+	+		–	R		R	+	
18	Generality over dedication		–				–	+	–	+			+	+		+		+		+			+		+	+	+
19	Low adaptivity	+			x		–	–	–	+						+		+				+	+				
20	Memory & CPU distribution	–	–	+	+	+		+		+		–	+	+	+	+			–	+		+	+	+	+	+	R
21	Performance over efficiency			+	–	+	–	–	–	+	+	–	–	+	–	+	+	–	–	+	+			+	+	+	
22	(Prot.) unit of execution									+		–		+		+		+	+						+	+	+
23	Redundancy support	+		+	+	+	–	+	–	+		+	+	+	R		+		–	+		+				R	
24	Schedulability of tasks		–		+					+	+			+				+	+				R			+	
25	Synchronization		+	+	+					+		+	–		+								+	+			
26	Virtual memory				+										+				+		+		+		+		

Table 1: Requirements Analysis. Used symbols: *empty* = irrelevant or unknown, x = mutually exclusive, – = conflicting/degrading, + = reinforcing/cooperating, R = required

2.5.1 Requirements analysis

One important issue in Requirements Engineering is the analysis of requirements. Analysis means looking for possibly conflicting requirements, checking for feasibility of requirements, and –when necessary– prioritizing requirements to determine the order in which they will appear when building the product.

Table 1 shows an analysis of the requirements that were elicited in the previous sections. The results as shown in the table have been derived by logic. However, in different cases and different context, the result can be different. Therefore, this table is not put here as *the* truth, but rather as *a truth*.

In the table, some rows/columns contain more symbols than others do. This means in some cases that it is hard to identify the relation, or simply, that there is not a direct relation between two items. It must be mentioned here that in some cases, terms as used in the table might be explained in different ways. In this thesis, the terms are meant in the context they were described in the previous sections.

In a software project, the results from such a table as the one above can be used to make a prioritization of requirements that can be used for incremental development.

2.6 Observations and conclusions

This chapter shows that a sub-set of techniques from requirements engineering can be used to elicit information, in this case a number of characteristics for a mobile system, and more specifically, characteristics with respect to concurrency.

The subset of techniques used here is domain analysis. In this chapter, the requirements have been derived by 'brainstorming' about the possible uses of such a device, as the device concerned here is synthesized, rather than being an actual, existing, device

This chapter does not claim completeness, it would follow too far to do so here and there is a large amount of literature on the subject of requirements engineering. What it does show however, is that a software engineer can derive a number of requirements for a device by

analyzing the product he or she is working on. An elicitation process such as described in this chapter can be used both during the actual requirements elicitation process for the product, or at a later stage, when the engineer requires more information.

The results from this chapter are used in chapter 6, where different concurrency constructs are compared to the requirements posed here.

3 Introducing concurrency

3.1 What is concurrency?

Most modern computers systems have some kind of mechanism that allows them to logically execute multiple programs at the same time. While the concept is simple, implementing it is not that easy: when a program executes on a system, it uses resources of that system (resources being other entities in the system the program needs, such as memory, execution time and/or peripherals). However, when multiple programs are competing for the same resource(s), conflicts arise. To make handle the issues that arise, and to make sure programs can perform their tasks, different solutions have been developed and used. The term generally used for this is known as *concurrency*. [Hesselink 01] describes this as 'the art or craft of programming cooperating sequential processes'.

3.2 Motivating the use of concurrency

There are different reasons to use or not to use concurrency in a system. The decision on whether or not using a concurrent solution depends on a number of factors, a few of which I will address.

Often, a system can be implemented more efficiently [Stallings 00], [Tanenbaum 92] when a concurrent solution is used. This is the case, when there are tasks that cannot keep the processor(s) fully occupied throughout their execution (this is especially the case when the process performs I/O-operations. Processes that do so are called I/O-bounded). In this case, idle or wait time of tasks may be used to perform other tasks. Doing so, can result in a system that is more responsive to its users. In addition, a concurrent solution sometimes better reflects the actual system or problem domain: some systems are inherently non-deterministic and concurrent, for example, embedded systems [Burns 93], so a system supporting concurrency is the only way to go.

There are, however, situations where a concurrent solution is overkill: there is often a nontrivial overhead involved in supporting true concurrency, for which there may be not enough space[1] in the system, or there simply is no need for a full-blown concurrent system, and other solutions may perform as well, or even better. Therefore, the use of concurrency in a system needs to be considered well, and if it is used, one should think of what kind of concurrency control is used.

3.3 Issues in programming

When a system supports concurrent constructs, this has its reflections on the way programs are written: not all programs cope well when they are executed concurrently.

3.3.1 Sequential programming

Traditionally, programs are designed to execute their statements[2] in one particular order. Through the use of branching, the order in which code is executed can be changed. However, for each set of inputs, there is only *one* possible path of execution. There is an advantage to this, which is that assumptions can be made on execution of the program. In other words, for a given set of inputs, the program is *deterministic*. The result of the example below would be { x = 4, y = 2 }.

```
x := 1;          (* Statement P: assign value of y to x *)
y := x + 1;      (* Statement Q: assign value of x + 1 to y *)
x := y + 2;      (* Statement R: assign value of y + 2 to x *)
```

Example 1: Sequential programming

[1] E.g. embedded systems
[2] Executable commands

3.3.2 Concurrent programming

When programming with concurrent constructs, there are a number of issues that might not be obvious at first. For example, concurrent programs might be no longer deterministic. In Example 1, the result is known. However, when the three statements (P, Q, and R) are executed concurrently, the result can be any of a set of possible solutions, because it is impossible to predict the execution order of the three statements P, Q, and R. Programs that behave non-deterministically are generally unwanted, and precautions should be taken to ensure that where needed, execution order is deterministic, even in a concurrent environment.
Preventing non-determinism is done by posing constraints on the execution of statements. This is known as *synchronization* of statements (synchronization is also used on higher level of abstraction, e.g. process synchronization).

3.4 Unit of execution

Most modern operating systems support different unit's execution. In general, there are two kinds: processes that function as the basic unit of execution and protection and threads that are generally sub-units of execution within one process.

3.4.1 Processes

In traditional uniprocessing, single-tasked systems, the only process that would be running would be the actual, running program. In concurrent systems that support multiple processes at once, this concept does not change: in most systems, the operating system even supplies every task with its own address space and to the process, it would look like it is the only program running[3].
A process generally has a number of properties attached to it such as a memory context and file descriptors, which are mostly provided and controlled by the operating system. To allow for secure and reliable computing in such an environment it is of vital importance that each process' properties are protected from other processes, so that the integrity of the environment of the programs is guaranteed.

In short, the most noticeable characteristics of processes are:

- Protected unit of execution
- Scheduled by the operating system
- Own context (virtual address space, file descriptors etc.)
- Can own one or more threads

3.4.2 Threads

Threads are, as processes, units of execution. The main difference with a process, is that a thread is generally viewed as a subpart of a process ([Stallings 00] mentions threads as *lightweight processes*). To the operating system, threads mostly appear as a special type of process that is associated with another process.
A normal process generally consists of one thread of execution, the main thread. In addition, a number of systems support *multithreaded* processes. Most systems support dynamic creation and destroying of program threads throughout the execution of the program.
Threads owned by a process have in common that they all share the properties and structures that are associated with the process by the operating system. Because of this, inter-thread communication can be done very efficiently (generally by sharing some memory location to and from which the different threads write and read). Another benefit of threads over processes is that they are generally (a bit) more efficient in creating and destroying, because they do not have the overhead (structures) that processes have associated with them. Only upon creation or destruction of the 'parent' process, the structures need to be created or destroyed.
One of the strengths of threads is also a disadvantage: because different threads share the same structures as the owning process, threads can corrupt one another by writing in each other's memory areas. In short: because of the lack of protection that threads have, it is essential to verify

[3] There are, of course, exceptions.

that, when using threads, memory access code is well written in order to prevent program or system corruption.

Thus, the main reason for using threads instead of processes is efficiency. For example, in a file server, the load caused by the amount of communication that is needed between different tasks, could be relieved by using threads [Stallings 00].

The most important characteristics of threads are:

- Unprotected unit of execution (within one process)
- Scheduled by the operating system
- Owned by a process or other thread
- Memory and context shared with other entities within owning process
- Can own one or more threads (depends on system)
- Has less overhead in creation, destruction and switching than processes

3.4.3 Fibers

One special kind of thread is the thread that is not visible to the operating system as a separate process. [Burns 01] refers to this kind of threads as *fibers*. Handling fibers is done by an in-program scheduler. There is no further distinction between threads and fibers other than the one mentioned.

Fibers are characterized by:

- Unprotected unit of execution (within one process)
- Scheduled by the owning entity (a thread or process)
- Shares memory, context and most properties of owning entity with other threads

For further reading on processes and threads I recommend [Tanenbaum 92], [Stallings 00] and many other excellent sources available.

3.5 Scheduling

The control of resources is usually not managed by the processes themselves, but rather by some entity in the operating system. In the case of the resource 'processor time' (the time that a process can execute), allocation is done by an entity called the *scheduler*. The scheduler's task is to make sure processes get executed, but must also take into account a number of requirements such as, for example, fairness, efficiency and possibly real-timeliness. Other schedulers may need to incorporate demands on process distribution in a multi-processor system.
Often, the scheduler needs to be able to distinguish between different levels of priority for different processes or threads. Different scheduling methods have been proposed and used throughout the years, of which I will address some.

There are, basically, two types of schedulers: preemptive and nonpreemptive ones. A preemptive scheduler is able to interrupt running processes in favor of other processes. Nonpreemptive schedulers let running processes run until they have finished or explicitly indicate that control can be transferred, before handing execution to the next process.
The choice for a preemptive or a nonpreemptive scheduler depends on the requirements and constraints of the target system. Often, a preemptive scheduler will be preferable, but there is an overhead involved in preempting processes that might be nontrivial on certain systems: interrupting a process, saving its state[4], loading another process, and begin executing the other process. Nonpreemptive schedulers are generally easier to implement and require less overhead, at the possible expense of process responsiveness.

[4] A process' state is also known as *context*

Different scheduling methods have been developed throughout operating system history, and it would lead too far to try to describe all of them here, but nonetheless I will try to give a short, non-exhaustive overview of schedulers in use:

3.5.1 Preemptive schedulers

Round-robin scheduling (RR) – the simplest and – at first sight – fairest form of scheduling is round robin scheduling. A round robin scheduler is a preemptive scheduler that works with fixed time quanta or so-called time slices. Each task is placed in a queue. At regular intervals, the scheduler switches to the next task, placing the previous task at the end of the queue. Although this method is one of the fairest methods available, it is not the most efficient: I/O-operations generally spend most of their time waiting, thus needing little processor time. If such an operation is scheduled for execution for a fixed amount of time, this time is essentially wasted.

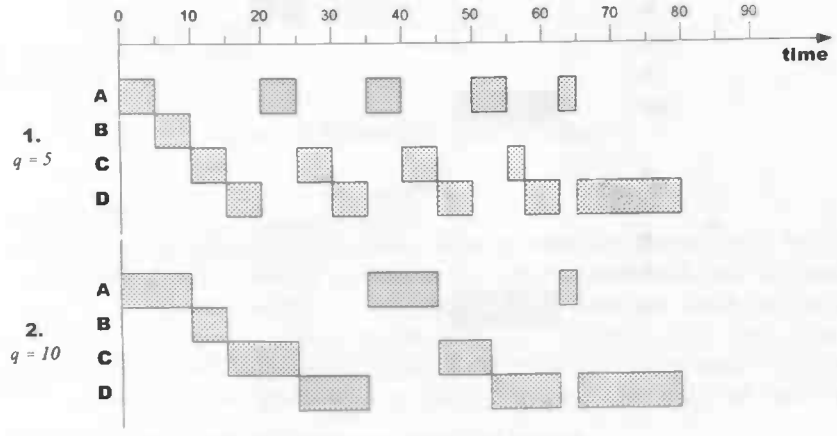
Shortest Remaining Time (SRT) – a SRT-scheduler is a preemptive scheduler that favors tasks that have a short (estimated) execution time *remaining* over tasks that have a longer estimated execution time remaining. This scheduler can suffer from the same drawbacks as the SPN-scheduler concerning long tasks.

3.5.2 Nonpreemptive schedulers

First-Come-First-Serve (FCFS) – this non-preemptive scheduler handles each task in the order in which they come in. Only when the currently running task finishes executing, the next task can be executed. This method of scheduling results in long latency times. This scheduler is also known as the First-In-First-Out (FIFO) scheduler.

Shortest Process Next (SPN) – this non-preemptive scheduler acts by first processing the task that has the shortest (estimated) execution time. Although this method is more responsive for shorter tasks, it could potentially lead to starvation[5] of long tasks when new short tasks are added at a steady rate. As the execution time for most tasks is not known at the instant the tasks starts, this scheduler tends to be less predictive than other schedulers are. [Tanenbaum 92] calls this method 'Shortest Job First' (SJF).

Highest Response Ratio Next (HRRN) – this is a non-preemptive scheduler that favors tasks with a smaller percentage of idle time (idle time is, for example, waiting for I/O) over tasks with a larger percentage of idle time. With this scheduler, there are no performance penalties for long processes as with SRT- and SPN-scheduling. In addition, the possibility of starvation is eliminated because this scheduler takes into account the amount of time a task has waited to be scheduled in the decision of which task to execute next, so eventually all tasks will be executed.



[5] Starvation: an indefinite denial of access to one or more resources

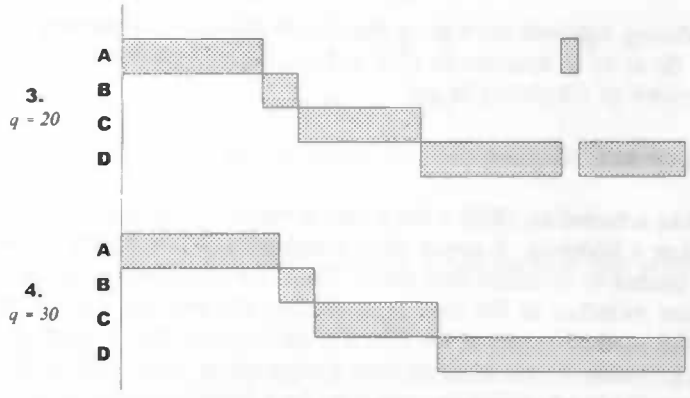


Figure 4: Comparison of example Round-Robin Schedulers with time quanta of $q = 5, 10, 20,$ *and* 30 units

Process	Execution time (units)
A	22,5
B	5,0
C	17,5
D	35,0

Table 2: Execution times for Figure 4

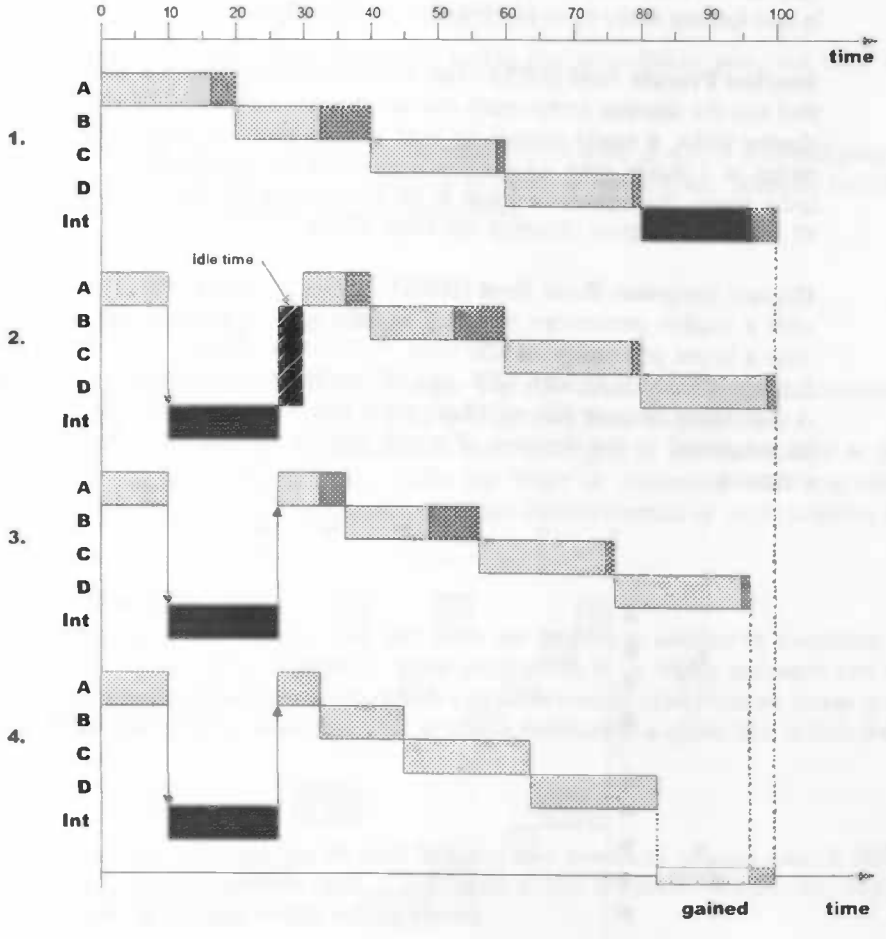


Figure 5: Timing examples for four simple schedulers

Process	Execution time (units)
A	16,25
B	12,50
C	18,75
D	18,75
Int	16,25

Table 3: Execution times for Figure 5

3.6 Cyclic Schedulers

One kind of scheduler that differs from the aforementioned ones is the so-called cyclic scheduler. Cyclic schedulers are nonpreemptive schedulers that mostly do their work at application level. In fact, for some applications (like in embedded systems), a cyclic scheduler may make a full-blown operating system unnecessary.

3.6.1 Standard implementation

Generally, a cyclic scheduler is a loop in an application (see Figure 6) in which different routines ('tasks') are called one after another. This is clearly an inflexible way of scheduling tasks, and makes changes to the application hard.

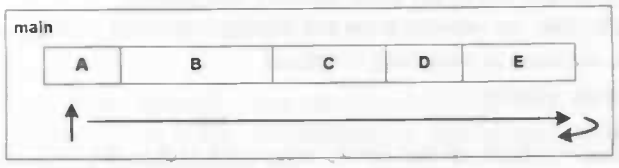


Figure 6: Cyclic scheduler with five tasks

3.6.2 Providing flexibility

In an extended version of the cyclic scheduler (see Figure 7), the actual scheduling and the different tasks could be 'decoupled' by providing a list to which tasks can be added at design time, or even throughout the application's lifetime. This approach makes the design more flexible and better maintainable.

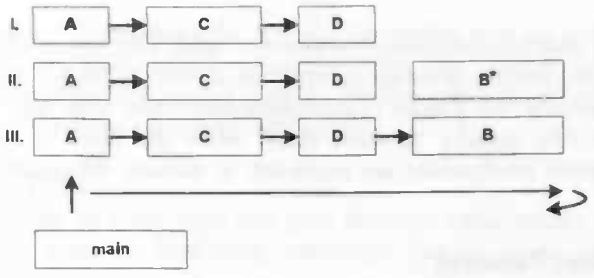


Figure 7: Extensible cyclic scheduler

3.6.3 Adding priorities

Although better than a standard cyclic scheduler, there is room for improvement in the solution above. For example, there is no notion of priority in this version: each 'task' still is executed in the order in which it was originally added to the scheduler. Furthermore, tasks that require to be executed more often than others cannot be implemented efficiently this way. This problem becomes even more obvious when a number of new tasks are added: the occurrence factor of tasks goes down. One way to solve this problem is to attach priorities to the different tasks (see Figure 8).

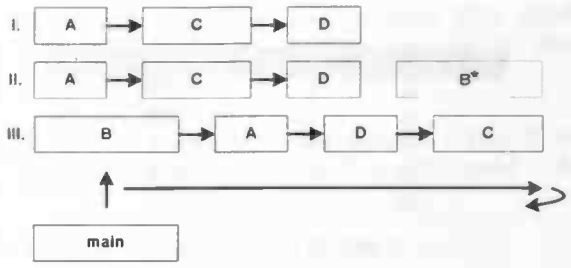


Figure 8: Extensible cyclic scheduler with task priorities

3.6.4 And further

For a number of tasks the lightweight solution as described above will do. For certain applications however, performance is non-optimal. The solution as described in Figure 8 is not power efficient as it executes a tight loop and event handling is hard to do in a clean way with the proposed solution. Symbian has extended the solution from Figure 8 to overcome these problems, which resulted in the Symbian's Active Object/Active Scheduler framework. A description of this framework is given in section 4.4.
In short, the 'pure' cyclic scheduling approach can be characterized by:

- Scheduling by repeating function calls
- Unit of execution: function
- All entities run in the same memory and context
- Inflexible: no prioritization and adding/removing entities
- No memory or switching overhead
- 'Niche-solution'

The extended versions of the cyclic scheduler tackle the inflexibility part, but have the same drawbacks as their naïve cousin.

There are problems that can arise when a cyclic scheduler is used in object-oriented programming and the used class is subclassed. Such a situation is called an inheritance anomaly (see chapter 5 for more on object-oriented languages and concurrency).

3.7 Communication and synchronization

Often, tasks need to communicate with other tasks. The way in which different tasks communicate depends on the way the tasks are constructed, as well as the system they run on.
There are roughly two groups of communication here. One type of communication is concerned with actual data transfer between tasks, while the other is concerned with synchronization. Synchronization mechanisms are important to indicate the occurrence of asynchronous events to other tasks.

3.7.1 Message Passing

Message passing uses system calls to send data from one process to another by executing a system call. In UNIX, each process has a message queue attached to it, in which messages can be stored until the process is ready to process them. When a process tries to read from an empty queue, it is suspended. Analogously, when a process tries to send a message to a queue that is full, the sending process is suspended.

3.7.2 Pipes

A pipe is a system structure that can be used between two processes to send data. It differs from message passing, that processes first open a 'pipe' to and from the involved processes can read and write a continuously (of course within certain limits).

3.7.3 Sharing Memory

As already mentioned in section 3.4.2, threads can communicate by using the memory they share. This is an efficient way to communicate, because there is no overhead involved, which is at the same time its Achilles heel: when not coordinated, different threads may overwrite the same memory location, which leads to program errors.
In processes, whose context is protected by the operating system, there is no possibility to write in each other's memory locations. Therefore, there will have to be a central memory location that processes can write to and read from.

3.7.4 Semaphores

When a process uses a resource exclusively, and another process tries to access that resource, it should be denied access by the operating system. However, if the resource is essential for the latter process to complete execution successfully, the process should be suspended until the resource becomes available. One solution for signaling the latter process that the resource is free, is the use of semaphores. Semaphores are, in essence, counters that are used to indicate whether the resource is free. More on semaphores in [Tanenbaum 92] and [Stallings 00].

3.7.5 Signals

Signals [Stallings 00] are a mechanism from the UNIX world that has an analogy in hardware interrupts. Signals are used to inform of the occurrence of asynchronous events, which makes them in essence non-queued messages.

3.8 Real-Time systems

Apart from 'normal' concurrent systems, there are so-called Real-Time (RT) systems. The main characteristic of Real-Time systems is the incorporation of timing constraints on the execution of tasks [Gomaa 93], [Burns 01], [Spaanenburg 99].
A subset of these timing constraints is referred to as limits or deadlines, which deals with the time a task needs to be in a certain state. In case a task does not meet its deadline, terms like 'timeout' and 'deadline overrun' are used: as [Burns 01] describes it, a timeout is the ability to recognize, and act upon the non-occurrence of some external event. The difference with a deadline overrun is subtle: a deadline overrun is a less-general form of time-out and is concerned with some action not finishing in time. Other timing constraints used in Real-Time systems are the minimum and maximum delay (amount of time to elapse before a process starts executing) and the maximum execution time.

The need for a RT-system follows from the domain the system is to be used in. These are often systems where reliability and safety of the system play a big role. Typical examples of RT-systems are flight control systems, industrial control systems, medical systems, and often systems that have to deal with some external system (generally hardware).

3.8.1 Flavors of Real-Time

There are different 'flavors' of Real-Time that pose different requirements and consequences on timing constraints. The type of Real-Time constraint that is to be used depends on its requirements:

Soft Real-Time – missing a so-called 'soft' deadline degrades performance, but is not fatal to the system. If the task delivers its result at a later time, the results can still be used – the system will still draw some value from the task. As Spaanenburg puts it: 'You might miss a train, have to hurry to the shop or have to walk the last ten minutes, which may be strenuous, but is nothing really severe.'

Hard Real-Time – missing a hard deadline, however, causes the system to fail. Special care must be taken not to miss a hard deadline. Often, hard deadlines are used in systems that perform control tasks: for example, a robot arm that is being moved, must be stopped in time, otherwise it could cause harm to its environment.

Firm Real-Time – a third kind of deadline is the so-called 'firm' deadline. A task that misses its (firm) deadline does not cause the system to fail, but the result from the action is useless/valueless to the system [Burns 01].

It is possible for systems to have a mixture of the different types of deadline scheduling or a combination of some Real-Time tasks together with non-RT-tasks (which is actually a quite general case).

3.9 Discussion and summary

History has proven the use of concurrency to be advantageous – CPU power can be used more efficiently and systems became more responsive. Real-Time systems are currently almost everywhere and are today an essential part of everyday life.
When an engineer designs a system, he or she should carefully consider whether to use concurrency and in what form – is, for example, a hard Real-Time scheduler really needed or will a less strict variety do? Furthermore, the simplicity of the problem or resource constraints might not justify the use of a full-blown concurrency solution, and a cyclic scheduling approach may prove to perform just as well.

It is important, when designing a system, to investigate what role design patterns (that is what a concurrency construct is) play. Sometimes a design pattern is of importance for the system's architecture, and sometimes it is only a minor implementational facet. One benefit of the methods described in this chapter is that there is a lot of literature an engineer can resort to for information when designing a system.

This chapter does not seek to give complete insight in the quirks of current concurrency methods, but it should give an introduction that is sufficient for understanding the rest of this thesis.

4 Symbian: Active Objects

4.1 Symbian OS

Symbian is a software licensing company, owned by wireless industry leaders, that is the trusted supplier of the advanced, open, standard operating system - Symbian OS - for data-enabled mobile phones. Initially, Symbian was formed by Nokia, Ericsson and Psion in 1998. Shortly after, Motorola joined the alliance. In 1999, Matsushita of Japan (better known as Panasonic) joined. At writing time, Ericsson's share is owned by Sony Ericsson Mobile Communications (est. 2001), a company equally owned by Ericsson and Sony. Recently, Siemens IC Mobile joined as a shareholder.

The history of Symbian OS [Tasker 00] goes back to 1981, and started with Psion that manufactured personal electronic organizers. The early organizers were equipped with a built-in programming language OPL (Organizer Programming Language). Technological innovations lead to the need for a new operating system, which became a 16-bit system called SIBO, which allowed for execution of more complex applications. The first machine that deployed SIBO flopped, but many devices after it where a huge success for Psion.
By 1994, Psion realized that SIBO's 16-bit architecture was beginning to suffer from its limitations in such a way that an entire rewrite would be needed to provide a platform for long-term future development. The design effort that followed resulted in the first release of EPOC, a 32-bit operating system, written in C++. In 2001, Symbian released its version 6.1 of the EPOC system, and at the time of writing, the last steps are being taken for the release of version 7, which is specially tailored for communicators and smart phones.

As Symbian currently puts it, Symbian OS is the advanced, open operating system licensed by the world's leading mobile phone manufacturers. It is designed for the specific requirements of advanced 2G, 2.5G and 3G mobile phones. Symbian OS includes a robust multi-tasking kernel, integrated telephony support, communications protocols, data management, advanced graphics support, a low-level graphical user interface framework and a variety of application engines.

As Symbian OS is the current name for EPOC, for clarity I will use the former of these when referring to either.

4.2 Symbian OS architecture

The architecture of Symbian OS (version 7) is given in Figure 9 [Mery 02]. Older versions of Symbian OS have an architecture that is globally the same. It is important to note that this figure does not depict API calls, but rather dependencies of the different parts of the system.

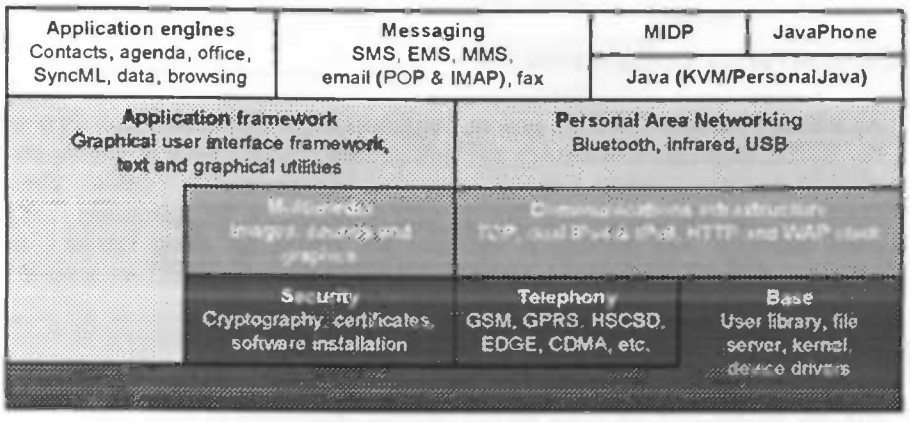


Figure 9: Symbian OS v.7 architecture (subsystem dependencies)

The operating system and its applications are divided into three types of components that are separated by boundaries. An overview of basic components and their boundaries is shown in Figure 10.

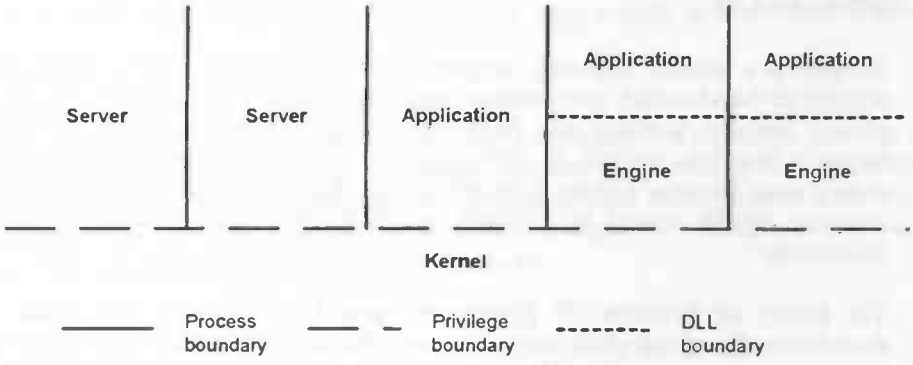


Figure 10: Symbian OS software components and boundaries

These basic components in Symbian OS, according to [Tasker 00], are:

The **kernel** – a software component that runs in a hardware-supported privileged mode. The kernel's task is to manage the machine's hardware resources such as system RAM and hardware devices. Other programs depend on the services supplied by the kernel for access to these resources.

Applications – software that interacts with the user of the system. In Symbian OS, each application runs in a separate process, with its own virtual address space.

Servers – an application that does not have a user interface, and thus not interact with users directly. It provides services to its *clients*, which can be application programs or other servers. The task of a server is to manage one or more resources (in the broadest meaning) in the system.

engines – the reason for using application engines is the benefit of reusing engines and making a clear separation between user interface and 'business logic', which is in general good programming practice. The boundary between engines and their applications is not a protection boundary, such as the privilege and process boundaries are, but rather is a boundary for promoting good programming practice.

In the following sections, I will go deeper into the concurrency constructs that Symbian OS provides.

4.3 Concurrency constructs

As with any modern operating system, Symbian OS provides constructs for concurrent execution. For this, it adopts the well-known and well-studied concepts such as threads and processes, but in addition to this, a construct that is known as 'Active Objects'[6] (AOs). The choice for Active Objects is, according to [Tasker 00], a result of the choice for optimizing the system for efficient event handling. The decision to optimize the system for event handling was driven by the observation that GUI systems (which PDA-systems, in essence, are) are event-driven in nature.

[6] It is worth noting that in literature (e.g. [Bergmans 94], [Gomaa 93], [Burns 01], [Sommerville 01], see section 5.3), there are several different definitions of what active objects are. Therefore, unless stated otherwise, when mentioning the term *active object*, I will refer to Symbian's implementation thereof.

4.3.1 Processes and threads

Symbian OS supports preemptively scheduled threads and processes with hardware-enforced virtual address spaces for processes. A more detailed description of threads and processes is given in section 3.4.

4.3.2 Event-handling

PDA-systems are user-interaction oriented. To illustrate this, an example [Tasker 00] is given in Figure 11, which shows a cascade of events that happen when a keyboard interrupt is received. The horizontal lines *Kernel/driver*, *Window server*, and *Application* all denote separate threads of execution. In the example, the following happens:

1. An interrupt is generated by the hardware that controls the keyboard,
2. control is transferred to the kernel that executes a short so-called Interrupt Service Routine (ISR), after which an event is generated and control is transferred to an application that is 'interested' (i.e. has indicated that it can handle this type of events),
3. the interested application (in this case the window server) does some processing on its behalf (e.g. checking which application is currently receiving keystrokes) and sends the events to that application,
4. the application handles the event (e.g. by performing an action like adding a character to a text field), after which (in this example), it issues an event to the window server to update the window,
5. the window server performs the updating of the window

Although this example is only a simplification of the real situation, a few observations can be made:

- understanding the flow of events is simple, thus making it easier to verify correctness of behavior,
- when the system is idle, the system can be put in a power conserving state,
- each task in such a system is, in essence, an event handler,

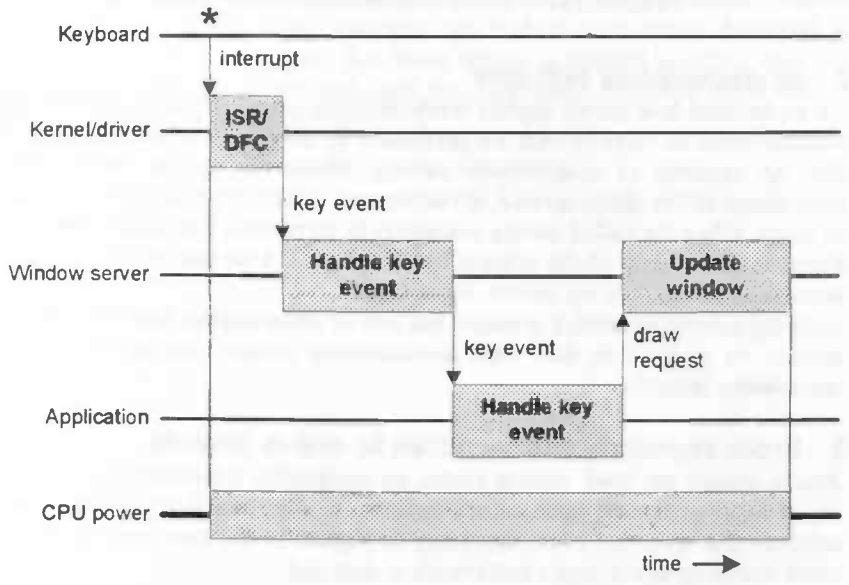


Figure 11: Event-handling example ([Tasker 00], pg. 97)

In Symbian OS, active objects are used to handle events. In the following section, I will describe why and how Symbian has implemented the event-handling paradigm.

4.4 The basics of Active Objects

The developers of Symbian OS have chosen to adhere to an event-handling framework that globally consists of a so-called *active scheduler* (AS) in combination with one or more active objects. In the following sections, I will explain the basics of active objects. After this, I will explain how active objects can be used in a different way, namely to implement long-running tasks.

4.4.1 Process, threads and active objects

As said, Symbian OS has native support for processes and threads. As no thread can exist without a corresponding, owning, process, so can no active scheduler exist without an existing thread or process. As opposed to processes, which can own multiple threads, each thread (and therefore, also each process, as they can be seen as a thread as well) in Symbian OS can contain only *one* active scheduler. This relation is shown graphically in Figure 12.

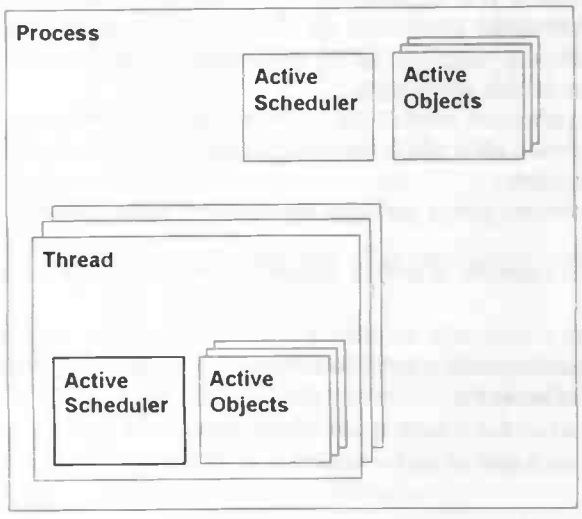


Figure 12: Processes, threads and active objects relation diagram

4.4.2 Asynchronous services

To understand how active objects work, it is important to understand the underlying principle of *asynchronous* services (which are performed by *service providers*), as they are called in Symbian OS. As opposed to synchronous calling, where the calling entity waits for completion or termination of the called service, asynchronous calling *requests* a service and continues processing or *waits*. When the called service completes or terminates, the calling thread is notified by a signal. Between the issuing of the request and the signal, a service is said to be *pending*. Examples of asynchronous services are timers, sockets etc.

Each asynchronous service provider has one or more request functions that are used to initiate the service. In addition to this, each asynchronous service provider has a function to cancel the outstanding request.

4.4.3 From asynchronous services to active objects

Active objects are used, among others, to encapsulate asynchronous service providers. An active object supplies request and cancel functions, as well as a function that handles the completion of requests (the so-called RunL-function). In Figure 13 the basic relation between the objects in an event-handling active object framework is depicted.

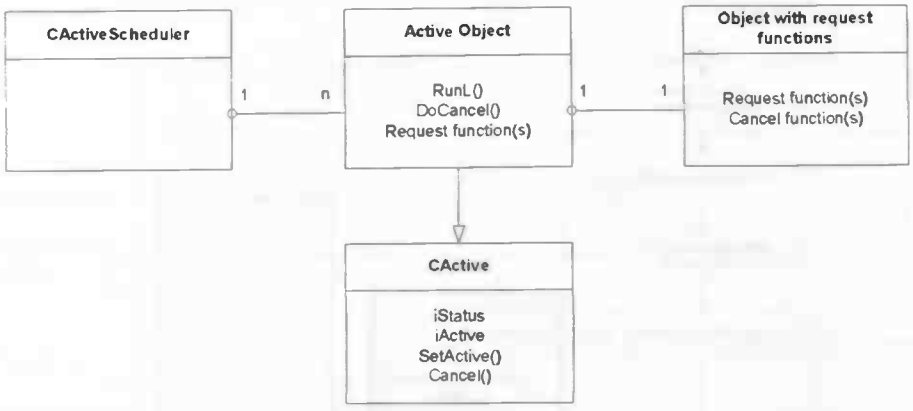


Figure 13: Object-relation diagram for an event-handling thread

In this picture, the object with request functions can be a service provider, or another active object, which in turn can call other active objects etcetera.

Figure 14 shows the life cycle of an application with one active object, together with its context. The solid lines indicate procedural flow; the dashed lines show the flow of control. What happens in the figure is the following: the active object framework is initialized by installing an active scheduler (and only one), creating active objects and adding them to the scheduler (top-left block). Then, an active object is called to issue a request, which results in that active object being marked *active* and the thread is said to have requests *pending*. Such an initial request is mandatory before the active scheduler is started. When the active object is called to issue a request, it calls some service provider (it can e.g. request the expiry of some timer). The service provider sets a flag (`iStatus`) of the active object, upon which control returns to the active scheduler[7].

When the system has requests pending, the active scheduler waits until it is signaled by a service handler that a service has completed. This is done by signaling the application's *request semaphore*. If this happens, the active scheduler scans the active objects in its queues to check for an active object that matches that signal, and invokes that object's so-called *RunL*-function for handling the completed request. If the scheduler can find no such object, the signal is said to be *stray*. Upon calling the RunL-function, the active object is marked inactive, until it is called to issue a new request. This can be repeated until the application is closed, upon which the created structures are released and memory is freed[8].

[7] This obviously is only the case when the service provider is in a separate thread.
[8] Actually, Symbian OS uses an extensive garbage collecting framework/cleanup stack to prevent memory leakage.

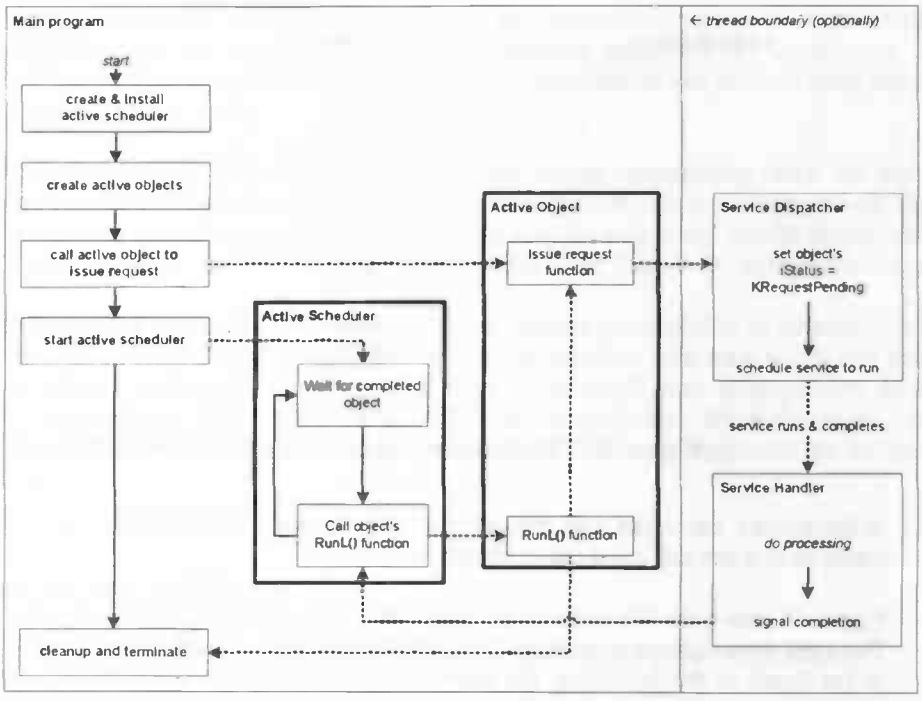


Figure 14: Flow diagram showing the life cycle of an application with one active object

In Figure 15, this behavior is illustrated with a diagram that shows the interaction between different entities through time. The part to the left of the tear lines is performed at application start-up. (Of course, there will be a number of create/add-operations if there are multiple active objects). After this, the application behaves as explained above.

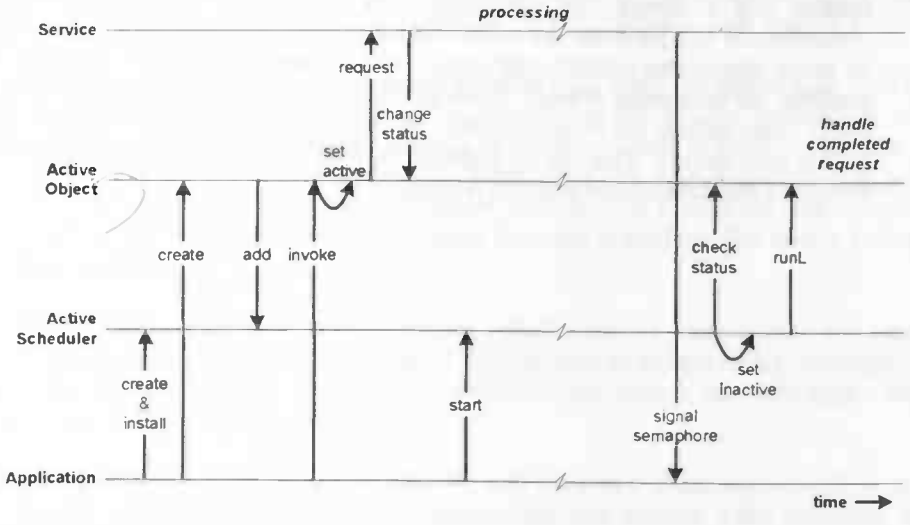


Figure 15: Active Object example time-relation diagram with one Active Object and an asynchronous service

Figure 16 schematically shows the four ways in which a request can complete [Tasker 00]: exit with an error, run to completion, being canceled before completion and being canceled after completion (which in essence is ignored).

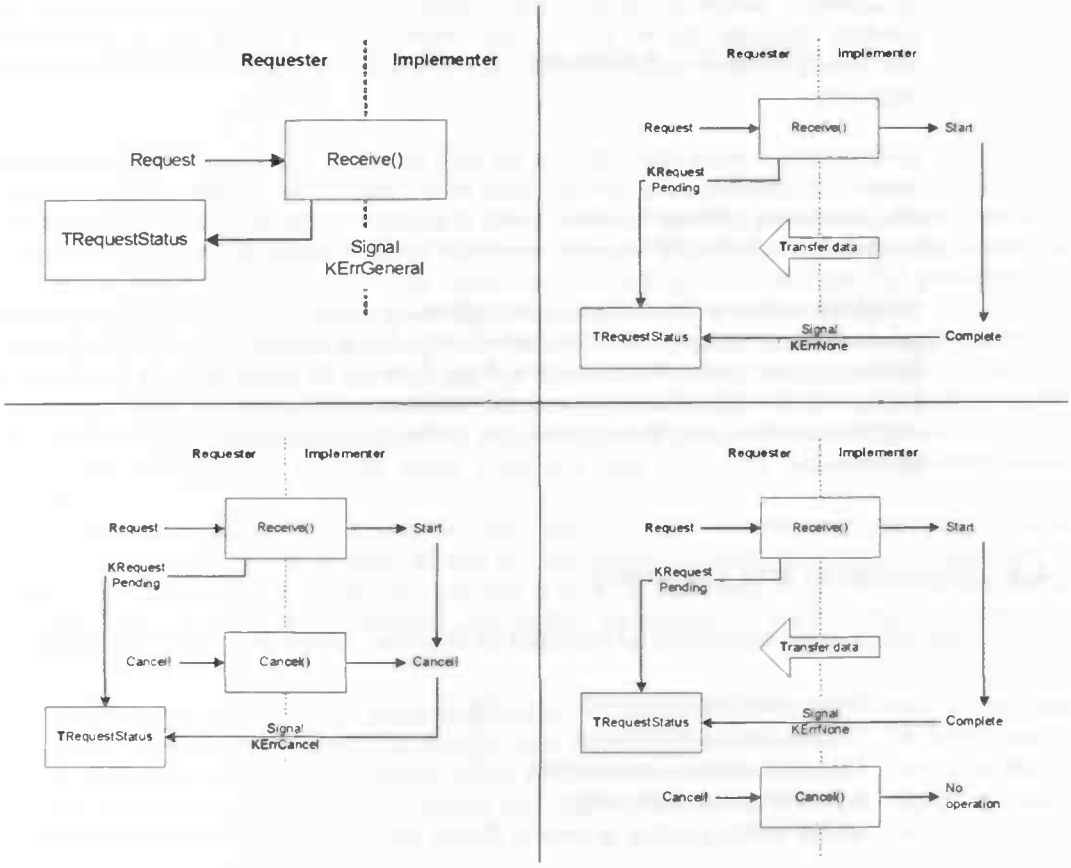


Figure 16: Four ways to complete a request. Top-left: error on initialization, top-right: normal completion, bottom-left: cancel before completion, bottom-right: cancel after completion

4.4.4 Implementing long-running tasks

Active objects such as used in Symbian OS are non-preemptable. That is, their RunL-function must complete before any other active object can be scheduled for execution. Therefore, when implementing long-running tasks, special measures have to be taken to prevent application lock-up. The way to handle long-running tasks with active objects is for the active object to re-schedule to reschedule itself by 'renewing' its request. In this way, the active object is rescheduled for execution. However, because active objects are non-preemptable and their state is not saved by the scheduler when the RunL-function finishes, when an active object's RunL-function is called, it will execute in its entirety. Thus when an active object would have a long-running task in its RunL-function the application would still be blocked for the entire execution period of the RunL-function. Clearly, this behavior is undesirable. The way to overcome this is to split the long-running task into shorter sub-tasks, and making the active object responsible for retaining its state (the active object becomes a state machine). In this way, every time the active object's RunL-function is called, it could execute a different sub-task, thus proceeding through the long-running task.
This way of implementing long-running tasks is a form of cooperative concurrency, which bears strong resemblance to so-called 'coroutines' as described in [Burns 01].

Symbian OS uses long-running tasks as described here for, for example, redrawing of secondary components, such as toolbars and status bars, with low-priority active objects.

4.4.5 Issues

Although active objects have a number of obvious benefits, there are also a number of issues when using active objects:

Timeliness – as mentioned earlier, active objects are non-preemptable. Although this might be beneficial from ease-of-programming point of view, it can compromise timeliness of applications: when a very high priority task has to be executed, while some other active object

is running, it cannot be executed immediately. Thus, when there are time-critical events to be handled, choosing for an active object-based solution might not be satisfactory. As an alternative, a multithreaded solution can be used, where threads can preempt each other when necessary.

Granularity – when active objects are used to implement long-running tasks, but there are a number of (possibly long-running) tasks to be executed in the main program thread, the tasks may have to be split into too short parts, in which case the switching[9] overhead may become a non-negligible factor. In this case, one might consider using threads for some of the tasks.

Breaking well-written routines – although most routines can probably be converted to a state machine form, doing so might be undesirable. Some routines have been tested and altered to deliver highest performance, and breaking them up in small steps to implement them with active objects might result in non-trivial performance degradation. When this is the case, one might consider keeping the routine intact and using a multithreaded approach for the routine in question.

4.5 Discussion and summary

The active object framework, as described by Symbian, can be characterized as follows:

- Event-handling framework
- Nonpreemptive scheduling
- Unit of execution: active object
- Scheduler: active scheduler
- Works within a thread or process (hence not visible to the operating system)

When looking at the framework and its issues as described in the previous sections, there are a number of questions that can be posed with respect to the active object-framework. For example, how system responsiveness is influenced by complex 'event cascades' (see Figure 11) when a system is scaled up.
While active objects within one application cannot preempt each other, the application itself can be preempted. Such behavior might influence active objects that are timing- or resource related. Furthermore, little is known about timing behavior of the framework in other application areas. Therefore, as a rule of thumb, no assumptions on timing should be made whatsoever. In addition to this, when creating a long-running task with active objects, how should the code be split and should it be split at all? When designing applications from the ground up, this means a change in programming paradigm.

Symbian OS is a modern operating system, written with the special requirements that resource-constrained devices pose in mind. The developers have chosen to adhere to a lightweight event-handling system that seems to be working very well with the mainly user-interaction oriented devices the system is targeted for.

Although the framework has its beauty in memory and overhead efficiency as well as ease of programming, an engineer should take these considerations into account when designing for the active object framework.

[9] As defined in the context of active objects and active schedulers

5 Concurrent Object-Oriented Programming (COOP)

5.1 Introduction

Traditionally, software was written from a functional or procedural perspective. This means that software was designed from a point of view that dictated what the product had to do, instead of looking at how the product had to be structured. Throughout the years, the OO paradigm has become a useful tool in software engineering in helping structuring software in a clear way.
It is beyond the scope of this report to give an in-depth description of the OO paradigm. Therefore, I will limit my description of the concepts on which the OO programming model is built. OO focuses on entities (*objects*) that combine data and operations (*methods*) thereon. The data within the object is hidden (*encapsulated*) within the object boundaries, and can only be changed from outside the object by invoking the object's methods (that is, objects could be considered black boxes).
In the following sections, I will limit my introduction to OOP to what is relevant, and address some of the benefits of this method, as well as some issues this method brings. I do not claim to address all issues of (Concurrent) OOP, as this is an area of research in its own right, and this is not the focus of this report. Therefore, for further information, I refer to the literature (e.g. [Bergmans 94], [Meyer 93]).

Objects and classes – As mentioned before, objects are entities that combine data and operations thereon. In addition to objects, there is a concept known as *class*. An object class is an abstraction of a group of objects, which defines their common behavior. Apart from this, it also functions as a template for creating new objects. Objects derived from classes are called *instances*.

Methods and attributes – Objects and classes (should) hide the data (also known as the object's *attributes*) they contain for the outside. That is, observers should not be able to modify the data within the object directly (*encapsulation*). For modifying the object's attributes, the object's methods should be invoked. However, not all of the object's attributes may need to be visible to the outside. This concept is called *information hiding*.

Inheritance – Inheritance is the notion reusing the specification of a class to form a new class that may extend or override functions of the class(es) it was derived from.

Conceptually, invoking an object's method could be considered as requesting a service from an object. In most OO-languages, method invocations are implemented as function calls. There is, however, no requirement in the OO paradigm that states that this is the way it has to go. When objects are well written[10], there is (at least conceptually) no objection to executing objects in parallel. The concept of executing objects in parallel leads to the notion of *concurrent object-oriented programming* (*Concurrent OOP* or *COOP*). Because of their characteristics, objects are, at least conceptually, well suitable for distributed computing. This might seem as a simple and logical step, but there are a number of issues that one has to deal with, when running objects concurrently. A number of these problems are the ones that generally arise when tasks run concurrently (see section 3), such as remote procedure calling (which is especially important in COO), synchronization etc.. Others, however, are related specifically to concurrent objects.

5.2 Types of object concurrency

There are two types of object-related concurrency: *inter-object* and *intra-object* concurrency. In the former, each object is an entity executing separately from other objects, while the latter provides multiple threads of execution within one object. Both methods have their influence on

[10] That is, they fully encapsulate their data, and provide suitable methods for accessing that data, as well as hide their internals etc.

how objects can be used, especially concerning reuse and inheritance (which is also a kind of reuse). For more information, see [Bergmans 94], [Nierstrasz 93].

5.3 Different active objects

As opposed to the definition given in the description of Symbian OS in section 4.4, literature on concurrent object-oriented programming mentions different definitions of what an active object is. [Gomaa 93] mentions active objects as *'active objects are concurrent tasks'*, while [Sommerville 01] describes active objects as objects that continually execute operations that may change the object's state. Both descriptions globally arrive at the same point: active objects are objects with a thread of control.
[Bergmans 94] on the other hand, describes active objects as objects that have control over what messages to accept, whereas [Burns 01] speaks of active objects that undertake spontaneous actions.
In this report, I will adhere to Symbian's definition of active objects, and mention the other definitions where necessary.

5.4 Advantages of Concurrent OO

Using objects has many benefits from a software engineering point of view, such as modularity, reusability, verifiability, composition, abstraction etc. Object models help us to construct a model of the real world [Nierstrasz 95], which helps in creating a conceptual model about which it is easier to reason. In the line of reasoning of modeling the real world, COO is a logical step, or as Wegner (see [Bergmans 94]) puts it: *'the real world is concurrent rather than sequential'*. One of the benefits of OO is that the object models can be used throughout the analysis, design and implementation phases. In addition, there has been thorough research after OO and there are a great number of tools available for software engineers.
The benefits of object-orientation apply to large extent to concurrent OO as well: when objects are written with concurrent execution in mind (e.g. for distributed systems), they can be a clean way of writing distributed application. When, on the other hand, objects are well implemented (according to the OO paradigm) with sequential execution in mind, they can be reused in other projects (cf. the *black box*), but might give problems when executed in parallel. I will discuss a number of these problems in the following sections.

5.5 Level of synchronization

[Bergmans 94] mentions two types of message synchronization that lie *past* the object boundary, these are synchronization at the code level (by e.g. semaphores, mutexes etc.), and synchronization at the object level (this is the active object in Bergmans' definition). Of these two, the latter is preferable, as it more closely resembles the notion of self-sufficient objects.

5.6 Problems with Concurrent OO

5.6.1 Reuse of sequential code

When objects are used as black boxes (that is, it does not matter how they are implemented), problems may arise when they are used in a concurrent fashion. Generally, no problems will arise with objects that just respond to messages without changes their internal state. However, when objects change their internal state upon reception of messages, corruption of data might occur. A solution herein lies with the programmer to take care of not simultaneously allowing access to an object by either encapsulating it in some kind of object that controls access, but this might compromise the 'cleanness' of the design.

Furthermore, an object written for execution in a nonconcurrent environment[11] (in short 'sequential object') but is operating in a concurrent one can influence the rest of the environment [Nierstrasz 95], because of the synchronous way these objects' services are requested. Therefore, other objects may have to wait until the object has replied. This clearly can influence the system's performance and thus limits the usefulness of reuse of some sequential objects.
Of course this problem can be overcome by known techniques (such as proxies), but this again breaks the cleanness of the design.

5.6.2 Inheritance anomalies

Most problems in concurrent execution of objects deal with synchronization mechanisms. In particular, there is a problem known as *inheritance anomaly*[12]. Inheritance anomalies arise when synchronization code is reused. In short, inheritance anomaly means that when an object with synchronization code is subclassed, some parts of the parent class might need to be re-implemented, which breaks the concepts of abstraction, polymorphism and encapsulation.
Inheritance anomalies can also occur when a class encapsulates a cyclic scheduling approach and that class is subclassed – there is virtually no way, in which the cyclic scheduler can be extended or changed without completely rewriting the scheduler.

5.6.3 Request/reply scheduling needs multiple threads

When objects are to support reply scheduling, it is important to be able to suspend running methods to service the request. To do so, the object needs multiple threads [Nierstrasz 95]. Clearly, seen from resource constrained (e.g. mobile) systems, this is undesirable.

5.7 Summary

The OO paradigm has proven to be powerful and, when combined with the benefits of concurrency, well-structured and efficient software can be written. Usually, there is no objection to using both the object-oriented paradigm and concurrency in a language. This brings the software engineering benefits of the object-oriented paradigm together with the (possible) efficiency increase of concurrency.
Although the combination of two techniques might sound as the way out for every software engineer, research has shown that simply integrating concurrency in an object-oriented language can introduce problems (e.g. inheritance anomalies). The solutions proposed often incorporate using additional code that guards the correct working of a system, which may not be desirable in resource-constrained systems. Other solutions involve changing the object-oriented language itself, which might not be an option.

Concurrent OO is not a technique in itself, but rather a combination of concurrency and OO. Its concurrency characteristics mostly depend on the concurrency solution it uses, and therefore no general concurrency-related characteristics can be given.

[11] The 'environment' can be e.g. an application or a (distributed) system.
[12] It is important to note that inheritance anomalies are not inherent to the combination of OO and concurrency, but rather follows from the synchronization scheme and inheritance semantics of a language.

6 Assessing Concurrency Constructs

6.1 Introduction

This chapter takes the results from the previous chapters to assess the different concurrency constructs with respect to the requirements posed. The characteristics have been derived by a requirements engineering process in chapter 2 and are based on concurrency requirements for a future mobile device.

The aim of this chapter is to produce –per construct– an overview of how well each requirement can be met by the selected method. The reason for doing so is in helping selecting or evaluating a concurrency solution for a given problem.

6.2 Characteristics and Constructs

6.2.1 Characteristics

The characteristics ('requirements') that are used in this chapter are the ones as derived in chapter 2. Table 4 lists the characteristics once more:

Characteristics		
Runtime binary code reuse	Real-Timeliness	Low adaptivity
Effective event handling	Resource reuse	Memory & CPU distribution
Fault tolerance	Scalability	Performance over efficiency
Flexibility / adaptivity	Task prioritization	(Prot.) unit of execution
Highly available	Transparency	Redundancy support
Memory efficiency	Verifiability	Schedulability of tasks
Openness (form factor)	QoS support	Synchronization
Power efficiency	Fairness of scheduling	Virtual memory
(Preemptive) multitasking	Generality over dedication	

Table 4: Characteristics

6.2.2 Constructs

The concurrency constructs that are assessed in this chapter are:

- Cyclic Schedulers
- Processes
- Threads
- Active Objects

The Concurrent OO solution is not included, because it depends too much on the choice for the methods above.

6.3 Assessing the solutions

This section relates the requirements from chapter 2 to the different specifics of the four-concurrency constructs. After that, a classification is made and the requirements are summarized per construct in three groups named 'supported', 'unsupported' and 'unknown/depending'. In some cases these terms can also be read as 'is (not) a general property'.

The second step in the assessment is the graphical representation of the data: requirements and constructs are listed in a table, in which only those requirements are taken into account that are classified differently for each construct.

The summaries and table(s) can then be used for selecting a construct for the given problem, or for evaluating a solution, that uses a certain construct.

6.3.1 Cyclic Schedulers

Section 3.6 lists a number of characteristics for cyclic schedulers. This section relates the requirements from chapter 2 to the different specifics of the cyclic scheduling approach:

Scheduling by repeating function calls – because of the hard coding of function calls in the naïve version of the cyclic scheduling approach the solution is inflexible and run-time adaptivity is low. Support for task priorities has to be hard-coded by repeating function calls. Because of the tight function call-loop, the cyclic scheduling approach in its naïve form is power-inefficient. Scalability of such a tight system is disputable as the interval between to calls to the same function increases – such might negatively influence systems that depend on strict timing (generally systems that communicate with hardware). As the scheduling is in a round-robin fashion, scheduling can be considered fair. Because of the lack of preemption, real-timeliness and the ability to handle events are disputable.

Unit of execution: function – because each function call runs to completion (function calls are not preempted), synchronization is handled implicitly. A clear benefit of using function calls is that there is virtually no memory and CPU overhead involved in 'scheduling' a function.

All entities run in the same memory and context – All functions share the same memory and context, which makes it possible for a function to corrupt the application/system, which is clearly not beneficial for fault tolerance. A benefit of sharing memory and context is that a resource allocated by the main application can often be reused by the tasks.

Inflexible: no prioritization and adding/removing entities – as the scheduling order, as well as the tasks to be run, are defined at build time, a cyclic scheduler is inflexible in its naïve version. This means that runtime adaptability is low. Therefore, a cyclic scheduler in its naïve form might be best suited for systems that require low adaptivity.
Section 3.6 shows that cyclic schedulers can be adapted to overcome a number of these issues.

No memory or switching overhead – the overhead involved in calling a function is negligible. As the 'scheduler' itself is consists only of a number of function calls, there is no overhead there as well.

'Niche-solution' – cyclic schedulers can, because of their limitations, and the availability of better[13] solutions, be seen as solution in only certain cases (e.g. for small and/or resource-constrained systems), which makes them a niche-solution.

The classification above can be summarized as follows:

Supported – memory efficiency, resource-reuse (sharing), verifiability, fairness of scheduling, low adaptivity, synchronization

Unsupported – effective event handling, fault tolerance, flexibility/adaptivity*, openness, power efficiency, (Preemptive) multitasking, scalability, task prioritization*, generality over dedication, memory & CPU distribution, performance over efficiency, protected unit of execution, redundancy support, schedulability*

Unknown/depending – run-time binary code reuse, real-timeliness, transparency, QoS support, virtual memory, highly available

*-marked items concern characteristics that are supported in the adapted/extended versions of the cyclic scheduling approach.

[13] Better in most cases, that is

6.3.2 Processes

Section 3.4 characterized processes by the following four descriptions:

Protected unit of execution – having a protected unit of execution is beneficial for system stability and fault tolerance as no process can corrupt another (directly). Redundancy and fault tolerance can be supported by assigning multiple threads for handling a task.

Scheduled by the operating system – generally, processes are scheduled preemptively by the operating system, which makes verifiability harder than nonpreemptive scheduling. Generally, an operating system and its scheduler support task prioritization (sometimes Real-Time scheduling as well), synchronization, virtual memory, and sometimes memory and CPU distribution. Events can be handled as high-priority (real-time) tasks. If the operating system supports it, power efficiency can be achieved by lowering power usage when the system is idle. In spite of this, because of the switching overhead, processes are less efficient in CPU usage than threads. Because of the 'central' scheduling by the operating system, QoS support is possible.

Own context (virtual address space, file descriptors etc.) – having a protected context means protection for the process, but creates memory overhead. Also, resource reuse (sharing a resource between two processes) requires communication and or synchronization mechanisms.

Can own one or more threads – owning and the ability to create and destroy threads provides run-time flexibility and adaptivity. Processes can do this transparently to the other processes in the system.

Summarizing the classification:

Supported – effective event handling, fault tolerance, flexibility/adaptivity, openness, (preemptive) multitasking, scalability, task prioritization, generality over dedication, memory & CPU distribution, performance over efficiency, protected unit of execution, redundancy support, schedulability, fairness of scheduling, synchronization, run-time binary code reuse, QoS support, virtual memory, real-timeliness, transparency, high availability

Unsupported – memory efficiency

Unknown/depending – low adaptivity, resource reuse (sharing), verifiability, power efficiency

6.3.3 Threads

Threads are conceptually the same as processes, but lack a number of protection facilities that processes have. In section 3.4, threads were characterized by the following descriptions:

Unprotected unit of execution (within one process) – a process can contain one or more threads. All threads are protected from other processes (and their threads) by the containing process. Within one process, threads are not protected against each other – therefore a stray thread within a process can corrupt the whole process. Fault tolerance within a process is therefore lower than with a pure multi-process solution.

Scheduled by the operating system – generally, threads are, just as processes scheduled preemptively by the operating system, which has the same verifiability issues as mentioned with processes. Scheduling, prioritization and synchronization is generally the same as with processes.

Owned by a process or other thread – there is no real issue here that is not already covered by one of the other characteristics mentioned here, so no special remarks are concerned with this characteristic.

Memory and context shared with other entities within owning process – because all threads share the memory and context of the owning process, resource reuse requires no

special communications mechanisms. However, when necessary, synchronization issues will have to be solved within the process to prevent two or more threads from corrupting a resource. Because of the sharing of memory, a distributed memory solution is harder with threads than with processes.

Can own one or more threads (depends on system) – threads can create and destroy other threads within one process. This allows for run-time flexibility and adaptability. Together with the property that memory and context are shared, redundant solutions within one process can be built, or problems can be split in sub-problems that are handled by multiple threads.

Has less overhead in creation, destruction and switching than processes – an advantage threads have over processes is that their overhead in creation, destruction and switching between threads is much more efficient than with processes. They are therefore more efficient in memory and CPU usage.

Summarizing the classification:

Supported – effective event handling, memory efficiency, flexibility/adaptivity, openness, power efficiency, (preemptive) multitasking, scalability, task prioritization, generality over dedication, performance over efficiency, redundancy support, schedulability, fairness of scheduling, synchronization, run-time binary code reuse, QoS support, virtual memory, real-timeliness, transparency, resource reuse (sharing)

Unsupported – fault tolerance, memory & CPU distribution, protected unit of execution

Unknown/depending – low adaptivity, verifiability

6.3.4 Active Objects

The active object framework, as described in section 4.4, works within a thread or a process, and therefore shares, when used in an application, a number of properties. However, the framework itself can be characterized as follows:

Event-handling framework – as the name already says, the active object framework is tailored to handling events. As the scheduler should handle all events, application behavior should be easy to verify. It is hard to tell if a system consisting solely of an active scheduler and active objects can be used in situations where currently threads and processes are used.

Nonpreemptive scheduling – as tasks are scheduled nonpreemptively, code can be written without specific synchronization constructs, as these are implicit in the scheduler. Such behavior makes code easier to verify and easier to write. Because of being nonpreemptive, no extra memory is needed to save the state of a task when switching. In addition to this, there is no need for specific synchronization constructs.
The lack of preemption has the drawback of not being able to guarantee timeliness of tasks. The active object framework can be characterized by trying to achieve efficiency over performance.

Unit of execution: active object – the unit of execution is an active object – a programming construct from sequential programming of which the instances are responsible for handling completed requests ('events'). Active objects do not have more memory overhead than other objects in a programming language, and can therefore be considered efficient in memory usage.

Scheduler: active scheduler – the scheduling entity in the framework is the so-called active scheduler. Active objects are added to the scheduler and are run when the request they made completes. This means the system is flexible and extensible, but currently not enough is known about how the system scales.

Works within a thread or process (hence not visible to the operating system) – as with threads, the AO-framework works within an owning process (or thread). Therefore, the

memory and context of the owning entity are shared, which makes reuse of resources within an application easier.

Summarizing the classification:

Supported – effective event handling, memory efficiency, flexibility/adaptivity, openness, power efficiency, verifiability, multitasking, task prioritization, schedulability, synchronization, run-time binary code reuse, transparency, resource reuse (sharing)

Unsupported – fault tolerance, memory & CPU distribution, protected unit of execution, performance over efficiency, redundancy support, real-timeliness

Unknown/depending – low adaptivity, scalability, generality over dedication, fairness of scheduling, QoS support, virtual memory

6.4 Documentation and Selection

To gain insight in which construct is, or which constructs are best for the given problem, one can represent the results from the previous sections graphically. Say, in a table. In such a table, each characteristic is listed with a symbol indicating their classification (e.g. S for supported, U for unsupported and D for unknown/depending). The table lists only those characteristics that are classified differently for each construct (see Table 5).

Characteristic	Cyclic Scheduling	Processes	Threads	Active Objects
(Preemptive) multitasking	U	S	S	S
(Prot.) unit of execution	U	S	U	U
Effective event handling	U	S	S	S
Fairness of scheduling	S	S	S	D
Fault tolerance	U	S	U	U
Flexibility / adaptivity	U	S	S	S
Generality over dedication	U	S	S	D
Highly available	D	S	S	S
Low adaptivity	S	D	D	D
Memory & CPU distribution	U	S	U	U
Memory efficiency	S	U	S	S
Openness (form factor)	U	S	S	S
Performance over efficiency	U	S	S	U
Power efficiency	U	D	S	S
QoS support	D	S	S	D
Real-Timeliness	D	S	S	U
Redundancy support	U	S	S	U
Resource reuse	S	D	S	S
Runtime binary code reuse	D	S	S	S
Scalability	U	S	S	D
Schedulability of tasks	U	S	S	S
Task prioritization	U	S	S	S
Transparency	D	S	S	S
Verifiability	S	D	D	S
Virtual memory	D	S	S	D

Table 5: Overview of differing characteristics for the constructs cyclic scheduling, processes, threads and the active object framework

In the table above, the only characteristic from the original list that is not included is 'synchronization'. The goal of the method that is described in this thesis is to provide a tool that can help in selecting or evaluating a choice. Doing this, requires the amount of information to be condensed, which is clearly not the case with the table above.
A solution to this is to take a sub-set of constructs. The sub-set should preferably consist of constructs that are alike, such as processes and threads (see Table 6), or the cyclic scheduling

approach and the active object framework (see Table 7). Again, the tables list only those requirements that are classified differently.

Characteristic	Processes	Threads
(Prot.) unit of execution	S	U
Fault tolerance	S	U
Low adaptivity	D	D
Memory & CPU distribution	S	U
Memory efficiency	U	S
Power efficiency	D	S
Resource reuse	D	S
Verifiability	D	D

Table 6: Overview of differing characteristics for the constructs processes and threads

Table 6 shows a narrowed-down overview of requirements and the constructs processes and threads. In the table, two requirements/characteristics are not classified differently, namely *low adaptivity* and *verifiability*. The reason that these are list is trivial: the requirements were classified 'depending', and should therefore probably be taken into account.

Characteristic	Cyclic Scheduling	Active Objects
(Preemptive) multitasking	U	S
Effective event handling	U	S
Faimess of scheduling	S	D
Flexibility / adaptivity	U	S
Generality over dedication	U	D
Highly available	D	S
Low adaptivity	S	D
Memory & CPU distribution	U	U
Openness (form factor)	U	S
Power efficiency	U	S
QoS support	D	D
Real-Timeliness	D	U
Runtime binary code reuse	D	S
Scalability	U	D
Schedulability of tasks	U	S
Task prioritization	U	S
Transparency	D	S
Virtual memory	D	D

Table 7: Overview of differing characteristics for the constructs cyclic scheduling and active objects

Table 7, which lists cyclic scheduling and active objects is longer than the previous table. The reason for this can be that, although both have the same basic principle, the active scheduler is far more sophisticated than the cyclic scheduler is.

With help of tables, such as the ones above, an engineer can choose for certain constructs. When there are multiple candidates, the engineer will have to prioritize (or, when available, use a prioritization from the requirements specification document) the requirements.

6.5 Summary and Conclusions

In this chapter, an attempt was made to make a classification for the different requirements for each concurrency construct. Mostly, when a requirement was placed in a certain group, the choice was driven by known facts from literature or by common sense. Requirements in the group

'unknown/depending' have been classified thus, because their characteristics depend too much on specific situations or other factors, or sometimes, because they were not relevant.

One can discuss the choice of requirements as used in this chapter. Some of the requirements used here are clear, others rather vague. The vagueness is not a goal, but a situation like this is, however, a situation that might occur in the requirements analysis phase for a real product.

The concurrency constructs and requirements as described in this chapter are not solely for the mobile device discussed in chapter 2, they are (far) more general. Therefore, the outcome of the analysis might be used in selecting or evaluating other (resource-constrained) systems as well.

The classification as made shows overlap between the different constructs and requirements. This chapter cannot, and does not, aim to give an all-in-one solution for a problem that requires concurrency, but rather aims to help in selecting or evaluating a concurrency solution for a given problem. The choice for one of the constructs, when building the mobile device for real, depends not only on the requirements as defined in chapter 2, but also on a range of other requirements and constraints for that product.

When designing a product, tradeoffs between conflicting requirements have to be made. In such a case, an analysis phase as shown in this chapter, together with a requirements analysis process as shown in Table 1 can help in relieving the pain of choice. Making a choice for a solution might therefore be made easier by carefully analyzing the requirements and taking the best alternative.
In requirements engineering and analysis, it is common practice to prioritize requirements, which means indicating the importance of every requirement (with respect to the others). Such a prioritization can help in selecting an optimal solution for the problem. Here, such a prioritization has not been made, which makes it harder to point out *the* optimal solution. Had such a prioritization been made, the results from this chapter might be suitable in further development of the synthesized mobile device.

7 Prototyping I – A control example

In this chapter, the implementation of a simple control system will be used to validate the results from the previous chapter in a real-world context. The example system will be implemented using three different methods: a cyclic scheduling version, a version using the active object framework, and a version that uses traditional concurrency constructs (threads). In addition, each implementation will be extended to gain insight in how the different methods perform under these changed conditions.

The example system used in this chapter is a simplification of a system such as could be found in a chemical process control system. The example is taken from [Burns 01]

7.1 The system

Figure 17 outlines the system: a process T takes readings from a set of thermocouples via an analogue-to-digital converter ADC and makes appropriate changes to a heater through a digitally controlled switch. Process P has a similar function, but is concerned with pressure (measurements delivered by a digital-to-analogue converter DAC). Both T and P communicate data to S, which presents measurements on a screen. Note that where P and T are active; S can be seen as a resource (it processes data from T and P).

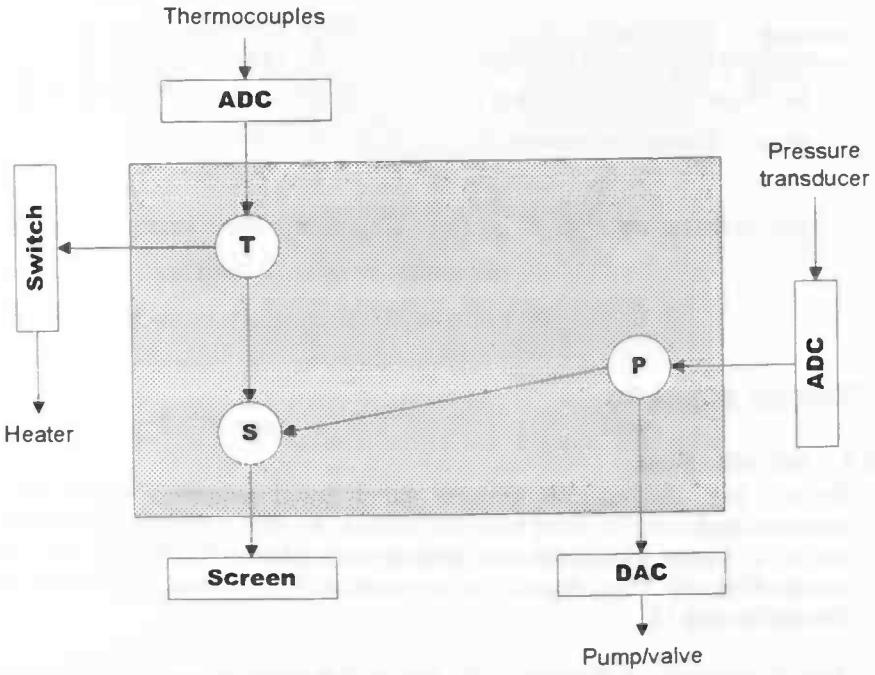


Figure 17: A simple control example [Burns 01]

7.2 The tasks in detail

This section describes the function and behavior of the different tasks, as described in [Burns 01]. In the descriptions, C-like pseudo code will be used to specify the tasks' behavior.

7.2.1 Task T – input/output

The behavior of task T is as follows: task T takes readings from a set of thermocouples and makes appropriate changes to a heater. As opposed to its analogous counterpart, the digitized version of

the control system takes its readings at short (regular) intervals, converts the reading to a setting and sends its output to the switch:

```
temp = read_temperature();          (* store reading in 'temp' *)
convert(temp, setting);             (* convert reading to setting *)
set_switch(setting);                (* output setting *)
output(temp, setting);              (* output data *)
```

Example 2: Task T pseudo code

7.2.2 Task P – input/output

Task P's behavior is similar to that of task T. The code is therefore analogous:

```
press = read_pressure();            (* store reading in 'press' *)
convert(press, setting);            (* convert reading to setting *)
set_dac(setting);                   (* output setting *)
output(press, setting);             (* output data *)
```

Example 3: Task P pseudo code

7.2.3 Task S – output

Task S is, with respect to T and P, a 'passive' task: it waits for input from either T or P to arrive and upon arrival displays the result on some output device. Because task S can receive input from either task T or P, some kind of 'protection mechanism' is needed to prevent the task's input from being garbled. This protection should preferably be enforced or encouraged by the chosen method. If task S is supposed to do some processing on the data streams, before relaying the data to the screen, the code becomes:

```
temp = get_temp();                  (* retrieve data *)
settingT = get_settingT();          (* retrieve data *)
press = get_press();                (* retrieve data *)
settingP = get_settingP();          (* retrieve data *)

data = format_data(temp,            (* process *)
                   settingT,
                   press,
                   settingP);
set_screen(data);                   (* output data *)
```

Example 4: Task S pseudo code

7.3 Change scenario

7.3.1 Introduction

To see how the methods perform (in a broad meaning) when the example system is extended/modified (i.e. extra tasks are added), the different implementations will be extended as shown in Figure 18. In the new system, two tasks are added: a new input task Y, and an intermediate task X that depends on the results of two different tasks, while serving as a 'provider' for another task (S).

Task Y represents a threshold light sensor that could be used, for example, for measuring the transparency of a liquid. The difference between this task and task P and T is that the input sensor for Y is aperiodic because of the thresholding performed. This means that Y does not 'poll' the device for data like P and T, but receives an event at irregular intervals.
I chose to extend the example with such a task, to see how a task of different nature (aperiodic) is handled by the different methods.

Task X, on the other hand, is an intermediate task: it does not communicate directly with devices outside the system, but rather depends on readings from task P and Y. Without going too much into implementation details, there are globally two ways in which this task can be implemented: by polling the results of task P and Y regularly, or by waiting until either P or Y (or both) indicates that it has data ready.

7.3.2 Realization and considerations

In an ideal situation, adding tasks to a system requires minimal or no changes or (negative) consequences to the existing system. The quality of a system is strongly influenced by the chosen architecture, and it is especially true in the case of system modifiability. Of course, software quality is not *only* affected by the architecture of a system: a bad implementation breaks a good architecture.
For the system we are looking at here, there are two general solutions of extending the system: the first being leaving the first implementation 'intact', and trying to glue the new tasks on, or 'integrating' the new tasks with the others.
The former of these is less intrusive for the existing tasks, but may not deliver the preferred result, while the latter is intrusive with respect to the existing system, but may result in a better implementation.

In a simple system like this, where tasks are written from scratch, the choice for either of the methods may be of minor impact. In real systems, on the other hand, the need to change the existing implementation may prove to be infeasible. Nonetheless, the implementation of the 'fabric' in which the tasks operate (the method of providing concurrency, in the figures the grayed rectangle) may prove to be of importance for the degree of modifiability of the system. (This means that the design pattern is made to an architectural choice).

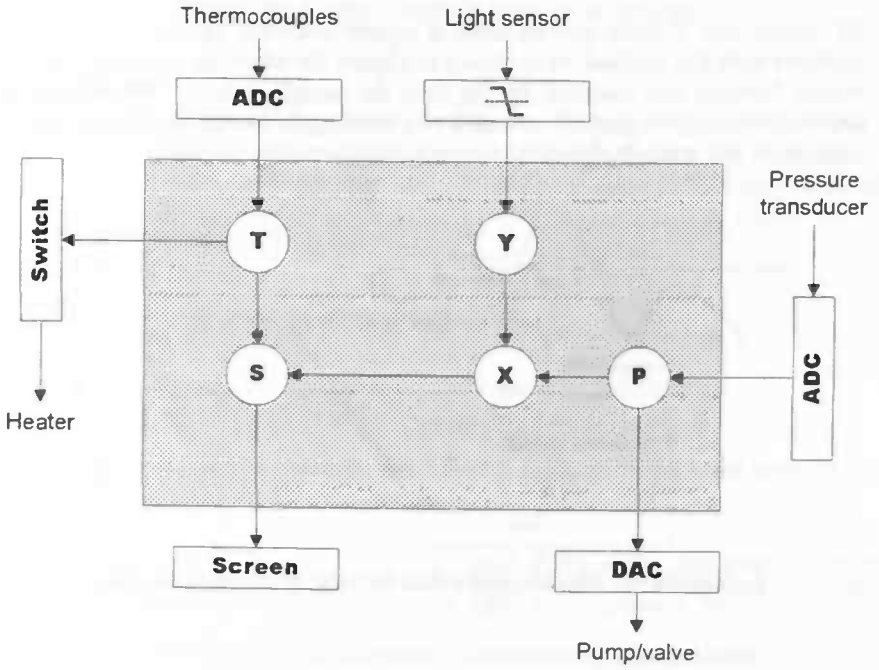


Figure 18: Example with two added tasks

7.4 The extended system in detail

Figure 18 shows the example with the two added tasks X and Y. Tasks P, T and S are not changed.

7.4.1 Task X – intermediate

As described in the change scenario, task X is an 'intermediate' (or internal) task. It does not directly communicate with entities outside the system, but rather does so through other tasks. The tasks surrounding X could therefore be regarded as 'buffers' holding data. The way in which task X accesses these data can be done either by regularly polling the buffers, or by, which might better map to the nature of task Y, waiting for task P and/or Y to indicate to X that new data is available. Supposing task X does some processing on the data, the pseudo code would be:

```
press = read_int_pressure();          (* store reading in 'press' *)
signal = read_threshold();            (* store reading in 'signal' *)
```

```
convert(press, signal, setting);      (* convert readings to setting *)
output (press, signal, setting);      (* output *)
```

Example 5: Task X pseudo code

7.4.2 Task Y – aperiodic input

Task Y is of a different nature than tasks P and T, which need a more or less continuous reading from the analog/digital-converters (ADCs), convert it to some setting to control other devices.
Task Y, on the other hand, is connected to a sensor that only gives a signal when some value (in this case, the amount of light passing through some detector) is reached or exceeded. This makes task Y both aperiodic and sporadic with respect to the other tasks in the system [Burns 01].
For example, the behavior of task Y could be as follows: if the sensor detects that the value drops below some limit, another –different– signal is given. The system should be respond upon and only upon, occurrence of one of these events by executing task Y. In this example, task Y should retain the signal that was received, until Y is triggered again, upon which it will change the stored signal to reflect the received signal. The pseudo code for task Y would be:

```
signal = read_signal();               (* store reading in 'signal' *)
present(signal);                      (* make available for other tasks *)
```

Example 6: Task Y pseudo code

Of course, task Y could poll its input at regular intervals, as task P and T do. However, there is a problem with this method, as is shown in Figure 19: when the sampling rate is too low, and a peak occurs *between* two samples, by the time the sample is taken, the readout will be invalid. This problem can only be partially solved by increasing the sampling rate, but that may not be desirable (especially not when peaks occur very sparsely), or even possible.

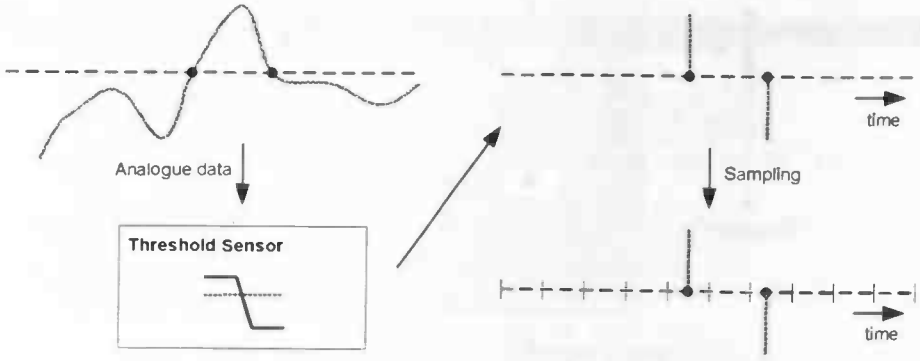


Figure 19: Missing input data because of the sampling rate being too low.

7.5 The implementation

7.5.1 Introduction

Symbian OS provides all described methods of concurrency (active objects, processes, threads, and, of course, cyclic scheduling), and is therefore the target platform for implementing the prototypes. While supplying a full-fledged graphical user interface, the focus on implementing these prototypes is on concurrency, and therefore a simple console-based interface will be used that provides a minimal level of control.
For the implementation, the 7.0b2 beta version System Developer's Kit (SDK) from Symbian was used. All implementations run on the Symbian OS emulator that is supplied with the SDK. As C++ is the native language of Symbian OS, this will be the implementation language for the prototypes as well.

All implementations of this example use a common set of classes that provide access to and control over the console.

7.6 The Cyclic Scheduler

7.6.1 Introduction

Constructing applications that use a cyclic scheduler is easy. Controlling behavior and changes however, is not. If all tasks have the same priority and are periodic, a simple loop that executes the tasks in order will do. More on the specifics of cyclic schedulers can be found in 3.6 and e.g. [Burns 01].

Both the original and extended implementation have a class structure as shown in Figure 20.

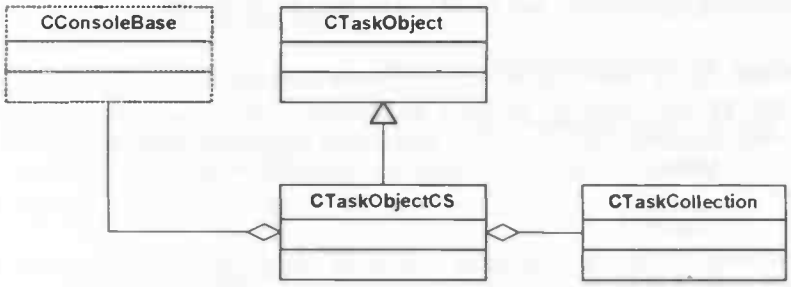


Figure 20: Cyclic scheduler naive class structure

7.6.2 Implementation I

Tasks P, T, and S have no specified priorities, so therefore they can be assumed to have the same relative priority. For the cyclic scheduler in this example (which uses the most naive form of cyclic scheduling), this means a simple loop with just one invocation of each task. In pseudo code:

```
while (ShouldIBeRunning) {
    taskP();                              (* call task P *)
    taskT();                              (* call task T *)
    taskS();                              (* call task S *)
    ShouldIBeRunning = checkIfIShouldBeRunning();
}
```

Example 7: Cyclic scheduler - naive implementation

To gain a finer-grained level of execution, the different subparts of the tasks could be interleaved (see section 7.2):

```
...
press = read_pressure();              (* task P-1: store reading in 'press' *)
temp = read_temperature();            (* task T-1: store reading in 'temp' *)
convert(press, setting);              (* task P-2: convert *)
... etc.
```

Example 8: Cyclic scheduler – interleaved implementation

This, however, poses new problems:

Dependency – tasks that depend on each other for data. In the example system: when should task S be put in, as it depends on the output of tasks P and T for its data? Here, the solution is simple: it could be put after task P and T, but in systems that are more complex the solution might not be as easy.

Maintainability and extensibility – when extra tasks are added, maintaining correctness of a cyclic scheduler can become hard. Especially when tasks are complex, splitting them may prove to be impossible.
Furthermore, when the definition of one of the tasks is changed, the implementation of the cyclic scheduler needs to be changed, which is generally not preferable and is bad practice from a software architectural point of view.

A solution to these two problems could be mixing both the interleaving of and the invocation of 'complete' tasks, or provide methods to 'step' through the different tasks, but this does not solve the fundamental problem of cyclic schedulers.

7.6.3 Implementation II

The simplest way of extending the cyclic scheduler is by inserting calls to the new tasks before task S, which is because of the dependency of task S on the output of task X. However, that approach fails to correctly represent the behavior of task Y.
Unfortunately, the naïve version of the cyclic scheduler does not offer methods to overcome this problem: the solution has to be introduced by methods that are external to the scheduler[14] (that is, the programmer has to provide them).
Having said this, I will adhere to extending the system as described above, because the choice for using a cyclic scheduler is mostly driven by specific needs, its limitations make it unfit for more demanding applications, and better alternatives are available.

In short, the extended example becomes:

```
while (ShouldIBeRunning) {
    taskP();                          (* call task P *)
    taskT();                          (* call task T *)

    taskY();                          (* call task Y *)
    taskX();                          (* call task X *)

    taskS();                          (* call task S *)
    ShouldIBeRunning = checkIfIShouldBeRunning();
}
```

Example 9: Cyclic scheduler - naïve implementation

Table 8 shows the output from both the first (naïve) and the extended example. The values that are shown are random numbers between 0 and 100.

Original system	Extended system
Inside Task P Reading Pressure: 97 Converting: 97 -> 25 Setting switch: 25 Output from task P: 97, 25	Inside Task P Reading Pressure: 69 Converting: 69 -> 20 Setting DAC value: 20 Output from task P: 69, 20
Inside task T Reading Temperature: 42 Converting: 42 -> 69 Setting switch: 69 Output from task T: 42, 69	Inside task T Reading Temperature: 88 Converting: 88 -> 2 Setting switch: 2 Output from task T: 88, 2
	Inside Task Y Reading signal: 31 (threshold: 0) Output from task Y: 0
	Inside task X Converting: 69 -> 94 Converting: 0 -> 23 Output from task X: 69, 94, 0, 23
Inside Task S set_screen: 42, 69, 97, 25	Inside Task S set_screen: 88, 2, 94, 20

Table 8: Cyclic Scheduling example output

7.6.4 Discussion

In the previous chapter, strong points of a cyclic scheduler were mentioned. These points concerned memory efficiency, resource reuse, verifiability of application behavior, fairness of scheduling, and synchronization. Of these, really only on the fairness of scheduling and verifiability of application behavior, something can be said: scheduling is, of course fair, because each task runs as often as its peers do. Application behavior can, because of the non-preemptiveness, be verified, but only if the scheduling requirements are not too complex. As no

[14] This judgment is, though, not completely fair: section 3.6 shows an extended version of a cyclic scheduler that supports run-time addition of tasks and task priorities.

memory usage measurements have been taken, little can be said on the memory overhead of the method, but one can expect this to be negligible.

The weaker points are easier to point out. Of these, the lack of effective event handling has been discussed together with the inflexibility of a cyclic scheduling approach.
In short, the most important drawbacks that were observed in this example were:

- Inflexibility of scheduling,
- Lack of event-handling,
- Dependency problems,
- Maintainability and flexibility issues

Not shown in this chapter, but nonetheless important, is a problem when using cyclic schedulers in the OO-paradigm: extending a cyclic scheduling class in its pure form by subclassing and extending it, is impossible because of the hard-coded calls to the different tasks. Using a list-based approach as described in section 3.6 could overcome this problem because task calls and the scheduler are uncoupled.

Although the cyclic scheduling approach in its pure form has a number of drawbacks, it can be improved quite easily to overcome a number of these drawbacks, as was explained in section 3.6. The problem of interleaving subtasks might be mitigated by providing a 'step function' for each task that handles the sequential execution of the subtasks, which makes it easier to add new tasks, but this does not overcome the inelegance of the solution. Stepped functions might be used to implement so-called long-running tasks.

The concept of a cyclic scheduler may be straightforward, writing and maintaining one, is not. For simple applications where overhead in terms of computational and memory usage must be kept to an absolute minimum, a cyclic scheduling approach could be considered. However, when these are not a consideration, or the limitations are not that strict, a different approach might be the way to go.

7.7 Symbian's Active Object Framework

7.7.1 Introduction

As said, the active object framework is an event-handling framework. Normally, active objects issue a request to a service provider and perform some action when the request completes (in the case of our example system, the expiry of a timer). After this, the active object is put in a state of rest (is marked 'inactive'), until a new request will be issued.
In this prototype, however, we want to simulate a continuous system. The way to do this with active objects is by letting the objects re-issuing their requests when they have finished processing (see section 4.4.4). As an alternative, the system could be built by modeling the tasks as active objects that act as service providers as well. In this way, task T would request a service (e.g. 'write to screen') from task S and so on. As mentioned, we want to simulate a continuous system here, so the long-running task approach is used here, a prototype that uses the multiple service provider approach is described in chapter 8.
Because the active object framework works within a thread or process, the active objects it uses can communicate through sharing memory. This is also the way the tasks 'communicate' in the prototype.

7.7.2 Implementation I

Figure 21 shows the class relation diagram for the implementation of the control system with active objects: each task instance is encapsulated by an instance of `CRunner`. `CRunner` is derived from the framework's base class `CActive`, which defines a basic active object. When a `CRunner` instance is created, the object adds itself to the active scheduler that is present in the application, but does not yet issue a request; the tasks remain inactive. When the application starts, three (but not limited to three) instances are created, and for each instance it is indicated which task it should represent.

When the user indicates that the system can be started (which is done by pressing a key), all task objects are called to issue their requests.
Upon completion of a request, each instance calls its corresponding body function (which is in the class `CTaskCollection`). The body function is called and runs in its entirety to completion, after which each instance re-issues its request. This brings the granularity of concurrency to a per-task level. To gain a finer level of granularity, each sub-part of the task could be encapsulated in a separate `CRunner` (see the remark about 'step function' in section 7.6.4).
The execution of the different `CRunner` instances is controlled through the `CTaskObjectAO` class, which is also responsible for creating the instances and cleaning up.

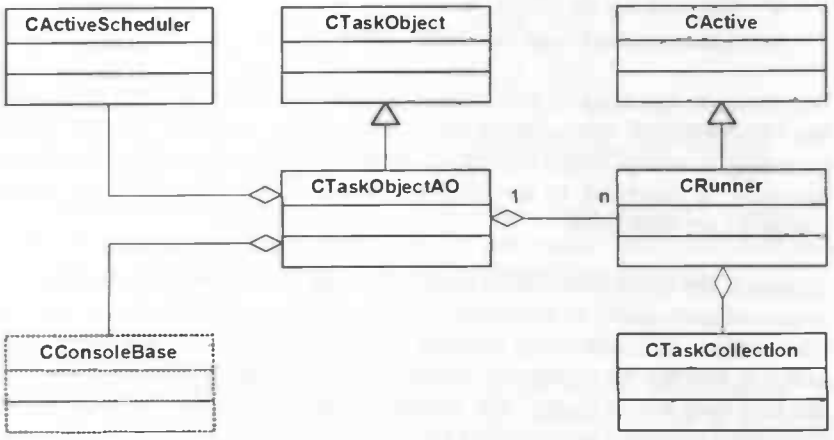


Figure 21: Active Objects – original system class structure

7.7.3 Implementation II

A system of active objects can generally be extended with new active objects, without too great impact on existing tasks, provided the added objects are not too intrusive[15]. In the extended implementation of our system, a new subclass of `CActive`, `CSporadic` is introduced that is to represent task Y (see Figure 22). The biggest difference between `CSporadic` and `CRunner` is that the former does not re-issue its request when complete.
To simulate the 'random' nature of task Y, that task is not started together with the other tasks (which would not make sense), but responds to a key being pressed ('k'). If a key press is detected by the operating system, task Y issues a request for expiry of a time, but for a very short period (say, 0ms, which corresponds to immediate time-out).

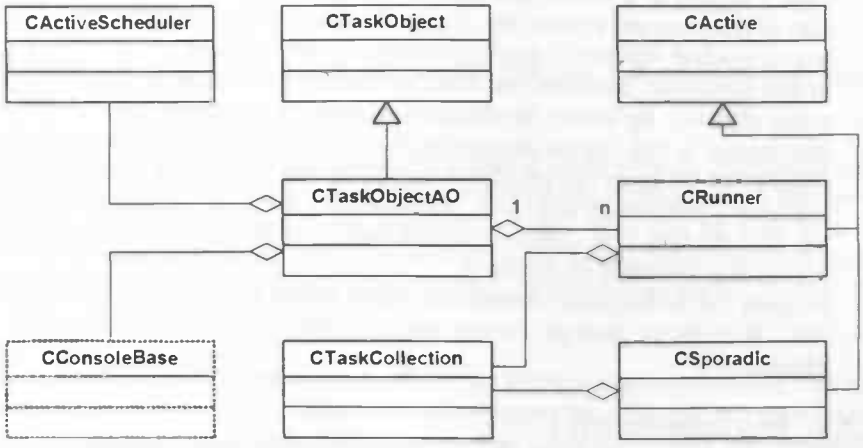


Figure 22: Active Objects – extended system class structure

[15] E.g. short interval between re-issuing requests or too long body functions – this degrades application performance

7.7.4 Results

Both the original and the extended versions of the control system were run a number of times to investigate how well they were able to model a continuous system. Table 9, left column shows the output for the original system where the time-out period for each task was 100ms. The order in which the tasks were started was as is shown (T, P, and S). Each object had standard (active object) priority, because a higher or lower priority would not make sense (each task is weighted equally).

The right column of Table 9 shows two example runs of the extended system. The upper half of the right column shows the sporadic task Y being issues ('CSporadic issuing request') and the handling of completion of the request ('Inside Task Y'). The reason why task T is handled before Y is that its request had probably timed out at around probably the same time the request for Y was issued.

The lower half shows also the extended system, but with task Y having standard priority just as the other tasks. Two observations can be made: 1. the duration between the issuing and handling of the request for task Y increased dramatically and, 2. the order in which the other tasks are executed is disturbed (the tasks were started and run in the order T, P, S, and X). The area marked in grey shows task S and X which have been swapped.

Original system	Extended system	
Inside Task T Inside Task P Inside Task S	Inside Task T Inside Task P Inside Task S Inside Task X	EPriorityHigh
Inside Task T Inside Task P Inside Task S	Inside Task T Inside Task P Inside Task S Inside Task X	EPriorityHigh
Inside Task T Inside Task P Inside Task S	CSporadic issuing request	EPriorityHigh
Inside Task T Inside Task P Inside Task S	Inside Task T Inside Task Y Inside Task P Inside Task S Inside Task T	EPriorityHigh
Inside Task T Inside Task P Inside Task S	Inside Task T	EPriorityStandard
	CSporadic issuing request	EPriorityStandard
	Inside Task P Inside Task S Inside Task T Inside Task P Inside Task X Inside Task S Inside Task Y	EPriorityStandard

Table 9: Active Object example output (edited for size and readability).
Time-out period: 100 milliseconds

So far, the system behaved as predicted. When, however, the time-out period for tasks was decreased (original system, to 1ms for each task), the system started to behave unpredictably, as can be seen in Table 10, left column: the tasks run as expected for a while (T, P, and S in order), but after a while, the order changes and task S does no longer get executed.

A problem of the same kind arises when a number of requests for execution of task Y are issued. As an active object can have only one request outstanding at any time [Tasker 00], previous requests are canceled before a new request is issued. Table 10, right column shows this situation. What can be seen from the table is that the number of *requests* is far higher than the actual number of times the completed request is handled.

Time-out period too small	Requests get canceled
Inside Task T Inside Task P Inside Task S ... Inside Task P ◄ Inside Task T Inside Task S Inside Task T Inside Task P Inside Task T Inside Task P	CSporadic issuing request Inside Task S Inside Task Y Inside Task T CSporadic issuing request CSporadic issuing request CSporadic issuing request Inside Task T CSporadic issuing request CSporadic issuing request Inside Task T CSporadic issuing request Inside Task P Inside Task Y CSporadic issuing request CSporadic issuing request CSporadic issuing request Inside Task T

Table 10: Active Objects: two problems with high frequency events.
Left: time-out period (1ms) approaches – or gets under – body execution time,
Right: requests are canceled before they are handled.

7.7.5 Performance

Although active objects support (per-object) priorities, the actual 'performance' (frequency of occurrence) in this implementation is determined by the time-out period of the request for each active object, which each object sets when issuing the first request[16].

In short, there are two factors that are responsible for the performance and responsiveness of a system that uses the active object framework in a way as described here:

Time-out period – this is the most obvious factor that influences performance: the shorter the period, the higher the performance. However, when the period gets too short, depending on the number of tasks, requests will complete before other completed requests have been handled, which can result in unwanted behavior (e.g. starvation of tasks).

Granularity of concurrency – because active objects run non-preemptable, the developer must make sure that objects do not run for too long, because that negatively influences application responsiveness. The solution to this is to split long 'bodies' into either multiple active objects (when subtasks are more or less independent), or use the long-running task mechanism.

Apart from these, there are the more obvious factors that play a role, like system load and per-object priorities.

When the extended system test program is run a number of times with the priority for task Y set to high, the number of tasks that are executed between issuing the request for that task and its actual body execution seems to be around one, occasionally two. This can be seen as a significant improvement over the standard setting (`EPriorityStandard`. See Table 9, lower right listing), where seven requests were handled before the `CSporadic` event was handled.

7.7.6 Discussion

The active object framework supplies a way of writing applications in an event-driven way, which differs from the 'normal' function-oriented way of programming. This section uses the framework to simulate the effect of long-running tasks, such as one could get using a (fair-scheduled) multi-threaded or multi-process approach.
If so much can be concluded from the test runs done, the approach seems to work, but only when the number of simulated task switches (completion of requests) does not get too high. In that case,

[16] Actually, the time-out period can be changed at every request, but here one –fixed– interval is used.

the system might start to display behavior as observed in section 7.7.4. This behavior is largely inherent to the framework, because of its non-preemptiveness.
A possible solution for the problem that occurs when the time-out period approaches or falls under body execution time could be to split the active object that represents the task in multiple subtasks, which are all represented by active objects again. However, this would needlessly increase the complexity of the application.
Another solution would be to rewrite the active object framework to support active objects that support multiple outstanding requests, but this would break the current active object paradigm as defined by Symbian.

With respect to the analysis as was made in the previous chapter, one can conclude that the active object framework is indeed an effective event-handling framework. Flexibility of the method is provided through the combination of the active scheduler and active objects, which can be added to the scheduler and scheduled at run-time. Because the framework works non-preemptively, verifying the behavior of a single object is easy. Extending the system is relatively easy: create an active object, add to active scheduler and if necessary, issue a request – and is generally non-intrusive for already existing tasks, provided that the existing tasks do not have too tight timing requirements.
The drawback the method has is that when performance requirements get too high, application behavior might become unpredictable. The reason for this is in the nature and origin of the framework – it was designed for efficiency, not performance. Because of the drawback, real-timeliness cannot be said to be a general property of the framework, although certain small and dedicated implementations might reach real-time behavior. The performance problems as observed here might be seen as a hint for how well the system scales up, although no exact information on scalability can be drawn from these simple prototypes.

7.8 Threads

7.8.1 Introduction

Implementing a system with threads or processes is the traditional way of creating a 'concurrent application'. Most systems nowadays support preemptive task switching, which means that tasks can be written almost just like their 'sequential' counterparts: long running tasks do not have to be split in subtasks, which might not be an option. On the other hand, using existing code unmodified or without inspecting for possible problems with respect to concurrency is generally not advised (for reasons set out in chapter 3).
In Symbian OS (and in most operating systems supporting them), threads are structures that are maintained by the operating system. Operations on the threads are performed directly, or indirectly, by calls to the operating system.
In the implementations of the example system that use threads, each task is represented by a separate thread. Taking the granularity of concurrency to the task-level, and not the sub-task level, seems a logical choice: the target system supports preemptive multitasking, which makes it unnecessary to do the interleaving 'by hand' – it is done by the operating system's scheduler. Furthermore, implementing each sub-task as a separate thread would increase the amount of switching between threads and memory overhead, which is in this case unnecessary and unwanted.

The 'functionality' of each thread is put in a function body that is executed while the thread is running. The function body is implemented as a continuous loop, for when the body completes, the operating system terminates[17] the thread.
This behavior of threads differs from the active object approach, where the active scheduler is responsible for implementing the 'continuous' behavior. Figure 23 shows this difference graphically.

[17] This is the case with Symbian OS. Other systems might behave differently.

Figure 23: Body functions with threads, simple, and long-running active objects

7.8.2 Implementation I

The implementation of this prototype has a class structure similar to the two previous ones: there is an extra class to encapsulate the unit of execution (`RThread`), but no separate class to encapsulate the scheduler (as is the case with the AO-framework). All threads are created at application startup, upon which (on the used platform) they are put in a suspended state.

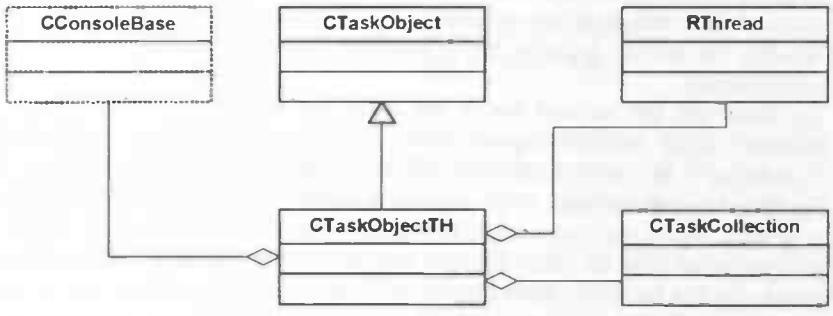


Figure 24: Threads – class structure

To perform an operation on a thread, applications first need to open a handle to the thread (which means finding the correct reference, based on some property, e.g. a name or number). Operations like starting, stopping, changing priority or other parameters, can then be performed by using this handle.
In the implementation of the original system, all threads have the same, standard priority, which represents the equal importance of each of the tasks. In a real system, the screen output might be of lower importance than other tasks, which can be expressed by lowering the priority of the output task. In general, it is good practice to always give a thread standard priority, unless there are reasons not to do so (e.g. background tasks can have a lower priority, while threads that interface with hardware generally have high or real-time[18] priority).

7.8.3 Implementation II

To extend the existing system with a thread for task X is trivial: a new thread is created that represents task X, while existing tasks remain unmodified. The only change required for adding new tasks is the initialization (and control) part of the application, which needs to be able to control the new thread.

Extending the prototype by using inheritance is relatively easy: a subclass of `CTaskCollectionTH` only has to add new code for controlling the threads. This can be done without changing the existing code. Virtually all I/O-operations are in the class `CTaskCollection`, which makes this an appropriate place for synchronization constructs. As this class is only used as a peer, and not subclassed, no inheritance anomalies [Bergmans 94] should occur.

There are two ways in which the thread that represents a sporadic aperiodic task can be used: the application could create the thread at startup (together with the other threads), and only when the services of task Y are needed, it is run. Alternatively, the thread can be created and run at the moment task Y is needed (and destroyed afterwards). The choice for either can have impact on the performance of the system. A few considerations:

[18] Real-time priority is a term that is often used for the highest available priority.

Memory usage – in systems that have very little memory available, allocating memory for a thread that is suspended most of the time (Y is a sporadic task), might not be an option. However, successfully creating a thread at application startup ensures that the memory for its structures is available, something that might not be the case with creating threads 'at run-time'.

Responsiveness – there is always some computational overhead in creating, initializing and destroying the structures that represent a thread. Therefore, in cases where a thread's services are needed as fast as possible, dynamically creating a thread might not be an option. In general, for short sporadic high-priority tasks, the overhead might not be in balance with the actual amount of work the task has to perform.

Such considerations generally apply to using dynamic structures, but are not limited to the above; in the case of threads, these should be taken into account, as well as the nature of the application, and predictability and frequency of occurrence of aperiodic tasks.
In this way, when there is some knowledge of the behavior of the system's tasks, applications could be optimized by carefully selecting the points where threads (or other dynamic structures) are created, initialized and destroyed.

In the extended prototype, task Y is created at application startup, and is only run when needed, to minimize the overhead that is involved with creating the thread. Upon creating task Y, it is given the highest possible priority to ensure that, when needed, it is handled as fast as possible. When its services are needed, the task is activated ('resumed'). Since task Y should only perform a short, single cycle of computation, it is suspended again when it has completed running.

7.8.4 Discussion

Because the target operating system can preempt threads, no assumptions can and should be made on the order in which the different tasks are executed. When simulating such a system, as in these paragraphs, where all tasks run logically in parallel, this should not matter. When explicit dependencies between tasks should be needed, synchronization mechanisms can offer a solution, or dependent tasks could be modeled as subtasks of the tasks they depend on. This is, however, the situation with active objects as well: no assumptions should be made on the order in which active objects run as well. However, because threads can be preempted, the code has to be checked and possibly modified to implement so-called critical sections (synchronization construct – portions of code which, when they are being executed, may not be preempted) to ensure correct behavior. Mostly, critical sections are used where I/O-operations are involved.

Table 11 shows example output of the two systems running.

Original system	Extended system	Extended System + task Y
Inside Task T	Inside Task T	Inside Task T
Inside Task P	Inside Task P	Inside Task P
Inside Task S	Inside Task S	Inside Task S
Inside Task S	Inside Task S	Inside Task S
Inside Task T	Inside Task T	Inside Task T
Inside Task P	Inside Task P	Inside Task S
Inside Task S	Inside Task S	Inside Task X
Inside Task S	Inside Task X	Inside Task P
Inside Task T	Inside Task S	KickIn();
Inside Task P	Inside Task T	Inside Task T
Inside Task S	Inside Task S	--Running task Y--
Inside Task S	Inside Task X	Inside Task S
Inside Task T	Inside Task P	Inside Task Y
Inside Task P	Inside Task S	Inside Task P
Inside Task S	Inside Task S	--Finished task Y--
Inside Task S	Inside Task X	Inside Task S
Inside Task T	Inside Task T	Inside Task X
Inside Task S	Inside Task P	KickIn();
Inside Task P	Inside Task S	Inside Task S
Inside Task S	Inside Task S	--Running task Y--
	Inside Task X	Inside Task T
		Inside Task Y
		Inside Task P
		Inside Task X
		Inside Task S
		--Finished task Y--
		Inside Task S

Table 11: Threads example output (left: original system, middle: extended system without task Y, right: extended system with occurrences of task Y) – edited for size readability

One thing that is remarkable when looking at the output of the different systems is that task S occurs more often than both task T and P. This also shows when the simulations are run a number of times, and the occurrences of the different tasks are counted (see Table 12).

Original system		Extended system	Without task Y	With Task Y
% Task T	22,26%	% Task T	17,35%	17,22%
% Task P	22,27%	% Task P	17,39%	17,21%
% Task S	55,47%	% Task S	43,58%	42,80%
		% Task X	21,67%	21,09%
		% Task Y	-	1,67%
Total number of samples	1909	Total number of samples	2127	2156

Table 12: Occurrences of the different tasks (data collected in three sample runs)

In the original system, task S is executed about two and a half times more often than the other tasks. This higher frequency of occurrence is likely to be a result of the less complex body function of task S, which causes it to complete quickly, upon which it is ready for a next round of execution.
Although the total percentage of calls decreases when new tasks are added, task S still is executed about two and a half times as often as task T and P, as in the original system. This behavior is caused by the 'fairness' scheduler, and can be influenced by changing the priority of e.g. task S. When, for example, task S does some screen updates, getting it executed might be a waste of processing power, and giving task S a lower priority might free up some CPU time to be used elsewhere.

With respect to the classification as was made in chapter 6, no real surprises show up: scheduling is handled preemptively and fair, resources can be reused (shared memory), and events can be handled as expected. No real conclusions can be drawn with respect to the performance over efficiency, although both seem to be in balance. While writing an early version of the thread prototypes, synchronization problems occurred while trying to output information to the screen.

7.9 Summary

The previous sections have shown all methods to be able to provide the basic constructs that are needed to implement a simple system that handles periodic and aperiodic tasks. However, the methods differ on several aspects, like CPU and memory overhead, ease of implementation, maintainability and extensibility (breaking current implementations and inheritance issues) and other required techniques (e.g. synchronization mechanisms).

For an embedded system such as the one described here, the choice for either of the methods depends on other requirements and especially constraints: when the target system does not support threads or any run-time support system, those are not an option. When the choice is between the active object framework and a cyclic scheduling approach, choosing for either depends on the requested performance of the system: when a system needs to perform the same actions, always, the cyclic scheduling approach is a good choice. This is probably why a large number of embedded systems use the cyclic scheduling approach.
When, on the other hand, more flexibility is needed, the active object framework might be an option. Especially from the software architectural point of view, the active object framework has advantages over the cyclic scheduler: the software can be modeled in active objects, which might more closely resemble the software architecture.

Concerning the performance and results of the different implementations, it is important to note that these should not be considered reliable, because other implementations might change the

results. Nonetheless, they should give an insight in how the methods perform globally in a similar system. In addition, no conclusions can be drawn concerning either distribution or fault tolerance, as both were not real issues for the target control system.

Summarizing the contents of this chapter:

- The cyclic scheduling approach can be a fast, easy, no-overhead implementation for a number of systems. However, the solution is nonflexible and they are tedious from a software engineering point of view.
- The active object framework is an event-oriented solution that can be fast under certain conditions, programming a system with active objects is easy and little overhead is involved. The solution proved to be a flexible, verifiable, and clean way of structuring a number of concurrent tasks, but when performance is an absolute requirement, a different solution might be necessary.
- Threads (and processes as well) are well-studied true concurrency constructs that are fast, well defined and flexible. Threads are generally preemptively scheduled by the operating system, so no special scheduling code has to be written by the programmer. However, preemption can make software verifiability hard, and may require special synchronization constructs.
- Not tested in this chapter, but worth mentioning here: only processes bring true protection, but have, when compared to the other methods here, a lot of overhead. When protection is a key requirement, processes are the way to go.

A careful conclusion from these prototypes would be that, depending on the specific needs of the problem, using some complexity in favor of creating a system that is more flexible, maintainable and possibly has a higher performance. If this is not a consideration, the choice for either of the method depends on the other requirements for the specific problem.

8 Prototyping II - A telecommunications protocol

8.1 Introduction

In the previous chapter, a simple control system was discussed. This chapter looks at a different area of a concurrent application: (a simulation of) a telecommunications protocol. The most common way to simulate a collection of independent communicating entities such as a telecommunications protocol, is to model each entity as a separate thread or process which communicates with its peers by methods of inter-process communication that are provided by the run-time support system.
This method has been used extensively and its advantages and disadvantages are well known in both theory and an in practice. As there are a large number of implementations that have shown this method to work, this chapter will focus on a different, not so well known way of implementing a telecommunications protocol.

This chapter discusses how to implement (a subset of) a formal specification to a protocol simulation using an event-handling framework. As in the previous chapter, this will be Symbian OS' active object framework. The method itself will be explained by the use of an example, although the prototype will implement only a subset thereof.

The reason for implementing a protocol with an event-handling framework is that, when analyzing the domain of telecommunication protocols, it seems that protocols are largely event-based.

The formal specification is given in SDL, the Specification and Description Language (ITU-T recommendation Z.100), which describes systems hierarchically in the form of extended state machines, the theory of which is well understood. SDL supports both a graphical and a textual notation for specifying systems. In this example, the former of these notations will be used. Appendix 10 gives a short overview of the meaning of different symbols in SDL.

I will first describe the example protocol, followed by a description of how an SDL specification maps to event-handling objects. Following this, an implementation will complete the description of the workings of the mapping. Finally, an evaluation of the method is made in section 8.6.

8.2 The protocol

The protocol used in this example is the Alternating Bit Protocol [Telelogic], which echoes back all messages as an acknowledgment that they have been received. The description of this protocol will be done top-down with the help of SDL specification diagrams.

8.2.1 The System diagram

The highest level of abstraction of the SDL diagram hierarchy is the so-called *system diagram*. In the example protocol, the system has two channels that send and/or receive signals that are sent to and from the block `OSI_Stack` from and to the system's environment, see Figure 25.

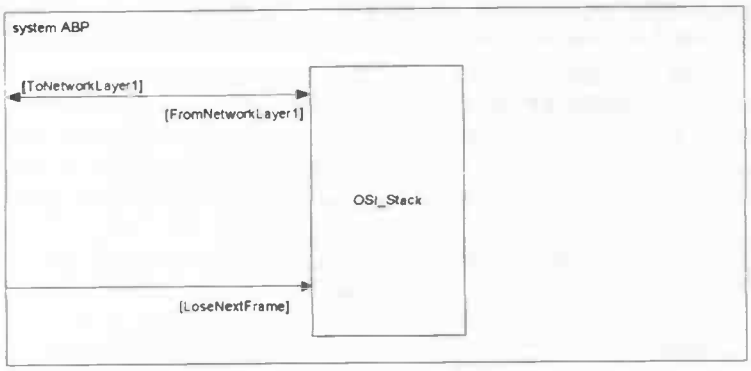


Figure 25: Alternating Bit Protocol - SDL *System* specification

8.2.2 The Block diagram

One step down the specification hierarchy is the block level diagram. Each block is consists of processes or sub-blocks that communicate through channels. In the example system, there are four processes: `PhysicalLayer`, `DataLayer1`, `DataLayer2`, and `NetworkLayer2`. Figure 26 shows the block `OSI_Stack`.

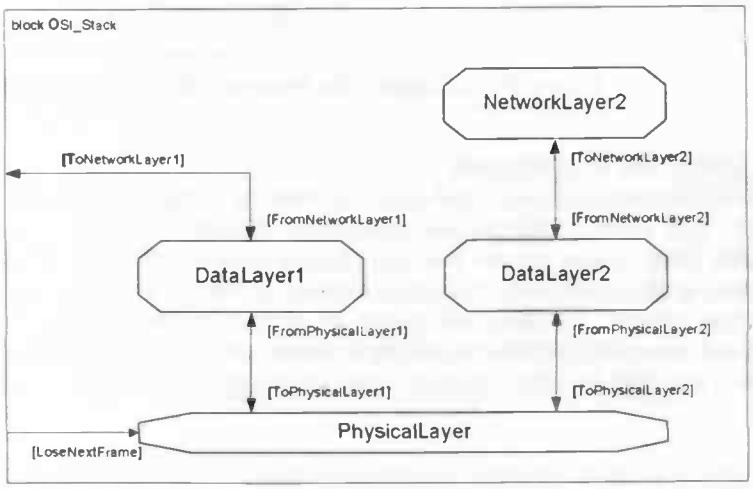


Figure 26: Alternating Bit Protocol - SDL *Block* specification

In the following section, I will only describe one of the processes (`PhysicalLayer`) to explain how the SDL specifies processes. The other process diagrams can be found in e.g. [Telelogic].

8.2.3 The Process diagram

As said, SDL processes are modeled as extended state machines. Figure 27 shows the SDL diagram for the `PhysicalLayer` process of the alternating bit protocol. In SDL, each process has a signal queue associated with it, to which signals are delivered, and where they are held until the process is ready for processing them. Signals coming from the process are directly transmitted through the process' channel(s).

Processes, upon starting, perform an optional initialization, after which they are put in an initial state (in this case `Idle`). While in the particular state, the process waits until there are signals in its queue. When there are any, the process works by removing a signal from the queue and then checking if there is a transition[19] corresponding to that signal. If there is such a transition, the actions corresponding to that transition are carried out, and the process is put in a new state [Edwards 01]. If there is no corresponding transition, the signal is discarded and the process' state is left unchanged. During the transition, a process can update its internal variables, transmit

[19] 'Transition' is a term from automata theory, which, in SDL, corresponds to an input/action/new state-sequence in a specification.

signals, and perform different actions depending on the process' internal variables. In the rest of this section, I will use the term *branch* for the actions that are performed during a state transition.

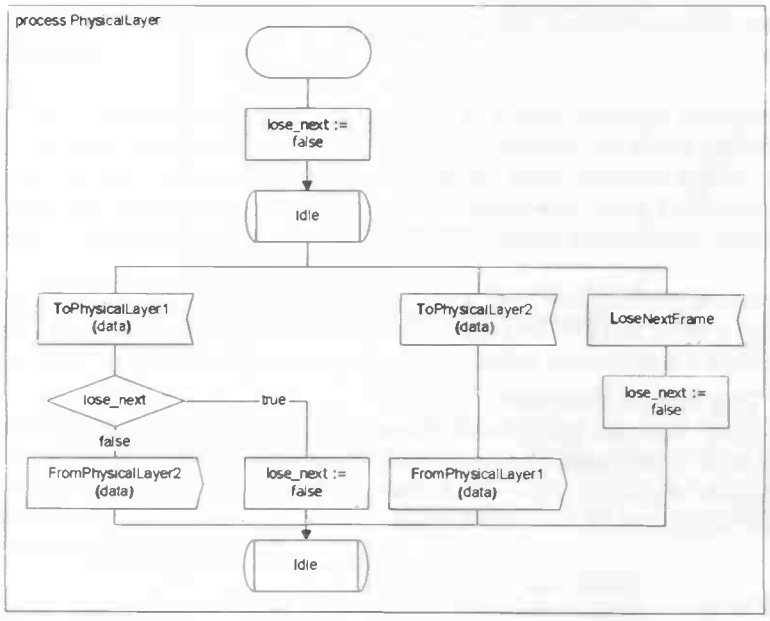


Figure 27: Alternating Bit Protocol - SDL *Process* specification

8.2.4 Channels and signals

In SDL, communication is performed by sending messages (*'signals'*) to and from processes, blocks and systems. Signals are transported through so-called *channels*, which are assumed reliable (that is, they do not lose any signals transported along them) [Bozga 00]. In an SDL (system or block) diagram, channels are drawn as (uni- or bidirectional) directed lines between the different entities, including the system or block's border. Furthermore, each channel has the signals it transports specified as the signal name between square brackets.
Signals can both be 'raw', or carry data, which can be of built-in or user-defined type [Edwards 01].

8.3 Mapping an SDL specification to Active Objects

This section describes the concept of mapping the different aspects of a protocol specification to the active object framework.

8.3.1 Processes

The life cycle of each SDL process follows a general pattern of initialization, entering a certain state and waiting for signals. Upon receiving a signal, that signal is removed from the queue, the correct state transition (if any) together with its corresponding actions are performed, and the process enters a new state.

There is an observation that can be made here: when the process is in a certain state, it 'waits' for a signal to arrive. In the active object framework, this has an analogy in requesting a signal, and the request being serviced. With this observation in mind, there are two ways in which such this request-response paradigm can be modeled: one active object for each input-branch, or one active object per SDL process.
The former of these two possibilities can be quickly discarded for the following reason: when a process is modeled as a collection of active objects, it can have multiple requests (one for each type of signal) waiting to be serviced. When one of these requests completes, the corresponding active object will run 'its' branch. This behavior is correct, with the exception that not all signals should be accepted in all states. In other words, state information would have to be communicated

to all active objects upon state change, or outstanding requests that are invalid in certain states would have to be canceled.
While this might work, it is clearly not a clean solution. Another problem is that the SDL describes that when a process is in a certain state, and a message arrives for which there is no transition, the message should be discarded. When modeling each branch as an active object, such a distinction is hard or impossible to make, and mechanisms that can keep the queue's state intact (such as re-queuing or 'peeking' at the queue) would have to be implemented.

The alternative is to model the whole process as one active object, the workings of which are depicted in Figure 28.

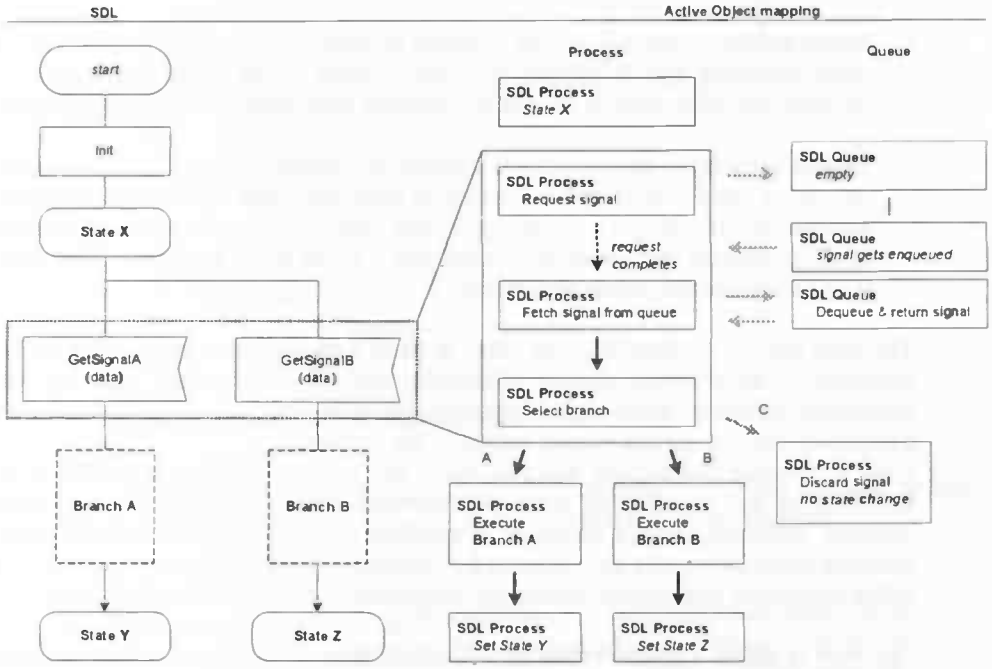


Figure 28: Modeling an SDL process as an active object

This method uses one active object to model all inputs of signals that are in the process. When the process enters a certain state, requests a signal from the queue and waits until the request is serviced. If a signal is available in the queue, the request completes and the process' run function is called. Because the whole process is modeled as just one active object, the run function knows the current state, and the signal can be checked to see if it was legal in the current state. If the signal is found legal, the corresponding branch can be executed. At the end of the branch, a new state is entered and the whole process can be repeated.

There is a third way in which an SDL process can be modeled with active objects: implementing each process as a long-running active object. As this is in essence simulating a thread, and it follows the general pattern of threads (with a few simplifications here and there), I will not discuss this method here further.

8.3.2 Queues

In the previous section, it was shown that SDL processes can be modeled as active objects that request signals from the signal queue. Thus, queues are services providers. Mapping the queue to use the active object framework involves looking at the requirements for a queue: it needs to accept signals from channels and deliver them to the process. When there are no signals in the queue, the process waits, when the queue is full, the channel will have to wait until the process removes a signal. There is no 'waiting' involved for the queue, which therefore needs not be modeled as an active object: it just needs to notify either the process or the queue when either performs an action on the queue.

8.3.3 Communication and Channels

Communication in an SDL system is performed by sending signals over channels. Channels are assumed to deliver signals within a block instantaneously. Between different blocks, channels are assumed to have delays. There are roughly three ways in which a communication channel can be realized [SDL-SIE5065]:

Procedure calls – sending a signal is implemented as a (synchronous) procedure call in the peer process. Because procedure calls are normally synchronous, the whole system 'blocks' for the duration of the call. This method can implement the event behavior that is implicit in the system, but does not handle continuously changing values very well. Furthermore, modeling delays is harder with procedure calls that are expected to finish as quickly as possible.

Signal buffers – channels can be modeled as signal buffers, which allows for modeling delays while still being able to capture the event-oriented nature of sending signals. A signal buffer can store and delay signals, as well as notifying other entities that a signal is available.

Shared memory – modeling each channel as a shared memory location which all processes can access has the benefit of being easy to implement, but has the disadvantage that processes have no way of telling if a message has been 'sent'. Only active entities that check the memory location regularly can see if the channel has a signal being sent on it. This makes this method not suitable in event-handling systems.

The third method of these does not seem to fit in with the event-oriented set-up of the system as described in the previous sections. Therefore, this leaves procedure calls and signal buffers as options when looking at an event-handling implementation: both methods are capable of acting as a transport channel that can trigger events on the 'other side'.
From a software engineering point of view, the latter of the two is preferable because of the modeling of the channels as separate structures. The procedure call implementation defines channels implicitly, which could be a problem when the channel specification changes: the involved processes would then have to be changed instead of the channel. This leaves the signal buffer alternative as the most interesting, and probably most suitable of the three.

One way to model a channel with active objects, is to use a signal buffer together with an active object that requests a queue at the receiving end to store the message. When a process wants to send a signal, it calls the channel synchronously to transmit the signal. The channel stores the signal, and requests the process at the other end to accept the signal. After the request is made, the sending process can continue processing. If the receiving process has space in its queue, the request completes immediately, which results in the channel to deliver the signal to the process' queue. If the receiving process was waiting for a signal, it is notified that a signal has arrived, and the process will start processing as described in the previous sections.
If, however, the receiving process' queue does not have free space to store the signal, the channel will wait for the process to drain its queue. If the process does so, the channel is notified, and the signal is delivered. Here again, there is the pattern of requesting a service (delivering the signal) and servicing the request (accepting the signal). Therefore, channels can be modeled as active objects. Because active objects can have one and only one request outstanding at any time, modeling a bidirectional channel requires two active objects: one for each direction.

8.3.4 Blocks and the SDL system

As SDL blocks are logical abstractions of collections of communicating processes and sub-blocks, modeling is trivial. Blocks should allow channels to be 'attached' to communicate with the block's environment (the system or a parent block). Such a socket-like approach can be achieved by supplying blocks with the same channel-queue mechanism as is used for processes to communicate. Blocks themselves need not to be modeled as active objects.

In the previous sections, I described how an SDL specification could be mapped to an event-handling framework. The following sections describe the implementation of a simple protocol using the mapping as proposed in these sections.

8.4 The implementation

8.4.1 Class structure

In the previous sections, a number of relations between different entities were mentioned. Putting these relations into a class diagram results in (for example) Figure 29.

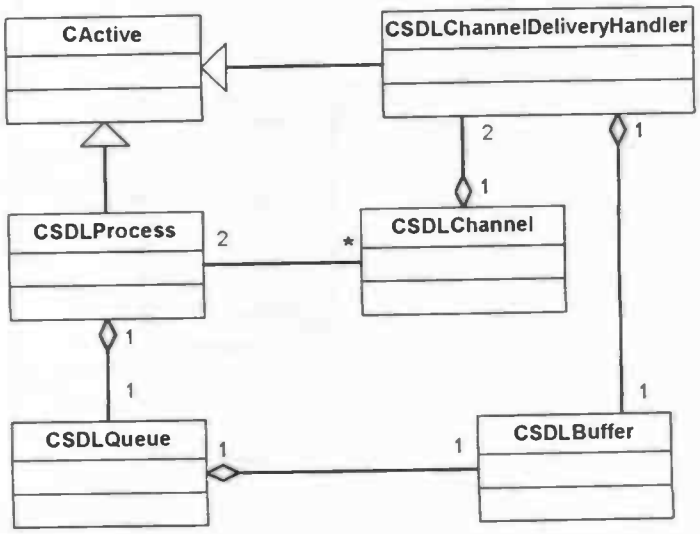


Figure 29: SDL-Active Object implementation class hierarchy diagram

In the diagram, the class definition for blocks and the system have not been included, as they have a class relation similar to `CSDLProcess`, without being active objects themselves.

8.4.2 Processes

When implementing a system, developers would have to subclass `CSDLProcess`, which defines a number of basic functions, to implement that process behavior. When a subclassed process is created, the base constructor takes care of creating and associating a queue with the process. As processes are active objects, their run functions (`RunL`) contain the process implementation.

8.4.3 Queues

Queues act as service providers, but need not be active objects themselves. Both processes and channels depend on the services provided by a queue and that is delivering or accepting signals. In fact, a queue, as proposed here, is a simple wrapper for a buffer structure, which can signal if there are changes to the buffer.

8.4.4 Channels

A channel consists of four objects: the channel class, a buffer, and two 'delivery handlers'. When a process wishes to send a signal, the signal is stored in the channel's buffer structure. The handlers are active objects that request receiving queues to accept signals that are in the channel's buffer. When the receiving party is not ready to accept, the handler waits until it can do so. While the handler is waiting, processes still can use the channel to send signals – they are stored in the channel's buffer.

Currently, channels are only assumed to be connected to processes. However, adapting the implementation to allow channels to be attached to channels (e.g. to simulate block or system boundaries, or to implement joining/splitting of channels) involves only minor changes to the constructors of channels and channel handlers to incorporate the new 'types' – the concept, and therefore the working of the channel does not change.

8.4.5 Sending signals

Processes send signals by offering them to all their attached channels. The channels are responsible for discarding signals that would travel in the wrong direction or over the wrong channel. Figure 30 shows the sequence of messages for sending a signal between two processes

using the method as describe above. In the chart, objects marked with a heavy border are active objects.

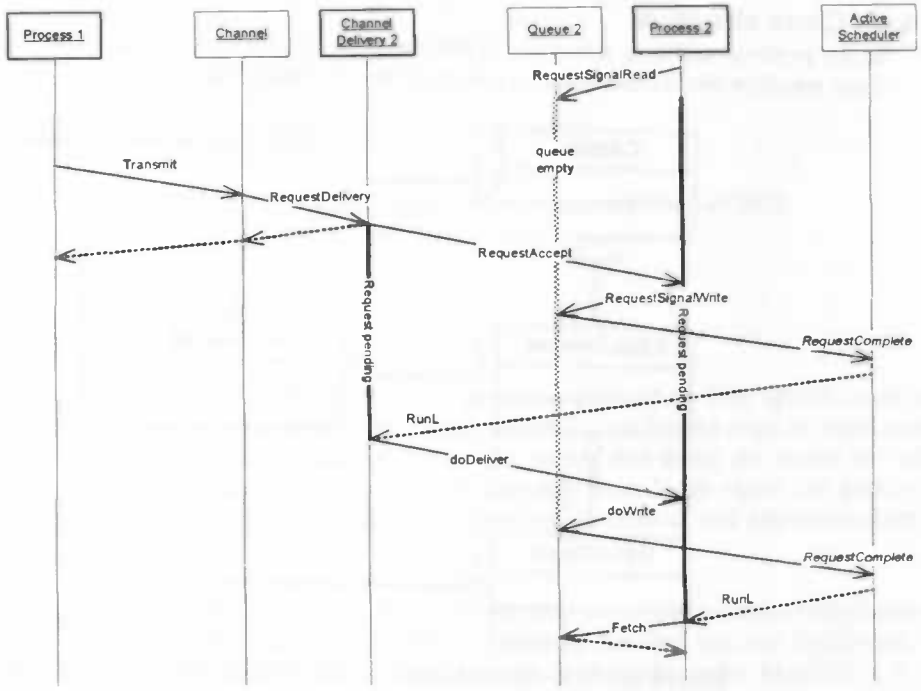


Figure 30: SDL-AO – Sending a signal between two processes.

Implementing a system using the method described in the previous sections is straightforward: creating the processes, creating channels and attaching them to the processes, starting the processes and, finally, starting the active scheduler.

8.5 Code example – Processes

It follows too far to include all source code of the method here, and therefore, I will include only a relevant part to show how the method can be implemented.
As said, the 'behavior' of an SDL-process is implemented in its (active object) `RunL`-function. The code below shows how to implement the state machine of process `PhysicalLayer`, as described in section 8.2.3. The code shows how signals that are invalid in certain states (in the example there only one state), are discarded.

```
void CSDLProcessPhysicalLayer::RunL() {

   // Request signal from queue and make sure it is not NULL
   Signal thisSignal = GetSignal();
   __ASSERT_ALWAYS(thisSignal != NULL, User::Panic(_L("Signal is NULL"), 1));

   // Produce debug output
   _LIT(KRunLDebugMsg, "\n[p%d]CSDLProcessPhysicalLayer:: RunL(signal: %d)");
   RDebug::Print(KRunLDebugMsg, ASDLProcessID, thisSignal->signalID);

   // Message literals
   _LIT(KPINGDebugMsg,
          "[p%d]CSDLProcessPhysicalLayer:: Received PING (%d), sending PONG");

   // Create new signal
   SignalBase *newSignal = new SignalBase;

   // Implement the behavior
   switch(ACurrState) {
          case IDLE: {
                 switch(thisSignal->signalID) {
                        case SIG_ToPhysicalLayer1: {
                               if (lose_next == true) {
                                      lose_next = false;
```

```
                                } else {
                                        // Create & send signal
                                        newSignal->signalID =
                                                SIG_FromPhysicalLayer2;
                                        SendSignal(newSignal);
                                }
                                break;
                        }
                        case SIG_ToPhysicalLayer2: {
                                newSignal->signalID = SIG_FromPhysicalLayer1;
                                SendSignal(newSignal);
                                break;
                        }
                        case SIG_LoseNextFrame: {
                                lose_next = true;
                                break;
                        }
                        case SIG_PING: {
                                RDebug::Print(KPINGDebugMsg, ASDLProcessID,
                                                ++ASignalCounter);
                                newSignal->signalID = SIG_PONG;
                                SendSignal(newSignal);
                                SetState(IDLE);
                                break;
                        }
                        default: {
                                // Incorrect signal received - ignore
                                SetState(IDLE);
                                break;
                        }
                } // switch signalID
                break;
        } // case IDLE
    } // switch State

} /* RunL */
```

Example 10: Implementing the PhysicalLayer process

The example includes signals SIG_PING and SIG_PONG, which are not in the specification of the process, but were put there for testing purposes. This code is an adaptation of the Alternating Bit Protocol simulation with two processes that send each other signals (PING and PONG). Listed below is example output that is generated by (among others) the code above.

The example output shows the function calls, much as depicted in Figure 30, a process (PhysicalLayer) is handling a completed request (RunL(signal: 8000)), upon which it sends a new signal (PONG) to its channel. The channel is called to transmit the signal, and requests the channel delivery handler at the other 'end' of the channel to deliver the channel to the process there. The process' queue is requested to allow writing to the queue, which is allowed, because the queue is not full. After this, the channel delivery handler is signaled upon which it delivers the signal to the queue. The other process' request, which was waiting for a signal to arrive in its queue, completes and the signal is removed from the queue and the process executes its RunL-function. Here, it all starts again (but then from the other process' point of view).

```
[p1000]CSDLProcessPhysicalLayer:: RunL(signal: 8000)
[p1000]CSDLProcessPhysicalLayer:: Received PING (1), sending PONG
[p1000]CSDLProcess:: Sending signal 9000 to 1 channel(s)
[c2000]CSDLChannel:: Transmit() - sending signal 9000 left->right (1000 to 1200)
[>1200]CSDLChannelDeliveryHandler:: RequestDelivery(signal: 9000), waiting...
[p1200]CSDLProcess:: RequestAccept()
[p1200]CSDLQueue:: RequestSignalWrite()
[p1200]CSDLQueue:: queue not full, signaling channel ready to write
[p1000]CSDLProcess:: SetState() - State transition 0 -> 0
[p1000]CSDLQueue:: RequestSignalRead()
[p1000]CSDLQueue:: queue empty
[p1000]CSDLProcess:: SetState() - requested signal, waiting
[>1200]CSDLChannelDeliveryHandler:: RunL() - [0]
[p1200]CSDLProcess:: doDeliver(signal: 9000)
[p1200]CSDLQueue:: doWrite(signal: 9000)
[p1200]CSDLQueue:: Fetch()

[p1200]CSDLProcessDataLayer1:: RunL(signal: 9000)
[p1200]CSDLProcessDataLayer1:: Received PONG (1200), sending PING
[p1200]CSDLProcess:: Sending signal 8000 to 1 channel(s)
[c2000]CSDLChannel:: Transmit() - sending signal 8000 right->left (1200 to 1000)
[>1000]CSDLChannelDeliveryHandler:: RequestDelivery(signal: 8000), waiting...
[p1000]CSDLProcess:: RequestAccept()
[p1000]CSDLQueue:: RequestSignalWrite()
```

```
[p1000]CSDLQueue:: queue not full, signaling channel ready to write
[p1200]CSDLProcess:: SetState() - State transition 2 -> 2
[p1200]CSDLQueue:: RequestSignalRead()
[p1200]CSDLQueue:: queue empty
[p1200]CSDLProcess:: SetState() - requested signal, waiting
[>1000]CSDLChannelDeliveryHandler:: RunL() - [0]
[p1000]CSDLProcess:: doDeliver(signal: 8000)
[p1000]CSDLQueue:: doWrite(signal: 8000)
[p1000]CSDLQueue:: Fetch()

[p1000]CSDLProcessPhysicalLayer:: RunL(signal: 8000)
...
```

Example 11: Example output of two processes (PING-PONG)

8.6 Discussion

In this chapter, a method was presented to do an implementation of an SDL specification using a collection of collaborating active objects. The main reason for not having done an implementation with traditional concurrency constructs here is for the reason that doing so, would not add new aspects to the contents of this chapter – there is thorough knowledge about the particulars of threads and processes, as well as experience in protocol simulation and implementation using those constructs.

The active object framework has the benefit of not needing any synchronization mechanisms that are required when using threads or processes. In addition, the fact that the framework works nonpreemptively makes reasoning about, and verification of the code easier. In addition, it has the benefit of being self-contained: it does not need to rely on operating system structures like threads, processes, semaphores and the like, which should make porting code written using the framework relatively easy. Apart from the easy verification of code, typical concurrent problems are avoided (such as race conditions, priority inversions and deadlock). The static (initialization, memory consumption) and dynamic (switching) overhead of the active object framework is negligible compared to threads and processes.

The drawback of non-preemption is that exact timing cannot be guaranteed, which might not be acceptable in certain application areas. As the system scales, the number of events increases, which can lead to a decrease in the system's 'internal' responsiveness. This problem is inherent to non-preemption, and is not a result of the method.

This chapter shows that a protocol specification can be mapped effectively to a collection of active objects, which could be expected because of the event-oriented nature of the problem domain. Although the working example implementations give hints to how more complex implementations using method work, it does not deliver hard facts concerning the actual performance or feasibility of such a system, nor does it say anything about how the active object framework performs in other domains.
To gain insight in how the method performs under different conditions, the method should be tested with systems that are more complex. A more thorough knowledge of the effect and particulars of using the framework could be gained by implementing the same system by using traditional concurrency constructs.

9 Conclusions

This thesis is about using requirements for the evaluation and selection of concurrency constructs. The reason for using concurrency to solve a problem is, that doing so, makes the solution easier, possibly more logical, or simply because there is no way of doing it without concurrency.
The research community has developed a number of different constructs for providing concurrency, which range from a naïve implementation of cyclic scheduler to process schedulers for distributed or real-time systems. Some of these implementations are more general than others, but at least all can logically execute a number of tasks concurrently. The previous chapters have shown a number of different constructs that provide concurrency. Furthermore, they showed that the methods indeed provide concurrency, and what the issues are when using those constructs.
The active object solution, as defined by Symbian, has shown to be able to provide concurrency within an application, but showed to be restricted concerning performance. On the other hand, the active object framework was designed for efficiency, not performance. The cyclic scheduling approach is an easy way of providing logical concurrency without overhead, but its limitations make it fit for only a subset of concurrency problems.

Choosing a solution for a selected problem depends on the requirements and constraints for that problem, which is what this thesis wants to show. In chapter 1, a requirements analysis method was introduced that takes an initial requirements specification or problem statement, analyzes that information and finally assesses that information with a number of concurrency constructs and their properties.
Chapter 2 derived a number of concurrency requirements for a future mobile device by analyzing the problem domain. These requirements were used in chapter 6 to assess a number of different available constructs. The assessment did not deliver *the* solution for the problem posed in chapter 2: as the different constructs showed to have overlap in their characteristics, all would probably suffice for at least a naïve implementation of the product. The choice for either of the methods therefore depends on the other requirements and constraints that are posed on the design of such a device. What the chapter does show, is how, from an initial requirement or problem statement, a number of characteristics can be derived that can be used in helping making the final selection.
Chapter 8 looks at a problem from the same angle, but the results are different. In that chapter, analysis of the problem domain showed that a telecommunications protocol is in essence an event handler. This observation lead to the idea that instead of simulating event behavior with traditional concurrency constructs (threads, processes), trying to solve the problem with an event-handling framework might be a solution more fit to the problem. Moreover, chapter 8 shows the implementation of a protocol with an event-handling framework to be easy and clean. As relatively little is known about active objects when compared to traditional constructs, little can be said on how the solution performs in a real-life high-performance situation. Performance problems, as observed in chapter 7, might be a result of the specific implementation, and a different implementation might overcome these. However, this does not change the basic observation that analyzing the domain of a problem with respect to concurrency constructs might lead to an implementation that is better from an engineering point of view.

In this thesis, I defined a requirements analysis method, which is used in the selection and evaluation of concurrency constructs. The method is applied on a (synthesized) mobile device and a number of concurrency constructs, validation has been done by writing a number of prototypes.
Furthermore, I used a sub-set of the method defined, namely *inherent concurrency*, to define a mapping from a formal specification language to a number of collaborating active objects. This mapping is effective and (simple) prototypes have shown the mapping to work. However, further work is needed gain more insight in how the mapping will perform in more demanding systems.

This thesis does not seek completeness, therefore the time span was too short and the area too broad. Nonetheless, I think this thesis gives an insight in how analysis of requirements can be a useful tool in selecting or evaluating concurrency constructs for a given problem.

9.1 Recommendations and Future work

The area of event-handling framework is relatively unknown when compared to traditional concurrency constructs. Symbian has designed a low-overhead implementation that showed to be able to provide a logically concurrent solution. However, as it was designed with efficiency in mind, not performance, no conclusion should be drawn concerning the performance of the method in a system with tight timing requirements. Here, a different solution might change the odds. These considerations make the active object framework an interesting area of further research.

9.1.1 Requirements analysis method

Concerning the method defined in this thesis, future work could be done on the analysis step, for example by defining additional analysis steps or by splitting current steps in sub-steps. Also, generalizing the method may allow for the assessment of constructs other than concurrency-specific ones, like, for example, a collection of design patterns. In the case of generalization of the method, the analysis step needs to be redefined: other selection criteria need to be used and possibly more steps are needed.

Future work, or experience with the method defined in this thesis, might also lead to a number of requirements that can be considered standard or default for the method, and should therefore always be taken into account. Extending the information-collecting step by using quantified requirements (e.g. performance in operations per second etc.) could prove to be a useful tool in the selection or evaluation step.

Summarizing the possible future work for the requirements analysis method:

- Generalization of the method – other design patterns
- Extending the analysis phase
- Quantified requirements
- Standard requirements

9.1.2 SDL-Active Object Mapping

The SDL-Active Object mapping as I have described in this thesis has, as mentioned, the drawback of the lack of information. So, especially concerning the performance and scalability of a system implemented with the mapping, more research should be done. It must be noted however, that the performance and scalability issues are not a property of the mapping, but are issues that are inherent to the non-preemptive active object framework. An interesting area for further work on the mapping is an extension of the active object framework so that different objects can be scheduled to run in on multiple processors.

Overall, the active object framework, its applications and particulars are an area of research in its own right, and future work on them is encouraged.

Appendix

10 SDL: Specification and Description Language

10.1 Introduction

SDL, the *Specification and Description Language* is a formal language that grew out of the European telecommunications market and is mostly used for formally specifying protocols. Here, a short introduction to SDL is given.
SDL systems consist of three basic components:

System – an SDL *system* consists of a collection of concurrently running blocks, which communicate through so-called channels. Channels are explicitly defined in the system specification. An SDL system specification can be seen as a number of distributed communication entities (computers).

Block – an SDL *block* is an abstraction level lower than the system, and consists of a number of concurrently running SDL *processes*, or one of more blocks. As with the system specification, a block specifies the communication channels that are also explicitly defined in the specification. Each block represents a single processor.

Process – SDL *processes* specify the actual working of the system. Processes are modeled as extended finite state machines. Associated with each process is a single input signal queue to which signals are delivered. Processes remove signals from their queue and react correspondingly.

As said, processes communicate by sending signals that are transmitted through channels. Signals can be seen as messages that correspond to certain events. More information on SDL in [Edwards 01], [IEC-SDL], [ITU], [SDL-SIE5065] and [SDLForum].

10.2 Used symbols

SDL uses a number of different symbols that correspond to different actions or states of the system. Table 13 gives a non-exhaustive overview of SDL symbols and their meanings.

Symbol	Description
	**Start/terminate** Indicates start and end point of the SDL process
	**State**
	**Signal input** Indicates start of branch / 'signal handler'
	**Signal output**
	**Action**
	**Decision**

Table 13: SDL symbols

11 Client-server computing examples

This chapter shows a number of client-server computing scenario's graphically such as can be found in e.g. Symbian OS.

11.1 Client-server computing

The figure below shows basic client-server computing with a client, and a two-level server: an engine and a service provider. In this scenario, all server logic is in remote entities.

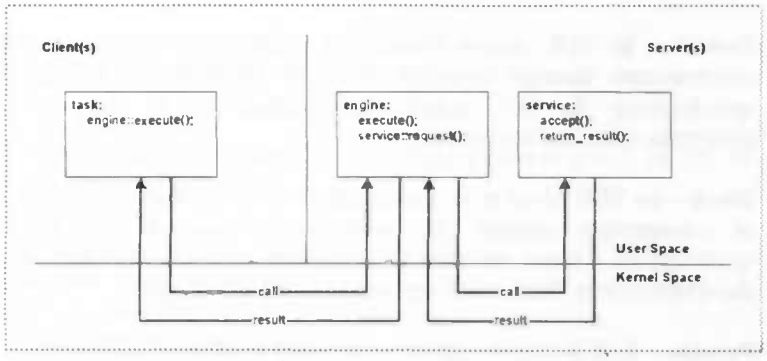


Figure 31: Client-server computing

11.2 Multiple clients with own engines

Figure 32 shows a 'traditional' way of programming: each client (note that this term is not quite true anymore) has its own engine. No calls are made (with respect to that engine) outside the client.

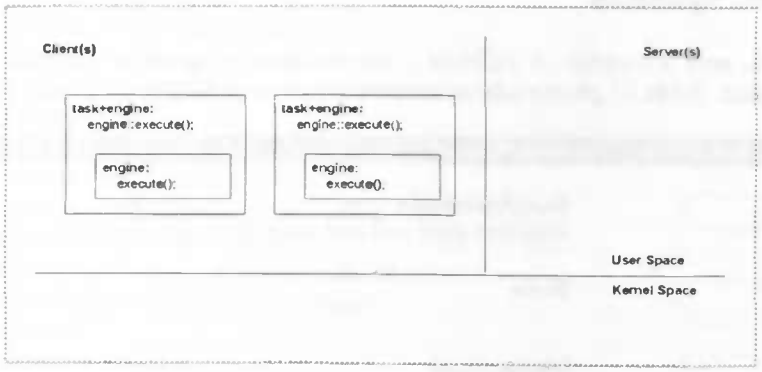


Figure 32: Multiple clients with own engines

11.3 Multiple clients sharing an engine

A common way of building client-server programs can be seen in Figure 33: a number of (different) clients that share a (central) server engine.

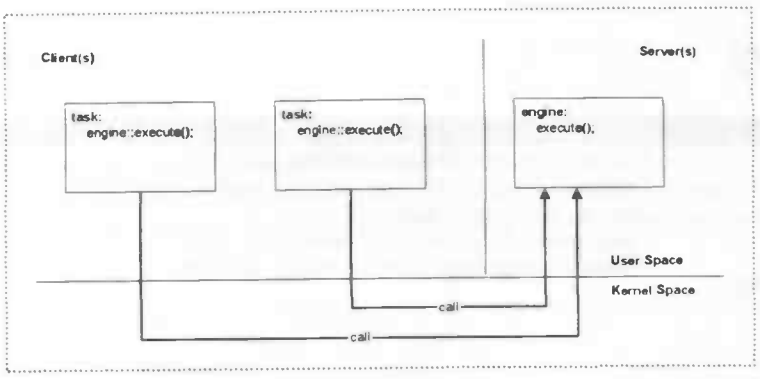


Figure 33: Multiple clients sharing a central engine

11.4 Remote computing

11.4.1 Direct connection and connection fails

Client-server computing can be used to defer (preferably transparently) computation to another device. Devices that use this way of computing should make sure that all critical systems reside on the device, for when the connection to the remote device fails, the systems are inaccessible. Figure 34 and Figure 35 show these situations graphically.

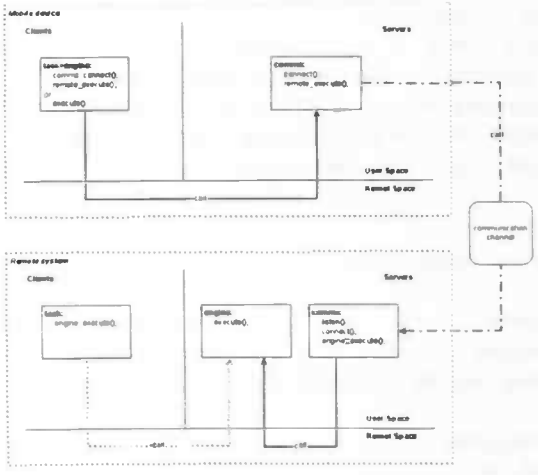


Figure 34: Remote computing, direct connection and connection fails

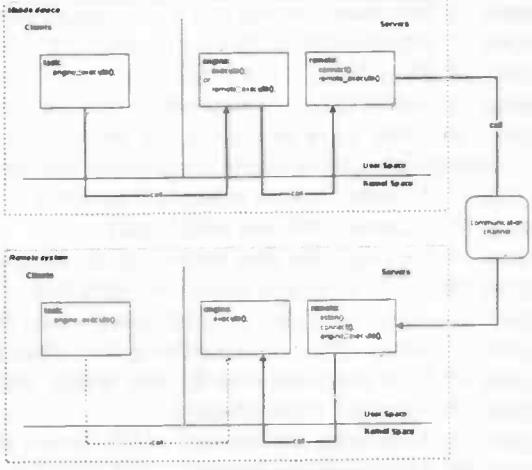


Figure 35: Remote computing with a proxy engine

12 Glossary

Term	Description
Communicator	PDA with communication facilities. See PDA.
PDA	Personal Digital Assistant. A lightweight, hand-held, usually pen-based computer used as a personal organizer.
Smart Phone	Telephone with PDA functionality. See PDA.
WLAN	Wireless Local Area Network (IEEE 802.11*x*)

13 Table of figures

14 Contact Information

Symbian is a software licensing company, owned by wireless industry leaders, that is the trusted supplier of the advanced, open, standard operating system for data-enabled mobile phones. Symbian's mission is to create a mass-market for Symbian OS mobile phones by enabling its licensees to build winning products. Symbian's website is found at www.symbian.com.

Symbian AB

Peter Molin
Symbian AB (UIQ Technology AB)
Soft Center VIII
372 25 Ronneby
Sweden

Email: peter.molin@symbian.com
Telephone: +46 – 457 – 38 64 02

University of Groningen

Jan Bosch
University of Groningen
Department of Computing Science
PO Box 800
9700 AV Groningen
The Netherlands

Email: jan.bosch@cs.rug.nl
Telephone: +31 – 50 – 363 39 41
Secretary: +31 – 50 – 363 39 39

My information:

Breght Boschker
Rembrandt van Rijnstraat 71
9718 PJ Groningen
The Netherlands

Email: b.r.boschker@wing.rug.nl
Telephone: +31 – 50 – 579 74 79
Mobile: +31 – 6 – 533 90 560

15 Literature

[Bergmans 94] Lodewijk M.J. Bergmans – *"Composing Concurrent Objects"*, Ph.D. thesis for the University of Twente, The Netherlands, 1994. ISBN 90-9007359-0,

[Bozga 00] Marius Bozga, Alain Kerbrat, Daniel Vincent *et al.* – *"SDL for Real-Time: What Is Missing?"*

[Burns 93] Alan Burns, Geoff Davies – *"Concurrent Programming"*, Addison Wesley Publishers Ltd. 1993. ISBN 0-201-54417-2

[Burns 01] Alan Burns, Andy Wellings – *"Real-Time Systems and Programming Languages"*, Third Edition, Pearson Education Limited, 2001. ISBN 0-201-72988-1. Online resources at: http://www.cs.york.ac.uk/rts/RTSBookThirdEdition.html

[Chen 00] Guanling Chen and David Kotz – *"A Survey of Context-Aware Mobile Computing Research"*, Dartmouth Computer Science Technical Report TR2000-381

[Coulouris 88] George F. Coulouris, Jean Dollimore – *"Distributed Systems – concepts and design"*, Addison Wesley Publishers Ltd. 1988. ISBN 0-201-18059-6

[Edwards 01] Stephen A. Edwards – *"SDL"*, lecture notes 2001, Columbia University, http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/sdl.ppt

[Golding 95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes – *"Idleness is not sloth"*, Research Report, Hewlett-Packard Laboratories, Palo Alto CA, 1995

[Gomaa 93] Hassan Gomaa – *"Software design methods for concurrent and real-time systems"*, Addison Wesley Publishers Ltd. 1993. ISBN 0-201-52577-1

[Hesselink 01] W.H. Hesselink – *"Concurrent programming"*, Lecture Notes, Computing Science Department of the University of Groningen, 2001

[Hofmeister 99] Christine Hofmeister, Robert Nord, Dilip Soni – *"Applied Software Architecture"*, Addison-Wesley Publishers Ltd. 2000. ISBN 0-201-32571-3

[Hopcroft 01] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman – *"Introduction to Automata Theory, Languages, and Computation"*, Addison Wesley Publishers Ltd. 2001. ISBN 0-201-44124-1

[IEC-SDL] International Engineering Consortium – *"IEC: Specification and Description Language (SDL)"*, http://www.iec.org/online/tutorials/sdl/

[ITU] International Telecommunication Union, ITU-T, Recommendation Z.100 – *"Specification and Description Language (SDL)"*.

[JavaFrame] Øystein Haugen, Birger Moller-Pedersen – *"JavaFrame: Framework for Java Enabled Modeling"*, Ericsson Research NorARC

[Kotonya 97] Gerald Kotonya and Ian Sommerville – *"Requirements engineering: processes and techniques"*, John Wiley & Sons Ltd., 1997. ISBN 0-471-97208-8

[Mery 01] David Mery, Technology Editor, Symbian Ltd. – *"Symbian OS Version 6.x - Detailed operating system overview Revision 1.2, October 2001"*, http://www.symbiandevnet.com/technology/symbos-v6x-det.html

[Mery 02] David Mery, Technology Editor, Symbian Ltd. – *"Symbian OS Version 7 - functional description, revision 1.1, March 2002"*, http://www.symbian.com/technology/symbos-v7x-det.html

[Meyer 93] Bertrand Meyer – *"Systematic Concurrent Object-Oriented Programming"*, Communications of the ACM, September 1993/Vol. 36, No. 9

[Nierstrasz 93] Oscar Nierstrasz – *"Composing Active Objects – The Next 700 Concurrent Object-Oriented Languages"*, Research Directions in Object-Based Concurrency, MIT Press 1993, pp.151-171.

[Nierstrasz 95] Oscar Nierstrasz, Dennis Tsichritzis – *"Object-Oriented Software Composition"*, Prentice Hall International Ltd. 1995. ISBN 0-13-220674-9

[Rakotonirainy 00] Andry Rakotonirainy, Seng Wai Loke, and Geraldine Fitzpatrick – *"Context-Awareness for the Mobile Environment"*, research paper, January 31, 2000.

[SDLForum] The SDL Forum Society, http://www.sdl-forum.org/

[SDL-SIE5065] Rolv Braek, Richard Sanders – *"Software design, SIE 5065"*, lecture notes, Institutt for Telematikk, Norges teknisk-naturvitenskapelige universitet, 2001. http://www.item.ntnu.no/fag/SIE5065/foiler/pdf/

[Sommerville 01] Ian Sommerville – *"Software Engineering"*, Sixth Edition, Pearson Education Ltd., 2001, ISBN 0-201-39815-X

[Spaanenburg 99] Lambert Spaanenburg – *"Real-Time Systems"*, Lecture Notes, Computer Science Department of the University of Groningen, 1999

[Stallings 00] William Stallings – *"Operating Systems"*, Fourth Edition, Prentice Hall, 2000, ISBN 0-13-031999-6

[SymSDK6] Symbian Ltd. – *"Symbian 6.1 SDK"*

[SymSDK7] Symbian Ltd. – *"UIQ SDK for Symbian OS v7.0" (beta edition)*

[Tanenbaum 92] Andrew S. Tanenbaum – *"Modern Operating Systems"*, Prentice Hall International Editions, 1992, ISBN 0-13-595752-4

[Tanenbaum 96] Andrew S. Tanenbaum – *"Computer Networks"*, Prentice-Hall, Inc., 1996, ISBN 0-13-394248-1

[Tasker 00] Martin Tasker, Jonathan Dixon, John Forrest, Mark Heath, Tim Richardson and Mark Shackman – *"Professional Symbian Programming – Mobile solutions on the EPOC Platform"*, Wrox Press Ltd., 2000, ISBN 1-861003-03-X

[Telelogic] *Telelogic Tau SDL and TTCN Suite 4.3*, © 2001, Telelogic. More information on www.telelogic.com.

[Wang 95] Randy Wang, Arvind Krishnamurthy, Jeanna Neefe – *"Analytical Model for Parallel Programs"*, Princeton University Computer Science Department 1995, http://www.cs.princeton.edu/~rywang/berkeley/258/

15.1 List of hyperlinks

ID	Location
[DevWeb]	www.symbian.com/developer
[Actv]	www.symbian.com/developer/techlib/papers/tp_active_objects/active.htm