ISSN 0280-5316
ISRN LUTFD2/TFRT--5739--SE

Hardware Simulation for Testing IEC 61131-3

Johan Holmqvist
Adrian Ulander

Department of Automatic Control
Lund Institute of Technology
March 2005

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* March 2005 *Document Number* ISRN LUTFD2/TFRT--5739--SE
Author(s) Johan Holmqvist and Adrian Ulander	*Supervisor* Karl-Erik Årzén at LTH Peter Thorwid, Alfa Laval in Tumba *Sponsoring organization*

Title and subtitle

Hardware Simulation for Testing IEC 61131-3 (Hårdvarusimulering för testning av IEC 61131-3)

Abstract

Testing of control code written in IEC 61131-3 has long been a strenuous manual process. The focus in this master's thesis is on building a simulator of an Alfa Laval separator to enable testing of control code.
In this particular case, the control code being tested is written and executed in a soft PLC called TwinCAT. The simulator for testing code is written in C#. When testing code, automated tests are preferred because it enables easy regression testing. For this purpose a testing tool using
a customized script language has been developed. Testing with a simulator is also beneficial because errors can be found early in the development process, thus reducing the number of errors left to be found when testing on hardware. Comparison tests show that the simulator performs sufficiently well for testing of control code.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language Sweden	*Number of page* 66	*Recipient's notes*
Security classification		

Contents

1. Introduction

Control of industrial processes is today dominated by computerized systems. Important requirements for such systems are reliability and robustness. For this reason a special type of computer called Programmable Logic Controller (PLC) was developed in the late 1960's. PLCs have since evolved into powerful computers for versatile use.

A PLC is a microprocessor based device that monitors inputs and controls outputs according to a user-created program stored in the microprocessors memory. Testing programs for PLCs, and especially PLCs that will control large complex systems is very strenuous and time consuming. Hassapis (2000, p. 345) claims in his article that testing constitutes more than 50% of the total development cost when constructing PLC programs. A factor that contributes to increasing testing costs is the fact that a PLC program can not be tested fully without access to the system it will control. Freiseisen, Keber, Modetz, Pau & Stelzmueller (2002, p. 195) establish the same fact in their article that software systems including a PLC program can not be tested unless the mechanical environment is available. Mechanical systems are very often manufactured and assembled late in the development phase to lower storage costs which leaves only a short time for software testing before delivery.

It can also be risky to run an untested PLC program on a real system. The program might contain an error that can cause the real system to break or in worst case injure someone.

1.1 Problem identification

Questions that arise are: How can the cost of testing PLC programs be reduced? Software development within other areas uses special tools to increase the quality of the code by testing that the code does what it is supposed to. Can such a tool be applicable when creating a PLC program and will it make it easier for PLC programmers to test their program without access to the real system? The answer to the last question is yes and no. A PLC program is usually written specifically for the system it will control. It is very hard to test if it controls the real system the way it is supposed to without being able to observe how the real system reacts. In other words, the key issue lies in having access to the real process when testing a PLC program.

One solution is to simulate the real process and supply the PLC programmers with a simulator which they can use to test their logic in a safe environment. Similar solutions, (Hassapis 2000) and (Freiseisen et al. 2002), have worked well in the past. Furthermore a simulator can to its advantage be used in a wider context than just for testing code. Li & Winitsky (1999) say in their article that a simulator can be used to train operators, adjust and optimize the control of the actual process.

1.2 Purpose and Goal

The purpose of this master's thesis is threefold:

- To build a simulator of Alfa Laval's separators for testing of control code.
- Provide ideas on how Alfa Laval can develop the simulator further.
- Describe benefits of using a simulator when testing control code.

The primary task of the master's thesis is to construct a simulator that functions as an aid to a developer of control code for separators.

1.3 Demarcations

The focus of this master's thesis is on building a simulator for one type of Alfa Laval's separators. The chosen separator, Clara 80, is a separator which clarifies liquids for the food industry. In addition the study will include a description of how the simulator can be transformed to work with other types of separators at Alfa Laval. Clara 80 was chosen because it was available at the time and it was suitable machine to start with due to its relatively low level of complexity.

1.4 Target Audience

The target readers for this master's thesis are supervisors, employees at Alfa Laval Products and Technology, last year students at technological universities and any reader interested in hardware simulations and testing of PLC logic.

1.5 Confidentiality

The equations that provide the base for the hardware model are confidential and have therefore been excluded from this report. Also measurements and various figures from the modeled separator are left out for the same reason. This detailed information is not important for understanding the essence of this master's thesis.

1.6 About Alfa Laval

Gustaf de Laval was a Swedish inventor responsible for 92 Swedish patents and who founded more than 30 companies. The most successful invention by de Laval was the continuous separator. The separator automated the separation process of milk, making it much easier than before.

Alfa Laval was founded back in the 19^{th} century by Gustaf de Laval and Oscar Lamm. The company was at first named AB Separator but changed to Alfa Laval in the 1960's. Oscar Lamm was responsible for production and sales while Gustaf de Laval was responsible for development of new products.

Over time Alfa Laval introduced products in other fields than separation. Alfa Laval is today a global company with three main markets, heat transfer, fluid handling and separation. Separation can be divided into two subdivisions, decanters used for separating large particles from fluids and disk stack centrifuges for separating fluids with or without small particles. Their list of industry partners include companies active in fields such as food, dairy, manufacturing, power, chemicals and pharmaceuticals.

1.7 Chapter Guide

This brief guide on the chapters is intended to help the readers identify their chapters of interest.

Methodology

An introduction to the methods used by the authors in this master's thesis is presented in Chapter 2 – Methodology. Both practical choices as interview techniques and programming methods are discussed, as well as theoretical standpoints on how information has been evaluated.

Separators

To fully understand this master's thesis, basic knowledge about the separator is needed. A brief introduction to the relevant type of separator is discussed in Chapter 3 – Separators.

Programmable Logic Controller

Programmable Logic Controllers, and especially TwinCAT, are discussed in Chapter 4 – Programmable Logic Controller. The chapter also contains an introduction to IEC 61131-3 in general, and Structured Text in particular.

Modeling and Simulation

Different types of models and simulators together with a discussion on how to validate and verify them is presented in Chapter 5 – Modeling and Simulation.

Simulation Solutions

Various simulation solutions, both existing simulation tools and how to write a simulation tool from scratch using a generic programming language, is discussed in Chapter 6 – Simulation Solutions.

Testing Software

Different testing techniques together with their benefits, limitations and risks are presented in Chapter 7 – Testing Software. Testing of PLC software is given special attention.

Practical Work Progress

A description of the established requirements and the chosen simulation solution is presented in Chapter 8 – Practical Work Progress. The chapter also describes the model and how work progressed when developing the model.

Results and Analysis

The final results, primarily the simulator and related programs, are presented in Chapter 9 – Results and Analysis. The programs are shown together with a detailed description on their structure. The chapter finally contains a discussion on how to move on from this master's thesis focusing at the situation at Alfa Laval.

Summary

Chapter 10 – Summary, describes the work from the beginning to the end as well as conclusions about the results.

1.8 Acknowledgements

The authors would like to thank Peter Thorwid, Roland Isaksson and Hans Moberg at Alfa Laval for their invaluable knowledge about separators and for their inspirational ideas about the use of the simulator. A special thanks goes to Karl-Erik Årzén at the Department of Automatic Control for his help in finding the right path for the thesis.

2. Methodology

To be able to achieve the research goals it is important to have basic understanding and knowledge about methodology. It is, however, important to remember that methodology is just a tool and will not in itself reveal any answers (Holme & Solvang 1997, pp. 11-13). Furthermore it is important to realize that the choice of methodological approach is often based on the traditions within the chosen research field. Traditional ideas are important because they control what is regarded as interesting problems within a research field and in some cases even in what direction solutions are directed (Wallén 1996, p. 21).

Another aspect that should be considered is that the authors' basic assumptions can have great influence on the way information is gathered and processed and what conclusions are drawn from it (Björklund & Paulsson 2003, 64-65). The authors feel it is necessary to give a view of their basic assumptions to make the master's thesis more valuable to the reader. All researched information in this master's thesis was judged from a pragmatical point of view. In other words if the information was not useful to the construction of the simulator it was disregarded, no matter how accurate it was. The focus of the work was concentrated on the construction of a useful simulator.

2.1 General Research Design

The initial work of this master's thesis was done in an analytical manner since only real physical objects on the separator were considered when constructing the simulator. The actions and reactions from these objects were defined in mathematical language which is a fundamental characteristic element in analytical research (Patel & Danielsson 1991, p. 24). When these objects are put together they will form a complex system that no longer is observable from an analytical approach. This system will have interacting causes which can lead to synergistic effects, small causes will be able to make large impacts and there will be a definite course of events. The cause and effect view from the analytical approach was no longer adequate and a systemic approach was better suited (Wallén 1996, p. 30). A systemic approach assumes that a system is more than just the sum of its elements and need to be studied as a whole. Important aspects that should be considered are how the system is marked off from its surroundings, how it interacts with its surroundings, how the internal elements interact and how the system changes over time (Wallén 1996, pp. 30,31).

An actors approach was used in the finishing stage of the master's thesis when evaluating the performance of the simulator. This approach is the complete opposite to the analytical approach and signifies the science of interpretation (Patel & Danielsson 1991, pp. 26,27). At this stage of the work the authors had to take a subjective stand to be able to assess if the simulator met its requirements.

2.2 Practical Research Method

When conducting research there exists two main methods: the *inductive* and *deductive* method. The inductive method starts from collected information and tries to extract general and theoretical conclusions without first anchoring it in accepted theory. In other words patterns that can be expressed in models and theories are searched for in the real world. The deductive method on the other hand starts from the existing theory and tries to draw a conclusion about the real world before trying to verify it with collected data. With that, conclusions can be drawn about separate phenomena based on the theory (Björklund & Paulsson 2003, p. 62). This thesis is not really applicable to these methods due to its constructive nature. According to Wallén (1996, p. 95) construction can be seen as a research area of its own with its own methods.

A basic idea in construction is that there exists no best way to construct an object based on scientific knowledge. A good solution is related to what the purpose and who the user of the product is. When constructing an object there are many different aspects that can be more or less important to consider (Wallén 1996, pp. 98-101):

- What are the basic requirements?
- Should the solution be tailored for a specific task or more general?
- What degree of complexity is required? In many technical contexts intentional simplifications are sought after.
- What are the demands on the product? Should it achieve the best possible performance when it comes to quality or cost?
- What is the system? What does the internal structure look like and what functions should be placed on different levels?
- What are the users values, needs and knowledge and how important are they?
- Did the final product turn out the way it was supposed to?

Construction not only comprises construction of physical object, but also software engineering which is one of the key elements of this master's thesis. The authors chose the construction approach because it has the most resemblance with software development practices.

2.3 Practical Work Description

The initial work was done in an explorative manner to gain knowledge of hardware simulations and available simulation tools. Several articles and books were studied and many simulation tools were tested. Alfa Laval was visited at an early stage to conduct empirical work in order to learn about the machine that would be simulated and what kind of software that controlled its behavior. The final functional details of the simulator were worked out and some experiments were made on the real machine before a decision on the type of solution could be made.

Information about different parts of the separator was now gathered and identified in the control program. A basic model was constructed before the work took a more practical turn as the implementation of the simulator started. Programming was done in an XP-like fashion discussed later on. The work was performed in an iterative manner constantly increasing the complexity of the model while maintaining functionality. Finally the resulting simulator was evaluated. A more detailed description of the practical working process can be found in Chapter 8.

Literature Study

Literature studies help to define what is essential within a research field and how a suitable line of action can be established (Patel & Danielsson 1991, pp. 36,37). It is above all a good way to acquire lots of information in a relatively short time and with scarce economic resources. During the work different information was gathered and sought after in databases and on the Internet. The databases were found at Lund's University Library and typical search strings were ”hardware simulation”, ”process simulation”, ”IEC 61131-3”, ”software testing” and ”real-time simulation”. When reviewing the researched information and especially the information found on the Internet the authors paid attention to the fact that the material could be angled to favor some interests, non-exhaustive or in some cases simply not true (Björklund & Paulsson 2003, pp. 67-69).

Interviews

Interviews were chosen by the authors as an efficient way of gaining knowledge about specific parts of a separator and the involved physics in a separator. This knowledge is unique within Alfa Laval and only few employees have a good command of the different fields. These key employees were interviewed by the authors about their individual fields. Before the interviews could be conducted the interviewers had to review written material concerning the different topics to better understand the nomenclature used within the fields. As a natural consequence of the authors' limited knowledge about separators, the interviews were conducted in an informal way. Patel & Danielsson (1991, pp. 60,61) confirms that informal interviews are effective when the interviewer has limited knowledge about a topic.

When investigating in interview form it is important to consider two aspects. First, the degree of standardization, where one considers how much responsibility should be left to the interviewer concerning the formulation of the questions and the order in which they are asked. Secondly, the degree of structuralization, where one consider to what extent the interviewee is allowed to interpret the questions freely based on his or her former experience (Patel & Danielsson 1991, pp. 60,61). The authors went to the interviews without any written manuscript and the interviewees were allowed to interpret the questions in their own way and elaborate the answers as much they wanted. This was motivated as the goal was to gain as much understanding as possible.

There are primarily two ways to register answers from an interview. The first and probably most commonly used technique is taking notes. When using this technique the interviewer can write more or less extensive notes. Too extensive notes can disrupt the flow of the interview and too sparse notes create a risk of loosing information. According to Patel & Danielsson (1991, p. 69) it is very important that the interviewer elucidates his or her notes immediately after the interview. The other technique is to record the entire interview with a recording device. This technique makes the interview itself go a lot smoother and the risk of loosing information is eliminated. The drawback with recording is that it creates more work for the interviewer who needs to go through the recording and extract the vital information afterwards.

The authors chose to take written notes and then immediately after the interview discuss and evaluate the answers to gain the best possible understanding of the topic. These discussions often led to more questions and new interviews had to be scheduled.

Programming

When programming in a larger project, software engineering methodologies can help maintaining code quality and on-time delivery. There are numerous techniques on the market where one of them is *XP* (Extreme Programming). XP is aimed for use in small agile projects containing no more than 20 developers.

In this master's thesis no particular software engineering methodology has been fully used, but XP functions as a good basis when explaining how the implementation has progressed. Both authors are familiar with the XP methodology and using parts of it in this master's thesis was a natural choice.

Some of the most important *XP Practices* related to this master's thesis are described below together with a comment on how and why or why not they have been used. A complete guide to XP can be found in (Beck 2000) and (Jeffries, Anderson & Hendrickson 2001).

On-site Customer Having the customer in the vicinity comes in handy when developing the program. By reducing the physical distance between customer and developer communication is greatly improved (Jeffries et al. 2001, p. 18). This enables both customers and developers to get questions answered quickly and without the inconvenience of mailing or calling. This was carried out by the authors by scheduling the most productive programming period at Alfa Laval. Preparations and small programming tasks where performed off site.

Test First When writing code in an XP project, automated tests are used to achieve confidence in the code. Before writing a method that could break, a test is written that should test the method (Beck 2000, p. 58). If the test is well written it will fail as long as there is more to implement or if a later implementation breaks the code, otherwise it will succeed. "Test First" was not carried out in this project for several reasons. First, the system was GUI-based which proved hard to test with automated tests. Second, a lot of the code is involved in network based communication which also proved hard to test. Finally, some parts could have been tested with Test First, but was replaced by ordinary debugging together with acceptance testing.

Pair Programming When programming in a pair the code is ultimately kept cleaner. The programmer not writing the code takes care of code proofing by checking the semantics and naming (Jeffries et al. 2001, pp. 88-90). Writing code in a pair is like having a conversation, the programmer instantaneously gets a second opinion which keeps him alert. Pair programming was used extensively throughout the project, only side stepped when performing monotonous implementations. The other programmer was however never more than an arms length away making communication regarding questions and suggestions easy.

Spikes A *spike* is a small implementation testing an idea that might have a use in the final implementation. The purpose of the spike is to drive through the problem and get an idea of how the problem is solved. There is no need to craft a perfect solution, it is enough just to learn how to go about it and to be able to make a rough estimate on how long it will take (Jeffries et al. 2001, pp. 41-44). Spikes were used frequently in the beginning stages of this master's thesis to test different ideas.

Coding Standards and Refactoring Two programmers seldom have the same idea on naming and layout of the code. This leads to problems not only when programming in pairs, but also when *refactoring*. Before work is started, a common coding standard has to be agreed upon by the team (Beck 2000, p. 61). Coding standards were never an issue as both programmers have the same background and experience of pair programming.

Refactoring is the process of rewriting code to make it easier to understand and to facilitate further extensions by removing duplications, obsolete code and misleading names. Refactoring is not the result of a bad implementation, it is a natural consequence of an evolving program.

Management Management in the authors' view combines the XP practices *Planning game*, *Small releases* and *Continuous integration*. Planning involves estimating costs, time and priorities. XP talks about *Stories*, as a description of specific functions or parts of the program from the user's perspective (Jeffries et al. 2001, s. 4). In the planning game the customer creates and discusses the stories together with representatives from the programming team. The stories are then prioritized, time and cost estimated and finally put into different stages of the implementation plan.

By continuously integrating changes, new releases can be shipped often. As each new change must pass all tests the program is always runnable. A new release should not include half finished features as this will only confuse the customer (Beck 2000, p. 56).

In this master's thesis, stories were generalized to a list of requirements where all requirements were discussed in a story-like manner. The stories themselves were never written down, this was compensated with regular discussions with the customer. The program was continuously integrated and after each day a runnable version of the simulator was available for the customer.

2.4 Trustworthiness

When evaluating the trustworthiness of a study validity, reliability and objectivity should be taken into consideration. Validity means to what extent a study really measures what it is supposed to. In the case of construction validity could be translated to how well the constructed object fulfills its basic requirements. Reliability means to what extent a study gives the same result when repeated and objectivity implies to what extent values and opinions affect the study. The goal of a study is of course to try and attain as high validity, reliability and objectivity as possible (Björklund & Paulsson 2003, p. 59).

A common practice to increase both validity and reliability of a study is to use triangulation (Björklund & Paulsson 2003, pp. 76,77). Objectivity can be increased by motivating and clarifying all the choices that are made in a study. This gives the readers freedom to make their own judgments. Another way to increase the objectivity is to show all researched information without errors and bias (Björklund & Paulsson 2003, pp. 61,62).

This master's thesis has as mentioned above used three different methods to acquire information. When it was possible information was gathered from different sources, however in many cases information was isolated to one source and only accessible with the use of one method. The constructive nature of the master's thesis has forced the authors to make many choices. These choices, when not supported by theory, have been motivated by the authors.

3. Separators

Separation is the technique of separating fluids from each other or for removing solids in a fluid. The idea is based on the fact that solids in a fluid are drawn to the bottom due to its higher density. The speed at which the particles sink to the bottom can be increased by rotating the fluid which replaces the gravity with a centrifugal force of thousands of G's. To further increase the separation speed several layers, called disks, are inserted to decrease the distance the particles must sink (Alfa Laval 1994).

The result is a rotating tank containing the disks. This is the most important part of the separator known as the bowl. A cross-section of a separator bowl is shown in Figure 3.1. The disks inside the bowl are angled upwards to form a conical shape and make the particles slip outwards more easily. These particles gather at the periphery of the bowl and must be extracted in some way. The extraction is achieved with different techniques depending on the purpose of the separator. For this report the *PX separator* is the most relevant. A PX separator has ports in the wall of the bowl, which are uncovered for a very short period at a certain interval. When the ports are uncovered, the centrifugal force ejects the particles through the ports. This is called a *discharge* (Alfa Laval n.d.). The PX separator can be used when the process liquid contains 1-10% solids.

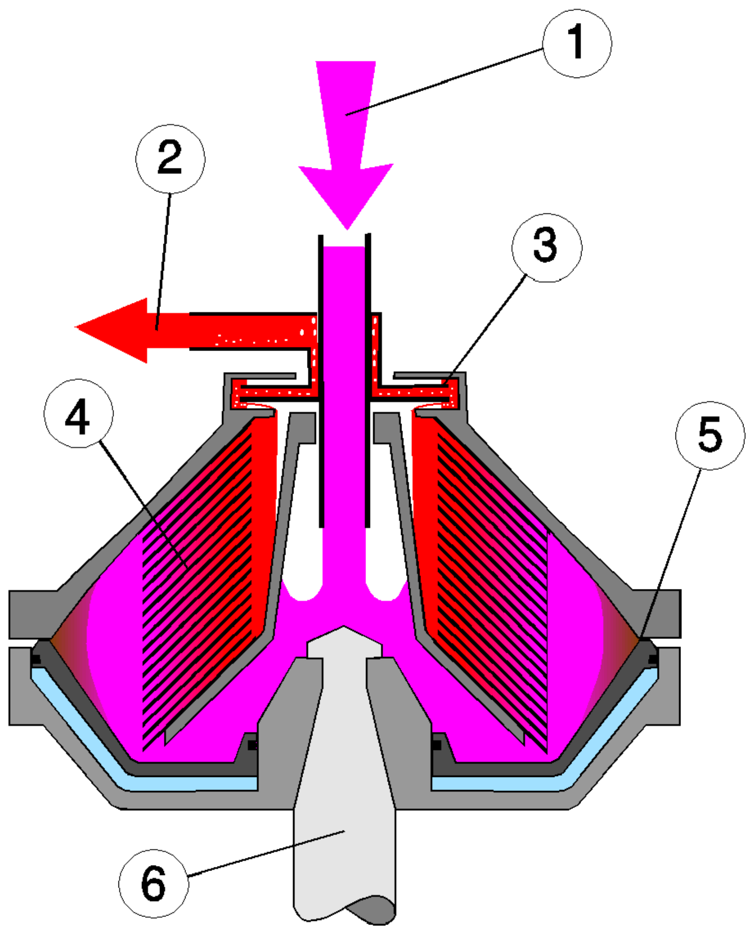
To be able to rotate the bowl it needs to be fitted with a driving mechanism. This is done by mounting the bowl on a spindle unit, also called a bowl drive cartridge. The bowl drive cartridge is then attached to a motor via a worm gear or a belt drive (Alfa Laval n.d.).

3.1 Clara 80

Clara 80 is a high-speed centrifugal PX separator designed for clarifying beverages. Clarifying is the process of separating solids from a liquid phase. An overview of the inlets, outlets, pumps and valves of the Clara 80 separator is shown in Figure 3.2. The beverage to be clarified arrives to the ”Process liquid inlet” from the ”Inlet valve” if the valve is open and the ”Feed pump” is on. The clarified liquid leaves the separator through the ”Clarified liquid outlet” which contain a back pressure. The particles that are separated from the liquid are discharged from the separator through the ”Solids outlet” at regular instances.

The discharge sequence begins with the ”Flushing valve” opening to flush the exterior of the bowl to lower its temperature. Temperature is lowered in order not to burn and stick the ejected solids to the separator walls. The ports are uncovered by opening the ”Discharge valve” and the solids are ejected. After the solids have been ejected the ”Bowl close valve” is opened and the ports are again covered. Finally the ”Flushing valve” opens again to flush the bowl clean from any remaining solids.

The separator is driven by an asynchronous induction motor regulated by a *VFD* (Variable Frequency Drive) and connected to the bowl via a belt drive. A VFD controls both the frequency and voltage supply to the motor reducing



1. Inlet
2. Outlet
3. Paring disk
4. Disc stack
5. Port
6. Spindle

Figure 3.1 Cross-section diagram of a PX separator bowl.

wear on the driving mechanism. Frequency is varied when starting and stopping to respectively accelerate and decelerate the separator bowl. The VFD keeps the frequency at a fixed value when running the separator at full speed and does not compensate for any changes in the load. Consequently the bowl speed will drop if the load is increased, which is the case when the process liquid is let through.

3.2 Incorrect Behavior and Consequences

Separators are complex machines and many things can go wrong while operating them. This is especially interesting in this master's thesis because the simulator needs to behave in a realistic way even if operation is incorrect. The most important task of the separator control code is handle erroneous behavior in the separator and bring it to a safe state if something goes wrong. To be able to test that the control code does what it should when an error occurs

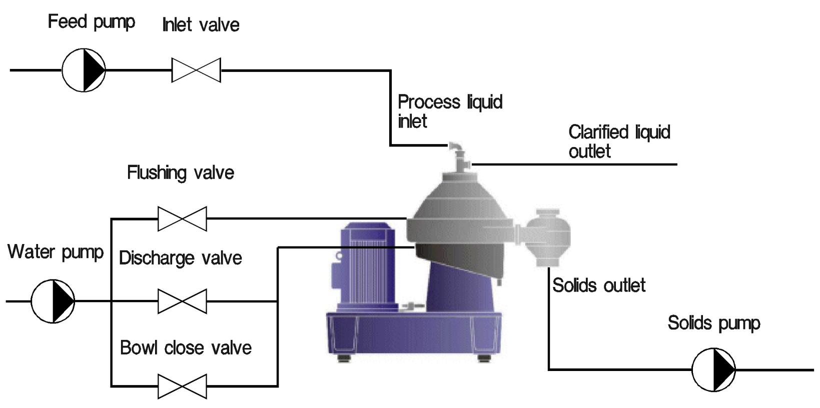


Figure 3.2 Overview of inlets and outlets of the Clara 80 separator.

the simulator must be capable of producing the same erroneous behavior as the real separator.

Some typical separator related errors are:

- Leakage
  - Internal leakage is when the unclarified liquid leaks to the clarified liquid inside the separator.
  - External leakage is when the clarified liquid level is below a certain radius and thus leaves the separator at the wrong place.
- Incorrect signals
  - Unresponsive pumps and valves.
  - Incorrect sensor data.
- Discharge
  - The sequence in which the valves are opened can be timed incorrectly.
  - Too long interval between discharges.
- Pressure
  - Too high clarified liquid outlet back pressure.
  - Too low process liquid inlet pressure.

Of course there exists numerous other errors that can occur and it is not feasible to consider all of them. It is not certain that the control code can handle all of these errors and some errors might only be observable visually.

4. Programmable Logic Controller

A *PLC* (Programmable Logic Controller) is a small computer used for automation of real-world systems, such as control of machinery on factory assembly lines. Where older automated systems would use hundreds or thousands of relays, a single PLC can be used as a replacement.

4.1 History

The first PLCs were introduced in the 1960's. A major US car manufacturer issued a request for a solution that helped them eliminate their complicated relay based machine control systems (PLCS.net 2005). This relay based system had to be re-wired by skilled electricians every time the production line needed to change which cost a lot of money and was very error-prone. Bedford Associates provided the winning solution when suggesting a Modular Digital Controller (MODICON) which became the first commercial PLC and was called MODICON 084 (PLCS.net 2005).

These new controllers had to be programmed by the same people that had worked with the old relay systems and the learning threshold had to be kept to a minimum. The answer was to use ladder logic, an old programming technique that very much resembled the way relays were wired (Hendricks 1996). This made it easy for maintenance and plant engineers to program and update the PLCs.

In the 1980's, the PLCs became programmable through symbolic programming on personal computers instead of dedicated programming terminals or handheld programmers. The 1990's brought the programming standard IEC 61131-3, which will be dealt with more in detail in Section 4.4 (PLCS.net 2005). In 1992, a manufacturer- and product-independent international organization was founded called PLCopen (PLCopen 2005). The aim of this organization is to promote the development and use of compatible software for PLCs and to help existing standards gain international acceptance (John & Tiegelkamp 2001, pp. 16,17). This organization is very committed to the IEC 61131-3 standard.

4.2 Soft PLC

A *soft PLC* is, simply put, an implementation of a PLC on a PC-based system. The basic idea behind soft PLCs is to make use of the enormous development of the PC hardware industry. PCs give remarkable computing power for their price and even though a PC is not primarily designed for real-time applications it can still be used in many demanding applications due to its high performance. In addition PC-based systems provide good networking possibilities both to existing computer systems and to existing PLC networks (Olsson & Rosén 2004).

Many different factors may have triggered the development of soft PLCs. One of them was probably that the industry was used to the scalability of PC-based systems and grew tired of the PLC-manufacturers whose products very seldom were compatible with anything but their own products.

4.3 TwinCAT

Beckhoff Industrie Elektronik has presented a tool, *TwinCAT* (The Windows Control and Automation Technology), that turns a PC using Windows into a real-time controller with a multi-PLC system containing four PLCs. TwinCAT is compatible with Windows NT/2000/XP, NT/XP Embedded and Windows CE (Beckhoff Gmbh 2005). TwinCAT also includes a development environment in which applications are written using any of the languages specified by IEC 61131-3.

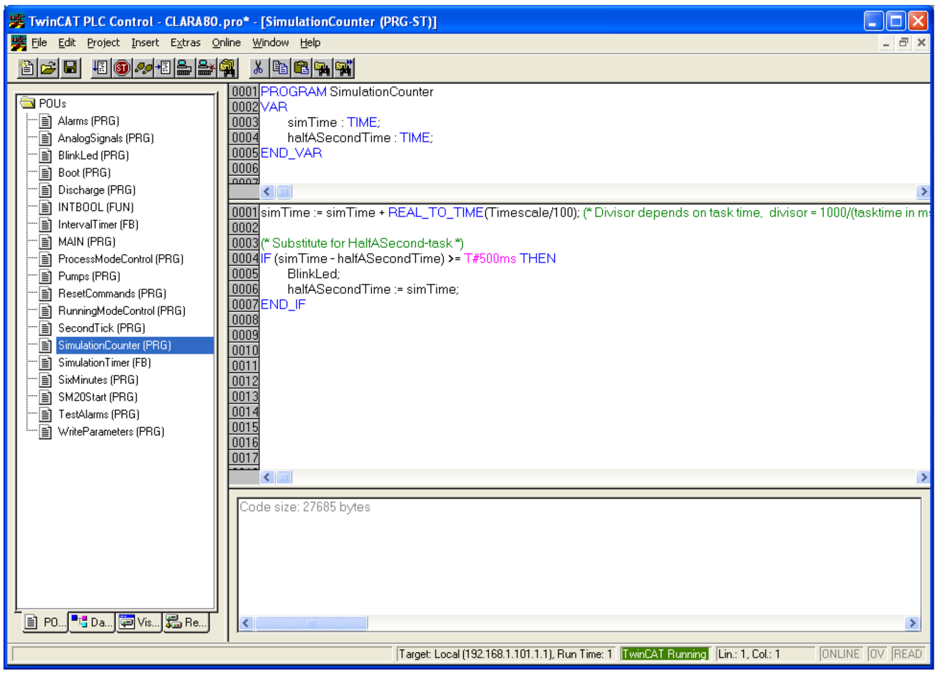
Using an operating system like Windows NT for real-time operations is bound to create problems. Windows NT is not truly capable of real-time operations since control tasks can be interrupted by events such as network operation, hard disk access and mouse movements. Beckhoff has solved this by developing their architecture as an independent real-time extension for Windows NT in the form of a kernel mode driver. With this kernel extension, TwinCAT operates on an exact time basis which executes programs with maximum priority and independently of other processor tasks that NT may be running (Hayes Control Systems 2001). This is achieved with 64 tasks which are executed with priority control, preemptively and deterministically with a maximum of +/- $15\mu s$ jitter according to Beckhoff Gmbh (2001).

The TwinCAT run-time system can support four virtual PLCs at the same time, each with up to four tasks. Each task executes a program block (for example a Main block) with a preset time period. It is possible to start and stop a PLC but not the individual tasks through the communication interfaces. A screen shot of TwinCAT can be seen in Figure 4.1.

Connection Interfaces

The TwinCAT architecture allows for independent modules of the system to be treated as individual devices; *servers* and *clients*. The devices are virtual devices in the form of software. The servers in the system are working devices which, as far as behavior is concerned, correspond to exactly one hardware device. The clients are programs that request services from the servers, such as a visualization application (Beckhoff Gmbh 2001).

All PCs with a TwinCAT installation contain a *message router* (Beckhoff Gmbh 2004*a*). This router manages and distributes all messages between the different devices in the TwinCAT system. All users, servers and clients, of the *AMS* (Automated/TwinCAT Message System) must register with the message router. TwinCAT servers have fixed port numbers and clients are assigned a port number by the message router. The message router itself has a unique number called *AMSNetId* (AMS Network Identifier). A PC in a TwinCAT network is defined by its IP-address and its AMSNetId. The message router uses a protocol called *ADS*/AMS (Automation Device Specification), see Figure 4.2. ADS will be addressed later in Section 4.3. This protocol is based on



```
PROGRAM SimulationCounter
VAR
	simTime : TIME;
	halfASecondTime : TIME;
END_VAR
```

```
simTime := simTime + REAL_TO_TIME(Timescale/100); (* Divisor depends on task time,  divisor = 1000/(tasktime in ms

(* Substitute for HalfASecond-task *)
IF (simTime - halfASecondTime) >= T#500ms THEN
	BlinkLed;
	halfASecondTime := simTime;
END_IF
```

```
Code size: 27685 bytes
```

Target: Local (192.168.1.101.1.1), Run Time: 1 | TwinCAT Running | Lin.: 1, Col.: 1 | ONLINE | OV | READ

Figure 4.1 TwinCAT.

TCP and ensures an exchange of data even with remote servers on other PCs or field devices (Beckhoff Gmbh 2001).

AMS/TCP Header
AMS Header
ADS Data

Figure 4.2 ADS/AMS packet structure.

TwinCAT is also able to communicate with other programs and hardware using the open standard *OPC* (Object, Linking and Embedding for Process Control) by providing an OPC-server which works as a gateway to ADS (Beckhoff Gmbh 2004*b*). OPC is further explained later in this section.

ADS Interface

ADS describes a device- and fieldbus-independent interface managing the access to ADS devices. Beckhoff provides the following software objects to add ADS communication to objects (Beckhoff Gmbh 2004*a*):

- ADS .NET component – for use in a .NET environment.
- ADS-OCX – for use in Visual Basic, Visual C++, Delphi, etc.
- ADS-DLL – for use in Visual C++, etc.
- ADS-Script-DLL – for use in VBScript, JScript, etc.

An ADS communication begins with an ADS request, which arrives at the ADS server where it is viewed as an ADS indication. The ADS server replies with an ADS response, which in turn is received by the ADS client as an ADS confirmation. Data can be exchanged in three different ways (Beckhoff Gmbh 2004*a*):

- On demand
  - Asynchronous
  - Synchronous
- On change
- Cyclic

On demand communication requires an explicit request to read or write data. An asynchronous request enables the program to continue without waiting for a response whereas a synchronous request will halt the execution until the arrival of the response. Synchronous communication is not available in all communication objects provided by Beckhoff.

One great feature with ADS communication is that data can be exchanged *on change*. The client registers itself at the server for a specific variable. When the server detects a change of value in the registered variable it autonomously notifies the client by call-back. Updates will be sent until the client cancels the request for the variable. An advantage with this communication is that it enables the possibility to write an *event-based* program. The benefits of an event-based program are both low program and protocol overhead as no unnecessary requests are sent (Beckhoff Gmbh 2004*a*).

Cyclical communication works in the same way as on change, with the distinction that data is sent cyclically even when it has not changed. A selection of the available ADS commands according to Beckhoff Gmbh (2002) can be found in Table 4.1.

ADS Command	**Description**
Read	Reads data from an ADS device.
Write	Writes data to an ADS device.
Read State	Reads the ADS and device status of an ADS device.
Write Control	Changes the ADS and device status of an ADS device.
Add Device Notification	Creates a notification in an ADS device.
Delete Device Notification	Deletes a previously defined notification in an ADS device.
Device Notification	Forwards data independently from an ADS device to a Client.

Table 4.1 Overview of ADS commands.

OPC Interface

OPC is an abbreviation for ”OLE for Process Control” and is a series of standards specifications primarily for data exchange based on Microsoft’s OLE COM (Component Object Model) and DCOM (Distributed Component Object Model) technologies. The vision for OPC is to provide interoperability for moving information from the factory floor through a corporation of multi-vendor systems as well as providing interoperability between devices on different industrial networks from different vendors (OPC Foundation 2005).

4.4 IEC 61131-3 Specification

IEC 61131-3 is a global standard for PLC programming which consists of five different languages; *ST* (Structured Text), *IL* (Instruction List), *FBD* (Function Block Diagram), *LD* (Ladder Diagram) and *SFC* (Sequential Function Chart).

SFC, FBD and LD are all graphical languages where different building blocks are connected to each other. IL and ST are both text-based programming languages where IL is similar to assembler and ST is closer to Pascal

When programming an application, the languages can be combined freely. According to (John & Tiegelkamp 2001, Langnau 1995) an application is usually structured with SFC which divides the program into smaller units and describes the flow between them. The units themselves, also called *POU* (Program Organization Units), are programmed in any of the other IEC 61131-3 languages or consist of further SFC structures.

POU

The POUs consist of a declaration part and a code part. The declaration part, which is seen in the top right corner in Figure 4.1, defines all the variables that are to be used in a POU. In the code part, which is seen in the middle right part of Figure 4.1, the logical circuit or algorithm is implemented in the desired IEC 61131-3 programming language. No POU may call itself (recursion), directly or indirectly. This would make it impossible to calculate maximum memory usage which is required by PLC programs (John & Tiegelkamp 2001, p. 54).

A POU can be of three different types; Program (PRG), Function Block (FBD) and Function (FCN). A function is a POU without memory that can be assigned parameters and when called with the same input parameters always give the same result. The Function Block has memory and its result will not only depend on the input parameters but also on the state of its internal variables. According to (John & Tiegelkamp 2001, pp. 31, 41) this is the most frequently used building block and is the main building block for structuring PLC programs. The Program represents the main program and must contain a declaration of all the variables in the program that are assigned to physical memory. In all other aspects it behaves like a Function Block.

Data Types

IEC 61131-3 defines a standardized set of data types available in all of the defined languages. Their name, data width and value range are defined by IEC except for a few types (date, time and string). An overview of the types and their data widths is presented in Table 4.2.

Structured Text

Structured Text is perhaps the easiest language for an experienced programmer to learn as it resembles other high-level languages like Pascal and in some cases C. Although in ST memory management is not an issue as it is in C because memory cannot be dynamically allocated. An algorithm in ST is composed of *statements* controlling the flow and computation of the program.

Type	Description
BOOL	Boolean (1 bit).
INT	Different kinds of integers are available. Short Integer (8 bits), Integer (16 bits), Double Integer (32 bits) and Long Integer (64 bits). It is also possible to use unsigned integers.
Bit strings	Available in sizes ranging from 8 to 64 bits.
REAL	Floating point numbers, available as Real (32 bits) and Long Real (64 bits).
Special	String, date and time of day.

Table 4.2 Overview of IEC 61131-3 data types.

A simple example of an ST program showing some of the most frequently used statement structures can be found in Figure 4.3. Note that all statements are separated by a semi-colon, not by line breaks. It is thus possible to write a complete statement on a single line or to break up a statement over several lines. A more thorough guide to ST can be found in (John & Tiegelkamp 2001, s. 4.2).

```
IF a < b THEN
  c := 1;
ELSIF b = c OR a = c THEN
  c := 2/a;
ELSE (* comment *)
  WHILE b > a DO
    a := b*4;
    IF a >= d THEN EXIT;
  END_WHILE;
END_IF;
```

Figure 4.3 Example of an ST program.

Sequential Function Charts

An SFC is basically composed by *steps*, *transitions* and links. Each step has an associated set of instructions written in any of the IEC 61131-3 languages that is executed when the step is active. Each transition has a boolean *transition condition* defining when the *token* showing active step should be passed on. The links connect the steps and transitions to a network. An example of an SFC can be found in Figure 4.4.

It is possible by parallel branches to be in two or more steps at the same time. A parallel branch has to be started and ended. It is not allowed to be in more than one step outside a parallel branch, such a network is called *unsafe*. If a network contains steps that can not be activated it is called an *unreachable network*. For a more detailed review on SFC, please review (John & Tiegelkamp 2001, s. 4.6).

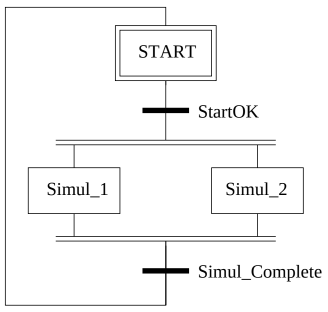


Figure 4.4 SFC example with a parallel branch.

5. Modeling and Simulation

When constructing a model of a real system it is important to understand what a model really is. The word model can have many meanings but according to (Zeigler, Kim & Praehofer 2000, p. 29) the most common concept of a model is that it is a set of instructions, rules, equations or constraints for generating an Input/Output-behavior. They also state that a model can be conceived as any physical, mathematical or logical representation of a system, but defining it in terms of system specifications gives it the advantage of a good mathematical foundation and definite semantics.

5.1 Choosing the Right Model

It is important to establish the objectives of the model as early as possible. Zeigler et al. (2000, p. 27-28) says that the objectives serve to focus the construction of a model on particular issues. Different users have different objectives when simulating a system. Olsson & Rosén (2004, p. 268-269) presents six different categories within the industrial spectra which all have different needs. Some examples are the *educator* who use simulations for teaching the system dynamics, the *control engineer* who needs to test different controllers and algorithms and the *designer* who will use simulations to test new design ideas.

All these users have immensely different objectives when simulating, and it may not even be viable to produce a simulator covering all the objectives. Some of the users need to change all parameters, sometimes even the model, whilst others value a logical graphical user interface. The important thing is to identify the users and their needs as early as possible as this will be of great help when creating a suitable model and designing the simulator.

5.2 Model Versus Reality

The one thing that can be said about all models is that they are all simplifications of their source systems. The complexities of any system introduce limitations that have to be made in order to make the model maintainable. Even when most or all of the dynamics behind a system is known, it may have to be simplified due to limitations in computational performance of the computer running the simulator. By no means will a model ever be able to fully mimic every aspect of reality (Zeigler et al. 2000, p. 32), but as will be discussed in Section 5.4 the model can be valid in the simulation scope anyway. In fact, when constructing a model one should strive after a model that is as simple as possible but still valid for the simulation purposes.

5.3 Simulator

A simulator can be distinguished by describing it as the process executing a model. Separating the simulator from the model opens the possibility for the model to be implemented in different environments and thereby offering portability (Zeigler et al. 2000, p. 30). Two different simulators based on the same model should give the same output and by comparing the outputs a form of simulator verification is achieved. A properly built simulator should be able to execute several different models in the same model spectra. In a simplified manner it can be described as a car simulator should be able to simulate different types of cars but does not have to simulate trains.

Simplifications do not only occur in the modeling phase, when constructing the simulator choices have to be made concerning numerical accuracy. Numerical methods for integration and other more complex mathematical situations differ in accuracy. There is often a trade-off between the level of exactness and level of complexity in the calculations. Some methods fit some situations better than other, and there is no ”best” way to perform a numerical integration (Olsson & Rosén 2004). This proves to be important in continuous system modeling and simulation discussed in Section 5.8.

5.4 Validation

Validation is often described as the process of confirming that the right model has been produced. The ”right model” is the model that correctly represents the real system. When validating a model for a simulation a narrower definition has to be used by practical reasons. A valid model correctly represents and mimics the behavior of the relevant inputs and outputs of the source system.

”Relevant” in this context are inputs and outputs that will be monitored and experimented with during a simulation. For a car simulator used for simulating braking behavior, relevant inputs and outputs could include velocity, brake disk plate size and applied force. Irrelevant information about engine efficiency and windscreen wiper position will not affect the validity of the model.

5.5 Verification

The simulator should correctly generate the desired outputs of the model. While validation can be expressed as ”building the right model”, verification is sometimes expressed as ”building the simulator right”. Verification can be hard work, but it pays off well. The results from a simulation are never more correct than the simulator and ensuring that the simulator is verified increases the confidence in the results. Much skepticism must be applied to the verification and consequently to the results of a simulator.

5.6 Time in Simulation

In all types of simulators time is a factor of great importance. On an abstract level, time can be defined as a quantification of the interval between events or observations. When simulating, time can be divided into two categories; logical time and physical time (Zeigler et al. 2000, p. 34). Physical time can be said to be a real time, it is measured by ticks of physical clocks. A logical time on the other hand is included in the model, the time base is abstract and actual time between successive ticks does not have to be constant.

Time can furthermore be divided into global and local time (Zeigler et al. 2000, p. 34). A simulator operating on global time shares the time (physical or logical) with all its components. Consequently, a simulator operating on local time use different clocks in its components.

Time Scale

Different models may require different time scales depending on the underlying system. This introduces the problem of time scales. When modeling an industrial process, different units have vastly differing time scales ranging from milliseconds to months (Olsson & Rosén 2004). One solution sometimes favored is decoupling the unit dynamics, this proves effective when organizing the model.

5.7 Discrete Time Models

A discrete time model is a function that returns information of the next state given the current state and the input from the environment. This is the simplest and most common type of simulator. Time between steps can be constant or determined by the current state and inputs. By choosing a small time step the discrete time model often approximates a continuous system.

The next state is dependent on current state and inputs

$$x(t_{k+1}) = f(x(t_k), u(t_k)), \tag{5.1}$$

if a non-constant time is used it can be calculated as

$$t_{k+1} = g(x(t_k), u(t_k)) \tag{5.2}$$

where t_k is the k-th time instant and u is the input.

5.8 Continuous Time Models

In a continuous time model states are not changed in discrete steps. When modeling a system, differential equations often arise. In a continuous time model time is not described by steps but as a continuous flow, consequently the state changes in the same way. Instead of calculating the next state, the rate of change of the state is calculated.

The continuous model can be described correspondingly to the discrete model as

$$\frac{dx}{dt} = f(x(t), u(t)). \tag{5.3}$$

The function $f(x)$ can be complex and contain logical decisions. Its behavior can vary depending on the inputs.

Continuous Model on Digital Computers

As a simulator most certainly will be run on a digital computer, the continuous model has to be discretized. An approximation of a continuous model on a digital computer results in new difficult problems that must be managed with care. Numerical integration is one of the problems. The simulator will only have knowledge about the state in discrete time steps, therefore values in between two steps will have to be estimated.

One of the simplest methods of numerical integration is the *Euler* method (Zeigler et al. 2000, p. 51)

$$q(t+h) = q(t) + h\frac{dq(t)}{dt}. \tag{5.4}$$

The Euler method is easy to use and understand but can introduce large numerical errors. The error can be decreased by using a smaller h (step size). More refined methods, such as *Runge-Kutta* (Olsson & Rosén 2004, p. 281), *Adams* and *Heun* (Zeigler et al. 2000, p. 53,54) are typically more accurate while preserving speed.

Both Olsson & Rosén (2004) and Zeigler et al. (2000) suggest that a variable step size method introduced by an error criterion generally is a good choice. This may however introduce new problems, for example when using the methods on a non-smooth trajectory it may not be possible to find a sufficiently small h to preserve the error criterion. The exact error is of course impossible to calculate as the true solution is not known, which causes the error to be estimated.

5.9 Discrete Event Models

An event is something that changes the state of the system (Olsson & Rosén 2004, p. 270). Events can be both internally generated (endogenous) and externally generated (exogenous). An exogenous event is something affecting the system from the outside, in the car simulator previously mentioned, this could be a switch turned on by the driver. An endogenous event is something affecting the system from the inside, in the car simulator this could be fuel level events.

In between events no calculations are made, state changes can only occur on the arrival of an event. This is often called an *event driven* model (Olsson & Rosén 2004, p. 270). New events can only be scheduled at the start of a simulation or on event arrivals. The arrival of an event can also cause previously scheduled event to be cancelled (Zeigler et al. 2000, p. 69), this is a situation that can bring problems to the simulation.

Two events scheduled to arrive at the same time may cause the other one to be cancelled, which of the event should then be evaluated first? One solution to this problem is prioritizing the events.

5.10 Hybrid Models

It is common to mix a discrete event model with a continuous time model (Olsson & Rosén 2004, p. 270). These systems are called *Hybrid models*. Shanthikumar (1983) divides hybrid models into different types where the one most important to this master's thesis contains a continuous model and a discrete event model who work in parallel with interactions between them. Most combined continuous and discrete event models belong to this type because the continuous portion of the combined model usually uses a numerical solution procedure to solve a set of differential or difference equations, which is a continuous model.

In many systems alarms occur after the system has been in a state for some time, valves can be opened or closed and the operator can manually change states while the system at the same time functions as a continuous system.

5.11 Variable Simulation Speed

A great feature a simulation environment often possesses is the ability to change the time scale. Some processes are very slow and a real-time simulation of such a process can be meaningless. Simulation of a cancer tumor or the earth's orbit around the sun may be unhelpful if not speeded up.

There are also some fast processes that may need a slower simulation. Examples of such processes might be simulation of the signals involved when an eye blinks.

6. Simulation Solutions

There is an abundance of simulation software available on the market. Many of these are highly specialized and others are aimed at a more generic use. No standards apply because different tools have different applications depending on the field of interest. Another possibility is to construct a specialized simulation tool from scratch providing only the required simulation features.

6.1 Writing from Scratch or Using an Existing Tool

Instinctively the most natural choice would be to use an existing tool since more focus can be directed at the model. One drawback can be that it is often only possible to simulate a subsystem and direct connection to a real control system can be hard to achieve. Another major drawback is that most simulation programs cannot provide a feasible stand-alone executable making it impossible to run the simulator without the simulation tool. If the system is complex, for example containing advanced integrations or several differential equations using different time scales an existing tool would prove very effective.

By writing the tool from scratch using a high level programming language full control can be achieved. The tool itself does not expose any limitations, it is the experience and proficiency of the programmer that sets the boundaries. For example it is easier to make a specialized user adapted interface. It is important to exhaust all other possibilities of using an existing tool before writing a new one as there is lots of time to save by not ”re-inventing the wheel”.

The complexity of the model sets the first boundaries when choosing simulation solution. It is also important to consider how and for what purpose the simulator will be used. It is not always possible to select a single best solution. A problem can be solved equally well with two different approaches.

6.2 Chosen Simulation Solutions

More than 20 different simulation software solutions were briefly reviewed, out of them six were chosen for further examination, three programming languages and three simulation tools. The six solutions were chosen primarily because their availability and documented use in other simulation applications.

The following solutions were considered:

- Simulation tools
  - VisSim (Visual Solutions Inc. 2005)
  - Matlab/Simulink (MathWorks Inc. 2005)
  - Modelica/Dymola (Dynasim AB 2005)
- Programming languages
  - Java (Sun Microsystems Inc. 2005)

- ◦ C# .Net (Microsoft Corp. 2005)
- ◦ Visual Basic .Net (Microsoft Corp. 2005)

6.3 Simulation Tools

When comparing simulation tools, cost, adaptability and usability are important factors. The requirements established later in Section 8.2 acted as a basis for important features when comparing different solutions.

VisSim

VisSim is a graphical simulation tool from Visual Solution Inc, which is highly configurable and has the possibility to model complex dynamics. It is possible to connect to TwinCAT via an OPC-server, but not directly to TwinCAT using ADS. Hiding dynamics is not realizable in a simple manner, and the program is not portable since VisSim has to be installed in order to run a simulation.

Simulink

Simulink from MathWorks Inc. is one of the most used simulation tools. Its model based design provides great possibilities of simulating complex systems. Together with plug-ins it is possible to generate C code which can be compiled to an executable. Plug-ins for communication over OPC are available and using blocks and C code compilation would make it possible to hide dynamics. The main drawback is the cost as the initial costs including needed plug-ins would exceed 50 000 SEK.

Modelica

Modelica is an object oriented programming language used for mathematical modeling of complex dynamic systems. It is possible to create advanced models of a great variety of processes. Dymola from Dynasim AB is a combined editor and simulation tool for the Modelica language. There are no available plug-ins for ADS communication. Dymola can create C-code for running the models in real-time and hardware in the loop situations. A different approach with a Modelica-to-C compiler is discussed in a paper by Freiseisen et al. (2002), but this compiler only use a subset of the Modelica language. Dymola is available in the same price interval as Simulink.

6.4 Programming Languages

Programming languages require a somewhat different approach when evaluating pros and cons. An efficient programming platform enhances the capabilities of a programming language and is therefore an important factor.

Java

One of the great advantages of Java is its portability. As long as no native code is included a Java-program can be run on almost any platform. There are public solutions for communicating with an OPC-server and it is possible to make an implementation of the ADS-protocol. Java as a programming language is free and there are even some free effective programming environments, such as Eclipse (Eclipse Foundation 2005). There are no standard solutions for advanced computations, it is however possible to implement such functionality through either third party packages or own ideas. This is not trivial and may be strenuous and time consuming. Dynamics are effectively hidden from the user as long as no source code is released.

C# .Net / Visual Basic .Net

C# .Net and Visual Basic .Net are both Windows-specific programming languages. In the .Net-version C# and Visual Basic are actually the same languages but with different syntax as they share the same MSIL (Microsoft Intermediate Language). C# and Visual basic share the same limitations and possibilities in computational abilities with Java. Communication can be done directly over ADS as Beckhoff provides both an ActiveX controller and a .Net component. Dynamics are hidden in the same manner as in Java.

7. Testing Software

Testing has been a natural part of software engineering as long as software has been written. It can basically be seen as the method which the programmer uses to find errors in his or her software. Graham (1994, p. 1330) claims that testing software is a way of assessing the quality of the software. With assessing the quality of the software she means determining what the system actually does, and how well it does it in its final environment. She says:

> A system without testing is merely a paper exercise; it may work or it may not, but without testing, there is no way of knowing this before live use.

This definition of testing does not include the actual debugging of the software. Debugging is just a correction process and writing correct software is just a small step on the road to high quality software.

The purpose of testing is of course to gain confidence that a piece of software does what it is supposed to. Confidence is boosted whenever a test is performed and the result turns out as expected. When constructing tests there are a few things to keep in mind to get better results (Graham 1994, p. 1331):

- Tests must be exactly reproducible.
- The result of a test must be defined.
- The result of the test must be compared with the expected result.

These conditions might appear obvious but are sometimes overlooked. Especially the second condition is vital because if the tester does not know what he is looking for he might accept the result of a test without really evaluating if it is correct or not.

To make testing even more efficient one must consider something called *the testing paradox* (Graham 1994, p. 1332). As mentioned above testing is about gaining confidence, but also about destroying it. The goal of testing should be to find errors. A successful test is a test that finds an error, such a test is more valuable than 100 tests that do not find any errors. This is summed as the testing paradox where you want to gain confidence by trying to destroy it.

7.1 Limitations

The only thing that is certain when it comes to testing is that there are always more errors to find and testing can never prove absence of all errors (Graham 1994, p. 1332). This is why it is so important to write efficient tests because testing is never finished and at some point the testing has to be abandoned. At that point the quality of the software is highly correlated with the efficiency of the tests.

7.2 Levels of Testing

Testing is done on several levels in software development. The testing has different objectives at each level (Graham 1994, p. 1333).

Unit Testing

A *unit* is often defined as the smallest testable piece of software (Graham 1994, p. 1333). Unit testing is usually performed by the developer and searches for errors in the individual units of the software.

Link Testing

A piece of software consists of several connected units or groups of units, often called components. A separate unit might not cause an error before it is linked together with another unit. Testing for these kinds of errors is called *link testing*. A more suitable term for this kind of testing would be ”Integration testing” but it is often used to describe the integration of high-level systems (Graham 1994, p. 1334) instead of the integration of low-level units.

System Testing

System testing is concentrated on the whole system and looks for errors in the functionality throughout the system. This is also where nonfunctional quality attributes, such as reliability, stress tolerance, usability etc., are tested and evaluated (Graham 1994, p. 1334).

Acceptance Testing

Acceptance testing is a way to safely hand over the responsibility of a system to the users. It gives confidence both to the developer that the system performs as promised and to the users that the system is ready for use (Graham 1994, p. 1334). The difference between the above mentioned test levels, often called *development testing*, is that acceptance testing is constructed to show the correct behavior rather than finding errors.

Regression Testing

Regression testing is about guaranteeing that the software has not changed in an unexpected way when it is modified. This provides a very valuable tool for a developer who can add, change or delete functions in the software without worrying about unexpected effects.

7.3 CAST – Computer-Aided Software Testing

The benefits of being able to use a computer to aid in software testing are numerous. It improves the quality of the software development process and not only the testing of software. The productivity can be increased dramatically which consequently reduces the cost of testing (Graham 1994, p. 1347). Some of the benefits are:

- Large volumes of tests can be run without supervision.
- Accuracy can be increased in monotonous repetetive tasks.
- Regression tests can run automatically whenever something is changed.

Risks with CAST

Computer automated testing is a great tool to find trivial errors however it is important to remember that a tool can only find the errors it is capable of looking for (Graham 1994, p. 1347). It is also important to remember to put realistic expectations when implementing a tool. A testing tool is merely an aid and does not take care of the whole testing process. Without proper testing routines there can be no guarantee of improved quality. As Graham (1994, p. 1347) claims:

> Automated chaos is just faster chaos

7.4 Testing Techniques

Software testing is often divided into *static* and *dynamic analysis*. Static analysis consists of techniques that examine the software in different ways to improve its quality whereas dynamic analysis techniques exercise the software with different sample inputs (Graham 1994, p. 1341). The dynamic techniques can be further divided into functional techniques (*Black Box*) and sequential techniques (*White Box*). Black Box testing is most relevant to this master's thesis and will be dealt with in the following chapter. The interested reader can find more information about other testing techniques from other sources, such as (Hetzel 1988). A common convention when testing is to first do a static analysis and then follow up with one or several dynamic testing techniques where the best result is achieved using as many relevant techniques as possible (Graham 1994, p.1341).

Black Box Testing

Functional testing is called Black Box testing because it does not concern what the inside of the system looks like. Tests are created based on some form of specification that describes the functions of the system at different levels. It is important in Black Box testing to choose test cases that test the system in a thorough and systematic way. A test case usually consists of a sample input and the expected result that is compared with the actual result (Graham 1994, p. 1341).

7.5 Testing PLC Software

What can be done with PLC software and testing? Why is testing so important? According to (Graham 1994, p. 1330) software developers spend around 40% of development costs and time on testing. This figure can be much higher depending on what type of software is written. Hassapis (2000, p. 1-2) estimates in his article that testing comprises more than 50% of the total development costs when programming software for PLCs. He believes that the reasons for this high cost are:

- Modern methodologies cannot be applied to PLC languages due to the unstructured nature and inconsistencies in the semantics of the languages
- There is a lack of tools that allow the software to be tested before being used live.

The IEC 61131-3 standard as mentioned in Section 4.4 takes care of the first problem by providing a set of structured programming languages. The second requirement is solved to some extent by the debugging facilities of today's PLC programming environments that allow the software to be simulated with simple input signals, such as steps and ramps. However Hassapis (2000, p. 2) explains in his article that there are ways to improve the pre-testing of software further by using a dynamic simulation technique.

8. Practical Work Progress

This section describes the working progress of the development of the simulator. The steps were performed in an iterative manner even though the presentation here suggests that they were performed sequentially. After the first version of the simulator had been implemented, the model had to be revised and a new iteration was started.

8.1 Users of the Simulator

There are several possible users of a simulator. Some would like to use it for education and others for testing dynamics. The focus in this master's thesis is on testing code. Therefore, the main users of the simulator are PLC software developers using the simulator as an aid when designing control code for separators. Users should be familiar both with TwinCAT and with some generic programming, preferably C#.

8.2 Establishing the Requirements

With the users identified, their corresponding requirements had to be determined. Together with the customer, ideas on practical program features were formulated. Each feature was discussed concerning its use and functionality, both in terms of complexity and flexibility.

Complexity and estimation of implementation time worked as a guidance when prioritizing the features. Three levels of prioritization where used, *Must-have*, *Ought-to-have* and *Nice-to-have*. It was estimated that all requirements in "Must-have" and "Ought-to-have" would be realized within this master's thesis. These requirements provided a good basis for deciding model complexity and choice of simulation tool.

Must-have

Must-have features are the basic requirements for the simulator. These requirements were given the highest priority when implementing the simulator. It is not possible to place all requirements under "Must-have" since that would be the same as having no prioritization at all.

Executable application The simulator must be executable on a Windows computer without the need of external third-party programs. The simulator must not reveal its internal dynamics to any unauthorized user.

Connection to TwinCAT The simulator must be able to connect to TwinCAT since TwinCAT contains the code to be tested. Software communication possibilities to TwinCAT are available via ADS and OPC.

Graphical interface The simulator must have an easy to use graphical interface. The interface must provide controls for the simulation.

Simple separator model The simulator must be based on a simple separator model which contains the most important dynamics. The simulator is allowed to generalize certain behavior if needed to keep the model simple.

User manual The simulator must have an easy to follow manual written in English.

Ought-to-have

Ought-to-have features are features that significantly increase the value of the simulator. These features are important but not essential for the simulator.

Variable simulation-time The simulator ought to be able to change its simulation speed during operation. The change in simulation speed must be done both in the simulator and in TwinCAT. The simulation speed ought to be changeable both upwards and downwards.

Automatic testing The simulator ought to be able to run automated tests. The automated tests should be written in advance and be able to run without supervision.

Manual configuration It ought to be possible to change the configuration of the simulator in order to be able to simulate other separator types. The changes should be done manually. A certain degree of programming experience will be required.

Logging The simulator ought to be able to log and present its variables. The presentation of the variables need not be available during operations.

Installation program The simulator ought to be packaged in an installation program for easy distribution.

Nice-to-have

Nice-to-have features are features likely to be included in a future version of the simulator. It is also possible to list features that someday might be possible to implement.

OPC communication The simulator could allow other programs to connect to it via OPC. This would allow code to be tested in other programs than TwinCAT.

Semi-automatic configuration The simulator could be configured using a user interface to work with other separator types. This would allow users without programming experience to configure the simulator.

8.3 Model Development

To model a separator, enormous amounts of knowledge about the separator physics are required. As the authors do not possess that kind of knowledge, help from Alfa Laval was required.

Peter Thorwid was particularly helpful with outlining the important parts of the separator system. Hans Moberg provided equations describing the separator dynamics and Roland Isaksson derived equations of the motor and VFD dynamics easily adaptable to the separator dynamics equations.

The equations describing the separator, motor and VFD (Variable Frequency Drive) dynamics form a continuous model of the separator system. In total there are about ten algebraic and one differential equation that constitute the continuous model. The contents in these equations are controlled by discrete events such as alarms, valves opening and closing and operator inputs. A hybrid model, as described in 5.10, fell as the natural choice because it combines the continuous model with a discrete event model.

The basic idea behind the model is a balance equation containing energy produced in the system, and energy currently consumed in the system. Any remaining energy will accelerate the system until balance is achieved, in the same way a lack of energy will decelerate the system. In the model, energy is produced by the drive system and consumed by the separator bowl. In the bowl, energy is consumed by the process liquid flow, air friction and friction on the paring disk. A constant mechanical friction from bearings and belt drive was also added to the model. The model can then be expressed as

$$M_{motor} - M_{flow} - M_{air} - M_{disc} - M_{const} = J\dot{\omega} \tag{8.1}$$

where M is the torque produced or consumed, J is the bowl moment of inertia and $\dot{\omega}$ is the bowl acceleration. The differential equation is solved with the Euler method discussed in Section 5.7. The Euler method was chosen because it was easy to implement and was sufficiently accurate.

The exact model is confidential, but some notes about the underlying physics can be revealed. The flow torque (M_{flow}) depends on the flow rate, bowl size and speed and the density of the liquid. The air friction torque (M_{air}) depends on air density and bowl size and speed. The paring disk friction torque (M_{disc}) is the most complex, it is primarily dependent on density, speed and bowl specific parameters.

The motor model is based on the asynchronous induction motor together with a VFD. In the asynchronous motor, the rotation speed is always lower than the rotating magnetic field in the motor. The rotation speed is also dependent on the load. The motor torque is controlled by varying the frequency using motor parameters such as rated speed, power and mains frequency to determine motor behavior.

Erroneous Behavior

Not all of the erroneous behaviors discussed in Section 3.2 was included in the final model. The focus was not on modeling all possible errors, but rather to demonstrate how errors can be generated. Most errors are modeled as discrete events, a pump either starts or not, a valve either opens or not and external leakage gives a constant increase in air density independent on how much fluid is leaking. Internal leakage was not included in the model, mainly because the quality of separation is not relevant when testing code. All these simplifications lead to a less accurate model of the separator, but the model can none the less be accurate enough for testing code.

Validating the Model

The dynamical model was implemented at an early stage in Simulink to give a rough estimation of its validity. Plots from several sequences such as startup, discharge and stopping were reviewed by supervisors at Alfa Laval to validate the model. This validation only tested the validity of the mathematical model concerning the relevant dynamics of the separator. Another part of modeling is to identify the inputs and outputs needed to fully test the separator code. This was performed by reading the actual IEC 61131-3 code written in TwinCAT and reviewing a process and instrument diagram. All inputs and outputs were located and their cause and effect were identified.

Some of the most important inputs and outputs are:

- Inputs to TwinCAT
  - FlowRate – Current outlet flow.
  - Alarm inputs – Emergency button and cover sensor.
  - Pumps – Pump status for inlet, water and solids pumps.
- Outputs from TwinCAT
  - Valves – Open/close discharge and inlet valves.
  - Pumps – Start/stop inlet, water and solids pumps.
  - VFD control – Starts and stops the VFD.

8.4 Choosing Simulation Solution

In the end the choice was down to either Java or C# as Dymola and Simulink proved to be too expensive in comparison with the goals and requirements of the project. They both provide greater simulation possibilities than required by the current model, but remain interesting in a future simulator as discussed later in Chapter 9.

Among the programming languages Visual Basic was not chosen simply because the authors unfamiliarity with the syntax. Java and C# are close in syntax, and since both authors have experience in both languages the choice between them was syntax independent.

With Windows being the chosen platform, Java's portability is not beneficial to this master's thesis. The main advantage of using C# instead of Java is the ADS component provided by Beckhoff which gives a performance gain, but is also better in terms of usability compared to an OPC solution which would have been the choice with Java.

Choosing C# as the simulation tool made the choice of programming environment easy. Microsoft Visual Studio .Net 2003 is an easy-to-use tool for implementing Windows applications. Using .Net has one drawback, the .Net framework has to be installed. The framework can be bundled with the simulator installer to facilitate installation, but it requires the user to have *administrator privileges.*

8.5 Simulator Implementation

Implementation of the simulator was performed in several steps. These can be described as spiking, implementation, testing and releases (see Figure 8.1). To test ideas and new functionalities, spikes were used. Spikes were especially performed on ADS communication ensuring that all questions concerning how to connect to TwinCAT were covered before the implementation started.

The implementation was done in an XP-like fashion, defined in Section 2.3, with short iterations and new releases after each iteration. The requirements were divided into smaller tasks to fit into the short iterations. Each implemented task was tested by comparing extracted data from the simulator with expected data. The extracted data was also evaluated by the authors based on the gathered information. This process was repeated until the authors considered that the simulator fulfilled all the requirements under ”Must-have” and ”Ought-to-have”.

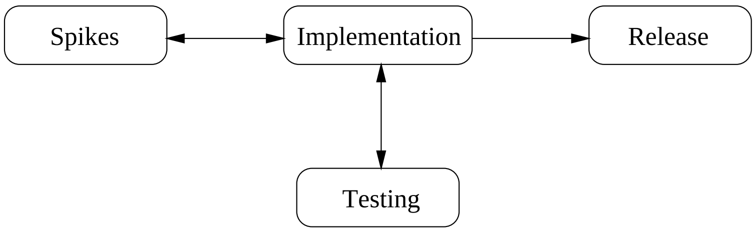


Figure 8.1 Implementation procedure.

Verifying the Simulator

There are several ways of verifying the simulator. By comparing data from the simulator to documented data from a real Clara 80 separator test, one level of verification was achieved. Some important courses of events that were compared are:

- Start
- Stop
- Discharge
  - Impact
  - Recovery
- Flow on/off

Another effective way of verifying the simulator is demonstrating it to experienced users of the real machine. In this project the simulator was demonstrated to the supervisors at Alfa Laval.

The Clara 80 separator was not available when the simulator was implemented which made direct comparisons impossible. By running both the simulator and the real separator concurrently and perceiving the same behavior a higher level of verification would be achieved.

9. Results and analysis

When concluding the results of this master's thesis, one must have in mind that the premier result is the produced simulator. During the project most of the time has been spent on making the simulator as versatile and accurate as possible within the given framework.

The important insight on how to continue and what to improve has evolved during the work. As the knowledge about the complexity of PLC programming and separator control has grown, both ideas and further questions have been formed.

9.1 The Simulator Suite

To facilitate further development of the simulator it is composed of modules as far as possible. An effort has been made to make a change of dynamics as smooth as possible. The simulator suite can further be divided into three different programs: a simulator, a tool for viewing log-files and a tool used for automated testing.

The Simulator

A screen shot of the simulator is shown in Figure 9.2. The slider at the bottom is used to control simulation speed. Pumps and valves use colors to show if they are on/open or off/closed. The "simulation control" is situated to the in a separate tool window and contains variables changeable at run-time. Starting and stopping of the simulator and logger is controlled via menus at the top of the windows. Connection to TwinCAT is also controlled via the menus.

The simulator is composed by a GUI module described above and a logic module. An overview of the separator system using the simulator can be seen in Figure 9.3. The logic module includes the actual simulator logic with exchangeable dynamics and a logger to enable logging of different simulator and separator variables at run-time. The simulator is partitioned not only due to program design reasons but also to make it possible to run a simulation without the need of a GUI. This is particularly desirable when performing automated tests in the Test Suite which will be explained later. It also facilitates when implementing new dynamics as the logic is concentrated to one interchangeable class.

The simulator is based on threading in order to handle GUI-requests without interrupting the simulation. By dynamically adjusting the sleep time, the mean period is kept constant. Tests have shown that the *period latency* is below $10ms$ at real-time simulation, see Figure 9.1. The period latency is the difference between scheduled wake-up and actual wake-up in the periodic thread.

Logging is performed by writing chosen values to a log-file. The log-file uses a simple XML-schema to store the values which makes it easy to parse the file using MSXML built-in functionality.

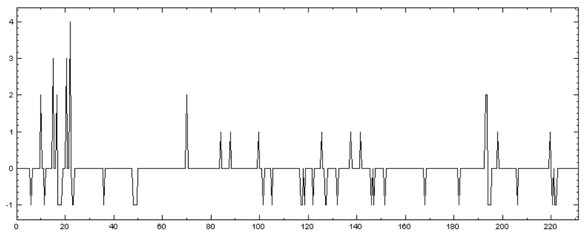
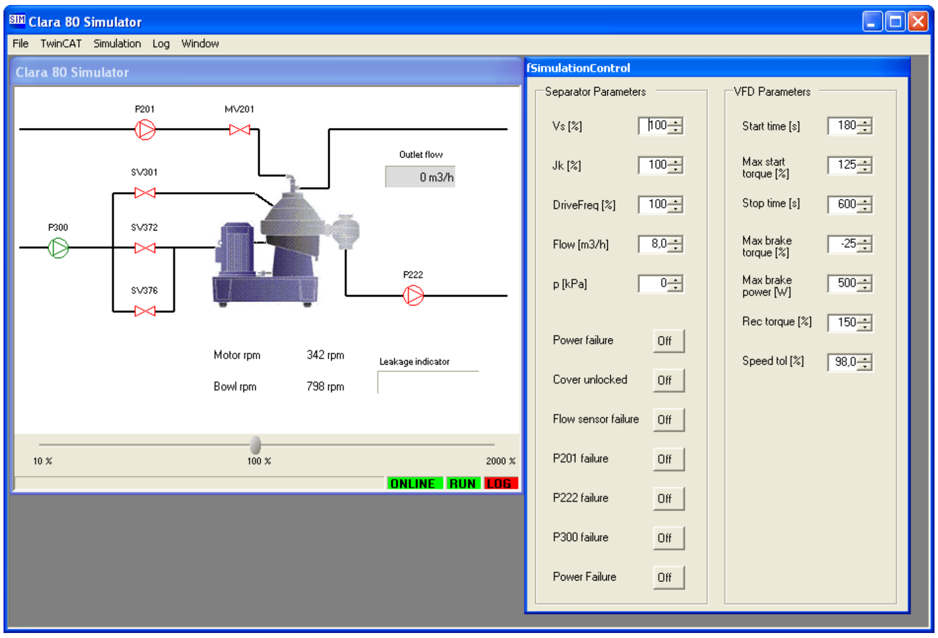


Figure 9.1 Period latency in the simulator at real-time simulation.



Clara 80 Simulator

Outlet flow 0 m3/h

Motor rpm 342 rpm

Bowl rpm 798 rpm

Leakage indicator

10 % 100 % 2000 %

fSimulationControl

Separator Parameters

Vs [%]	100
Jk [%]	100
DriveFreq [%]	100
Flow [m3/h]	8,0
p [kPa]	0
Power failure	Off
Cover unlocked	Off
Flow sensor failure	Off
P201 failure	Off
P222 failure	Off
P300 failure	Off
Power Failure	Off

VFD Parameters

Start time [s]	180
Max start torque [%]	125
Stop time [s]	600
Max brake torque [%]	-25
Max brake power [W]	500
Rec torque [%]	150
Speed tol [%]	98,0

Figure 9.2 Screen shot of Simulator.

Log View

Log View is used to open log-files produced by the simulator and to plot the results. Figure 9.4 shows the program in use. The menu is used for opening the log-files as well as printing and saving the plots as pictures. The tool is based on NPlot, an open source plotter with extensive features (Howlett 2005).

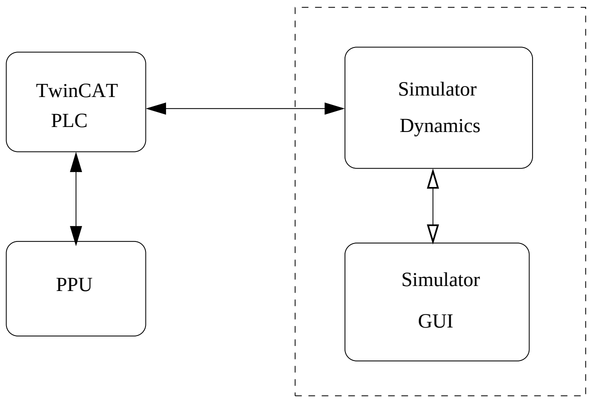
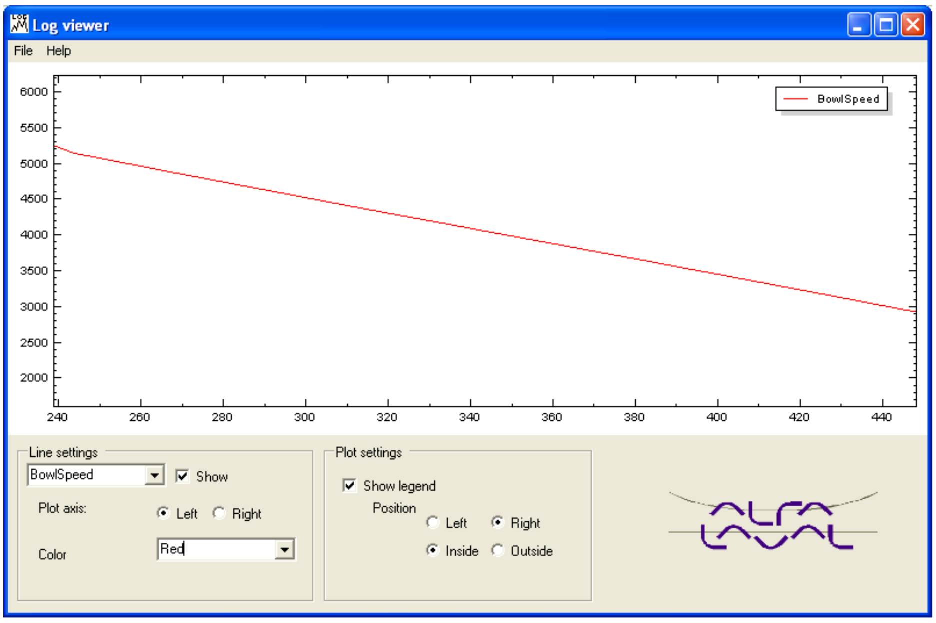


Figure 9.3 Separator system with simulator.



Line settings
BowlSpeed [✓ Show]
Plot axis: [✓ Left] Right
Color Red

Plot settings
[✓ Show legend]
Position
Left [✓ Right]
[✓ Inside] Outside

Figure 9.4 Screen shot of Log View.

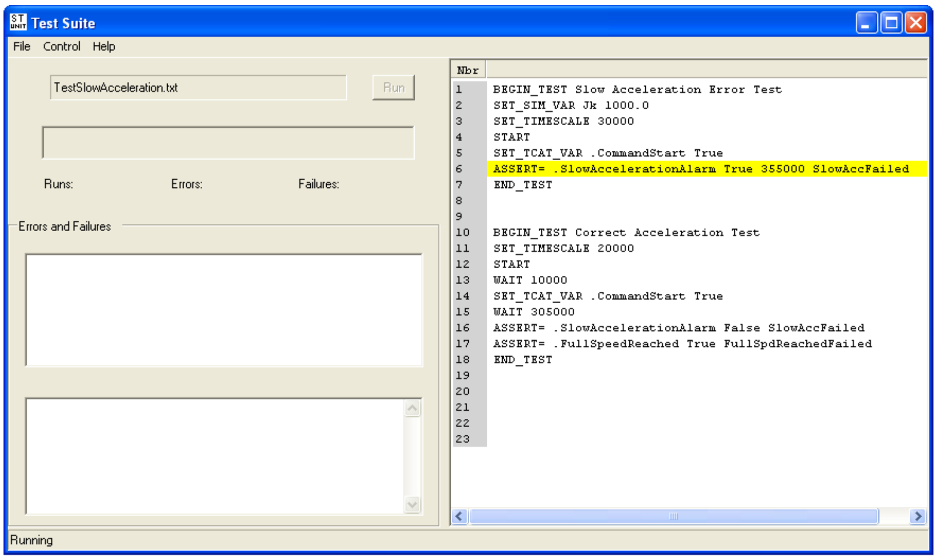
Test Suite

Test Suite has borrowed both idea and looks from the unit testing tool JUnit (Object Mentor Inc. 2005). JUnit is a unit testing tool for Java programs where tests are written in a separate class. JUnit test methods by using *Assertments*. An assertment calls a method and tests if returned data is equal to expected data. JUnit is an implementation of a framework called xUnit where 'x' is substituted by an abbreviation of the language being tested.

Test Suite is not an official xUnit implementation as it differs in many ways. One distinction between an official xUnit implementation and Test Suite is that in Test Suite test cases are written in a specialized script language instead of the language in which modules are being tested, in this case IEC 61131-3. The tool is used to test code written and executed in TwinCAT and uses the simulator previously introduced.

The *assert* command in Test Suite tests if a TwinCAT variable has the expected value. It is possible to test if a variable is less than, equal to or larger than an expected value either directly or within a given time interval. Both numerical and boolean variables can be tested. Each test can have three possible results. It can succeed, fail or produce an error. The test succeeds when the tested variable fulfills its requirement, it consequently fails when it does not. If the variable does not exist or if there is something wrong in the semantics of the test case an error will arise.

Test Suite is shown in Figure 9.5. The test cases appear to the right, control and results to the left. If the horizontal bar to the left is green after all tests have run, all tests succeeded. The bar turns red in case of failures or errors. The goal is to write test cases testing all possible situations in the simulator. The tests are written according to the script language presented in Figure 9.6 and an example of a test case together with explaining comments is shown in Figure 9.7. An overview of the separator system with Test Suite is shown in Figure 9.8.



TestSlowAcceleration.txt

```
BEGIN_TEST Slow Acceleration Error Test
SET_SIM_VAR Jk 1000.0
SET_TIMESCALE 30000
START
SET_TCAT_VAR .CommandStart True
ASSERT= .SlowAccelerationAlarm True 355000 SlowAccFailed
END_TEST


BEGIN_TEST Correct Acceleration Test
SET_TIMESCALE 20000
START
WAIT 10000
SET_TCAT_VAR .CommandStart True
WAIT 305000
ASSERT= .SlowAccelerationAlarm False SlowAccFailed
ASSERT= .FullSpeedReached True FullSpdReachedFailed
END_TEST
```

Figure 9.5 Screen shot of Test Suite.

Tests equality
`ASSERT= <TCAT var_name> <value> [timeout] <comment>`
Tests larger than
`ASSERT> <TCAT var_name> <value> [timeout] <comment>`
Tests less than
`ASSERT< <TCAT var_name> <value> [timeout] <comment>`
Sets variable in TwinCAT
`SET_TCAT <TCAT var_name> <value>`
Sets variable in Simulator
`SET_SIM <Sim var_name> <value>`
Sets time scale
`SET_TIMESCALE <value>`
Pauses script execution
`WAIT <time>`
Starts TwinCAT and Simulator
`START`
Marks beginning of test case
`BEGIN_TEST [test name]`
Marks end of test case
`END_TEST`

Figure 9.6 Available commands in Test Suite.

9.2 Implementation Details

The simulator suite consists of three programs called Simulator, Log View and Test Suite. Simulator and Test Suite share a simulator logic module. It is the logic module that contains the separator dynamics and performs the simulation. The authors have followed a naming convention where different prefix say what type of class it is. The prefix 'f' means that the class is a form, 'uc' that the class is a user control and 'I' that the class is an interface.

Simulator Logic

The core of the simulator suite is the logic module. Except from the separator dynamics the logic module also contains a communication class and a logger class. A class diagram of the logic module can be found in Figure 9.9.

```
% Marks the start of the test and gives the test a name
BEGIN_TEST Slow Acceleration Error Test
% Sets abnormally high bowl moment of inertia
% causing the bowl to accelerate slower
SET_SIM_VAR Jk 1000.0
% Makes the simulation go 20 times faster
% than real time
SET_TIMESCALE 20000
% Starts the simulator and TwinCAT
START
% Pushes the 'start' button
SET_TCAT_VAR .CommandStart True
% Tests if the correct alarm occurs
% within 355 seconds, otherwise an error
% message will be shown
ASSERT= .SlowAccelerationAlarm True 355000 Slow Acc Test Failed
% Marks the end of the test case
END_TEST
```

Figure 9.7 Example of a test case.

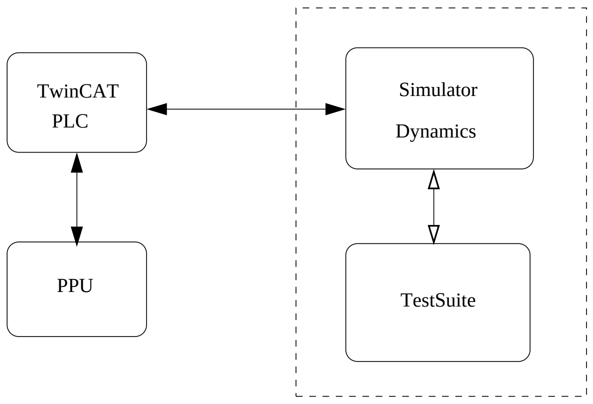


Figure 9.8 Separator system with Test Suite.

`SeparatorLogic` This class is intended to be used as a base class for all logic classes representing different types of separators. It contains methods and variables that are general for all separator types.

`SeparatorLogicClara80` This class inherits SeparatorLogic and does all the Clara80 specific calculations. It contains a thread with a period of 100ms that performs the calculations. Results from the calculations are sent by raising events. It has an inner class with all required parameters and corresponding values.

`ADSCommServer` A communication class that uses the ADS component provided by Beckhoff to communicate with TwinCAT. Variables are read from TwinCAT "On change" and this class registers and stores which variables are to be read. When a variable is changed this class raises an event with the updated variable. This class receives events raised by the logic class and writes them to TwinCAT using a synchronized method in the ADS component.

`XMLLogger` This class is responsible for writing the log-files. It has a thread with a period of 500ms that write chosen variable values to an XML-file.

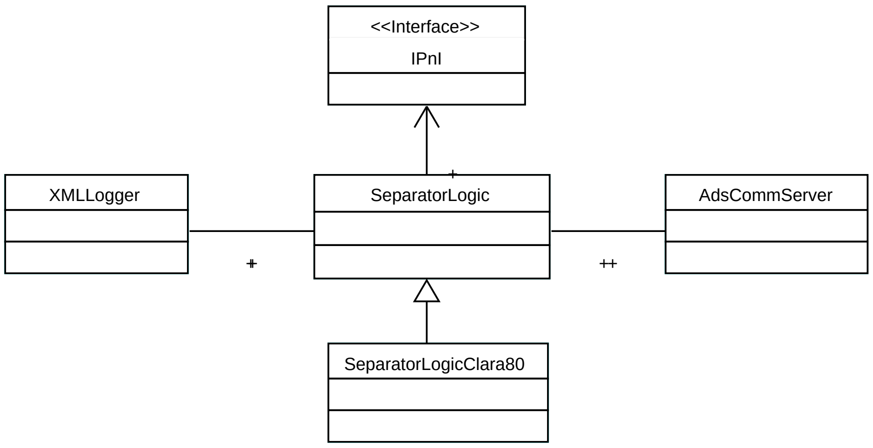


Figure 9.9 Class diagram of the logic module.

`IPnI` `IPnI` is an interface for a PnI class which is a graphical representation of the separator. The interface makes it possible to write different PnIs to the simulator and also to place the logic and the graphical interface in different packages.

Simulator GUI

The simulator GUI is used to control the simulator manually, together with the logic module it constitutes the Simulator program. A class diagram of the simulator GUI can be found in Figure 9.10.

`IucComponent` All user controls included in an implementation of the `IPnI` interface implements this interface.

`ucPnIClara80` This class implements `IPnI` and represents a graphical overview of the Clara 80 separator. This is where all `IucComponents` are placed. It catches raised events from the `SeparatorLogicClara80` and relays them to their corresponding user control.

`ucValve` This class implements `IucComponent`. It contains a boolean that is false when the valve is closed and true when it is open. The user control is colored red or green depending on the boolean.

`ucPump` This class implements `IucComponent`. It contains a boolean that is false when the pump is off and true when it is on. The user control is colored red or green depending on the boolean.

`ucSeparatorClara80` This class implements `IucComponent`. It displays the motor and bowl speed for a Clara 80 separator.

`ucLeakage` This class implements `IucComponent`. It contains a boolean that is false when the user control should indicate a leak. A leak is indicated by painting the user control red.

`ucFlowMeter` This class implements `IucComponent`. It displays the current flow through the separator.

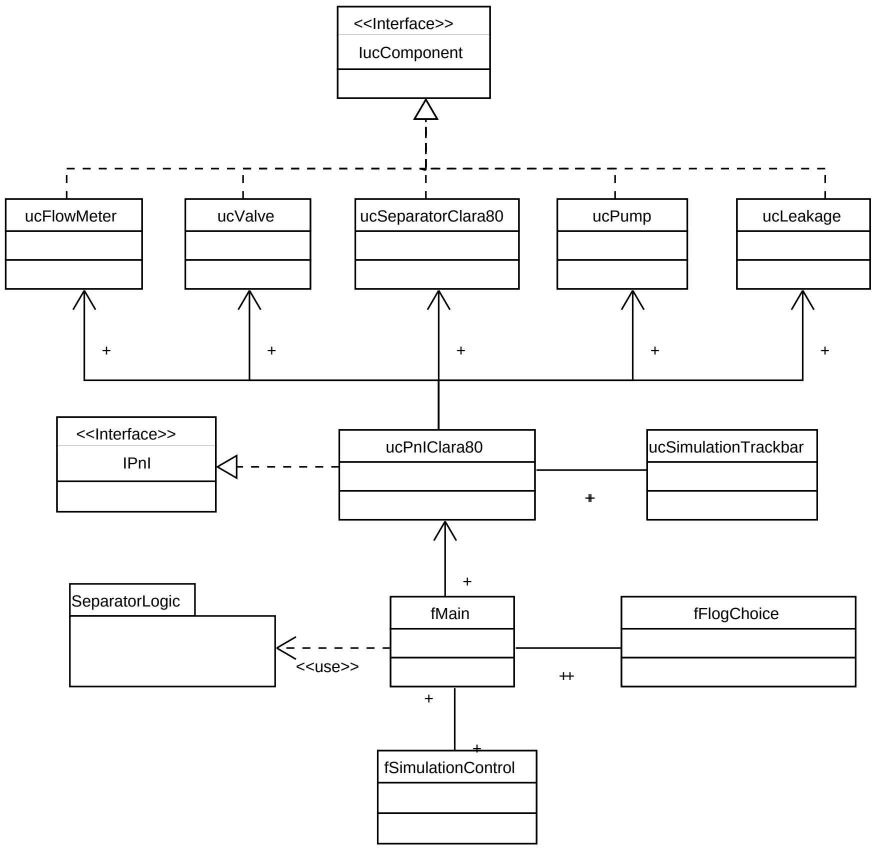


Figure 9.10 Class diagram of the simulator GUI.

`ucSimulationTrackbar` A customized implementation of a trackbar. It contains a slider where the first half contain steps from 10% to 100% and the second half contain steps from 100% to 2000%.

`fLogChoice` Contains a list of available variables that can be logged by checking a checkbox next to the variable name. The list is opened from the menu in `fMain` and sends the chosen variables to `XMLLogger`.

`fSimulationControl` This is the form for the simulation control. It contains buttons and numeric updowns that change parameters in the simulator logic.

`fMain` This is the main form for the simulator GUI. It contains `ucPnIClara80` and `fSimulationControl`. From its menus it is possible to connect the simulator to TwinCAT, start and stop the simulator, choose variables to log and start and stop the logger.

Log View

Log View works as a wrapper around the open source plotter NPlot and makes it possible to choose which variables to plot, change colors, change axis, display legend and zoom. A class diagram of Log View can be found in Figure 9.11.

`fPlotView` This the main form for Log View which contains the NPlot component and controls to modify the plots.

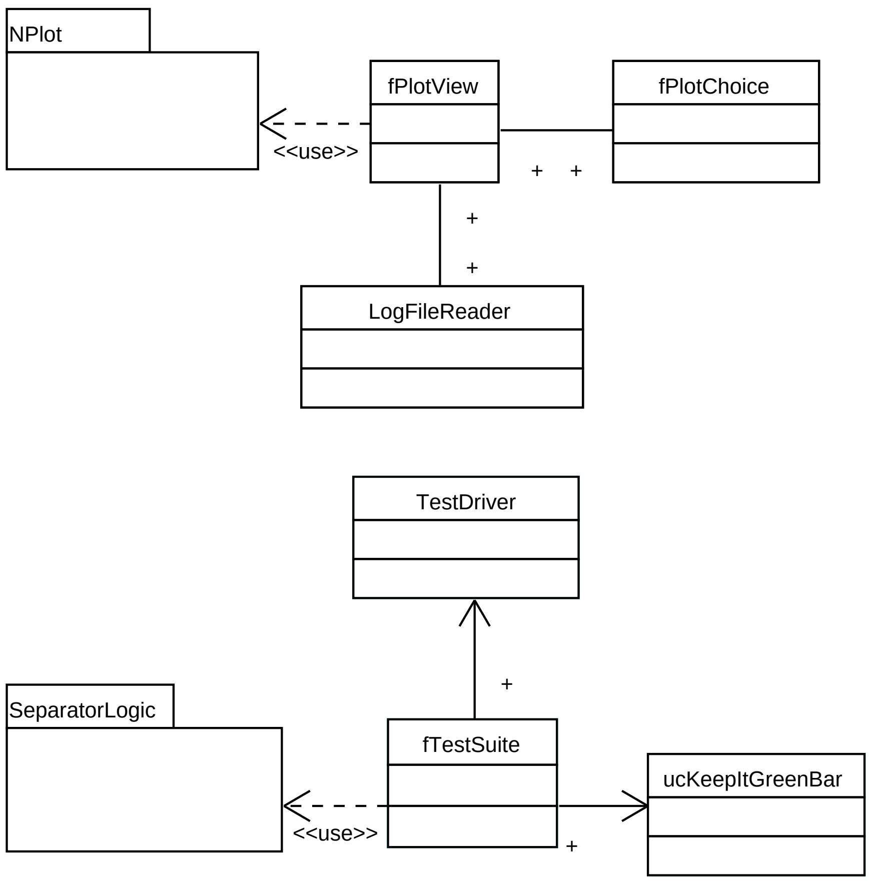


Figure 9.11 Class diagram of both Log View and Test Suite.

`fPlotChoice` Displays a list of available variables found in a log-file. The list is automatically displayed when a log-file is opened from the menu in the main form.

`LogFileReader` This class is responsible for parsing log-files. It extracts information about the logged variables from the log-file to the main form.

Test Suite

Test Suite opens and executes tests written in pre-defined text-files. It runs the tests using the logic module. A class diagram of Test Suite can be found in Figure 9.11.

`fTestSuite` This is the main form for the Test Suite which displays the contents of an opened file to the right and the results of run tests to the left. It also contains a `ucKeepItGreenBar` which is colored green if all tests passed and red otherwise.

`TestDriver` This file runs the actual tests and return their results to the main form. It connects both to the logic module and TwinCAT. The tests are run in a separate thread.

`ucKeepItGreenBar` This is a user control which resembles a progress bar. It is painted in steps where each step represent a run test. As long as no test have failed it is colored green otherwise red.

9.3 Solving the Problem of Variable Speed

Is variable speed really a problem? Yes, it turned out to be a great problem. The reasons for this are many. To begin with PLC programs are not normally suited for dynamical changes in the run conditions. TwinCAT uses predetermined task times that cannot be changed on-line. Furthermore, the time used by TwinCAT timers is directly connected to the system clock. This makes it impossible to change TwinCAT's perception of time without altering the code.

Even for a program created in C# variable time proves non-trivial to implement. The main reason for this is the limited resolution of the available system time. The approximate resolution in Windows NT/XP is $10ms$ (Microsoft Corp. 2005), far lower than needed in demanding real-time simulations.

Solution in C#

The problem of resolution has a simple solution. Using a special type of timer from `kernel32.dll` called `QueryPerformanceCounter`, a much higher resolution ($< 1ms$) is achieved. Two drawbacks are that this solution introduces a larger overhead and that the timer is not as easy to use as the original timer. In the case of the simulator, the period can be as low as $5ms$ which motivates use of the high resolution timer.

Using a time scale, the program calculates passed simulation time since last iteration to be used in the calculations based on actual time from `QueryPerformanceCounter`. This is enough to change the simulation speed but it introduces a new problem, numerical accuracy. If the iteration period is kept constant, simulated time will vary greatly under different time scales giving different exactness in the calculations.

The solution we propose is based on the idea that the simulator spends most of the time waiting for a new iteration to begin. Variable simulation speed is then improved by changing the period based on current time scale. If the simulation is sped up, the period is shortened. This way the average simulation period is kept constant maintaining numerical accuracy in integrations. Figure 9.12 shows the outline of the solution.

Solution in TwinCAT

The ideal solution to variable simulation speed in TwinCAT would be to change the "speed" of the TwinCAT system clock and thereby change the rate of all timers at the same time. This is however not possible by reasons described above.

It is possible to scale the timeout in a timer with a time scale, this would be a feasible solution as long as the time scale is not changed during a simulation. If the time scale changes, a timer that has come half the way to the timeout could be halted prematurely.

Our proposed solution to these problems is a combination of using an internal time which is incremented in each iteration using a time scale factor that can be changed. New timers replace the original ones by using the internal simulation time instead of the TwinCAT system clock. By using the internal simulation time no timers will be prematurely halted. The solution in ST code is outlined in Figure 9.13 and 9.14.

```
protected void run() {
  while(true) {
// Sets wake time
wakeTime = getPerformanceTime();

    // Performs simulation calculations
    // (and outputs new data if necessary)
    performCalculations(wakeTime);

    // Calculate new sleep until time
    sleepUntilTime += basePeriod/timeScale;
    sleepTime = sleepUntilTime - getPerformanceTime();
    if(sleepTime > 0) {
      try
        Thread.Sleep(sleepTime);
      catch(ThreadInterruptedException)
        sleepUntilTime = getPerformanceTime();
    }
  }
}
```

Figure 9.12 Variable simulation time in C#.

```
VAR
   simTime : TIME;
END_VAR

(*Divisor depends on task time, divisor=1000/(task time in ms)*)
simTime := simTime + REAL_TO_TIME(TimeScale/100);
```

Figure 9.13 Implementation of variable time scale in TwinCAT.

The new timers have the same interface as the original ones which make it almost effortless to change between them. In fact, a textual search-and-replace is sufficient. It might even be possible to keep the simulation timers when running the code with the real system. It is important for the reliability of the code testing that the TwinCAT code is altered as little as possible.

9.4 Evaluation of the Simulator

Comparisons with tests performed on the real Clara 80 machine show that the simulator is generally well modeled and implemented. Comparison plots of start and stop of the simulation can be seen in Figure 9.15 and Figure 9.16. Plots of some courses of events, such as ”discharge” (Figure 9.17) and ”flow on” (Figure 9.18), show that the simulator gives the correct behavior but needs to be adjusted. When evaluating ”discharge” and ”flow on” it is important to have in mind that no detailed data about the real Clara 80 tests except for motor speed and flow were known when making the comparisons. Discharged volume and back pressure, two important parameters when computing load and bowl speed drop, was unknown.

In some extreme cases there is no measured data to compare results with, making an evaluation fairly subjective. Furthermore, no exact models exist for extreme states, for example when the separator is leaking. Leakage increases the air density around the bowl, drastically increasing the torque consumed.

```
FUNCTION_BLOCK SimulationTimer
VAR_INPUT
   IN : BOOL; (*Starts timer with rising edge, resets with falling*)
   PT : TIME; (*Time to pass before Q is set*)
END_VAR
VAR_OUTPUT
   Q  : BOOL; (*TRUE PT sec after IN had rising edge*)
   ET : TIME; (*Elapsed time*)
END_VAR
VAR
   startTime : TIME;(*Time when timer started*)
   M         : BOOL;(*Last value of IN*)
END_VAR

IF (NOT M) AND IN THEN (*Start timer*)
   startTime := SimulationCounter.simTime;
   M := TRUE;
ELSIF M AND (NOT IN) THEN (*Reset timer*)
   M := FALSE;
   Q := FALSE;
   ET := T#0s;
ELSIF M AND IN THEN (*Timer running*)
   ET := SimulationCounter.simTime - startTime;
   IF ET >= PT THEN (*Timer finished*)
      Q := TRUE;
      ET := PT;
   END_IF
END_IF
```

Figure 9.14 Implementation of Timer in TwinCAT.

Currently this behavior is modeled only in a simplified way.

9.5 Evaluating Test Suite

To evaluate Test Suite, three different test scripts were written. Two testing erroneous behavior and one testing normal behavior. The tests show that it is possible to write automated tests. To evaluate whether the Test Suite is useful when testing code or not, the program needs to be used in a real project.

The problem is perhaps not *if* the Test Suite can test code automatically, but how and by whom the tests should be written. It might be possible to produce a collection of tests that all control code should pass. This could make the tests more standardized and would improve their usability.

9.6 How to move on

If the model turns more complex by introducing more differential equations, a transfer of the simulator from C# to a tool such as Dymola might have to be considered. As the authors have limited knowledge about Modelica and Dymola, it is hard to say how difficult a transition would be. It is however clear that work is needed in order to make a simple interface between Dymola and TwinCAT. It is also hard to say exactly when a transition should be made,

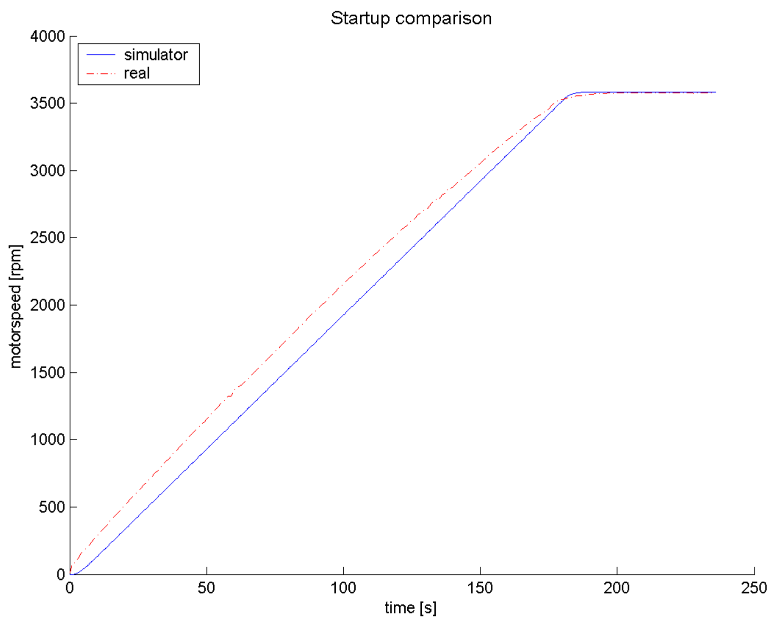


Figure 9.15 Plot of start comparison.

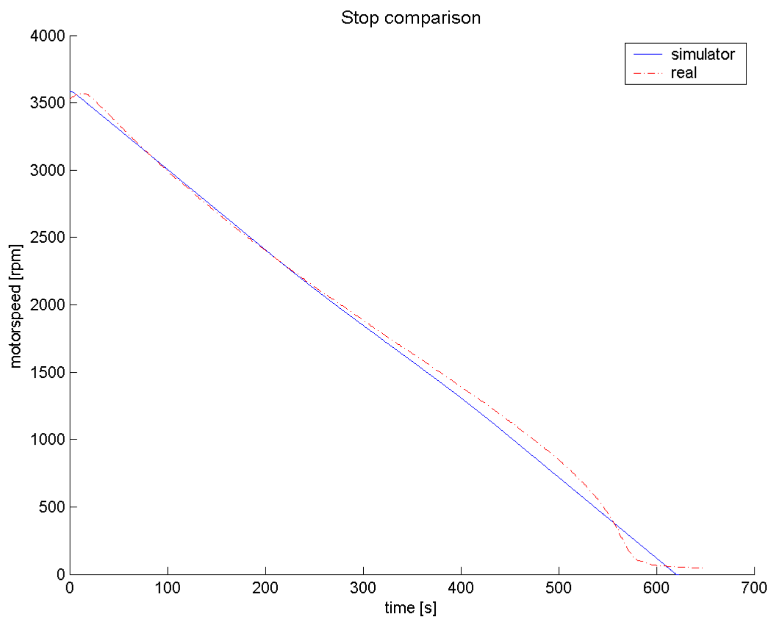


Figure 9.16 Plot of stop comparison.

for example if the model is extended with more differential equations with different time constants a transition would probably be necessary.

One of the benefits of using a tool like Dymola is that it can be easier to construct a library of models. It might be possible to break up the separator in several parts that can be modeled individually making it possible to construct new separators by combining existing units. In a longer perspective, using Dymola should make implementation of new models easier as new models can

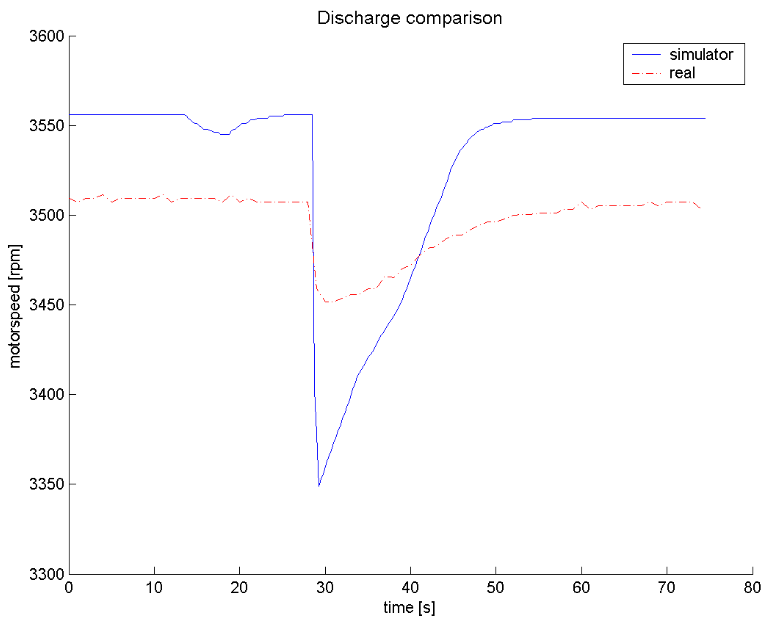


Figure 9.17 Plot of discharge comparison.

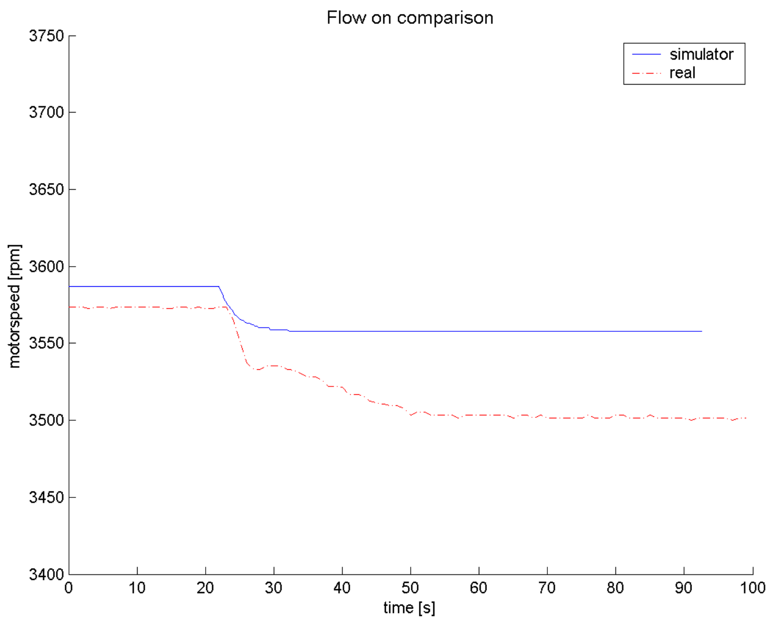


Figure 9.18 Plot of flow on comparison.

be constructed graphically by combining existing blocks. Development of new blocks may however be cumbersome, especially if a high level of configurability is needed.

Freiseisen et al. (2002) aims to make the conversion automatic by converting CAD-drawings to Modelica-code. This is motivated by "PLC programmers should not be bothered with simulation models" (Freiseisen et al. 2002, p. 195). The Modelica code is further converted via an XML-scheme to C++-

code. This is one possible solution to enable simple model configuration, but at Alfa Laval the CAD-drawings are highly complex and much work would be needed to convert them to Modelica-code.

Depending on how and by whom PLC code will be written in the future, the level of automatic configuration can be adjusted. It is also important to focus on the use of the simulator, if it is constructed mainly to test PLC-code a higher level of flexibility can be favored over a high level of simulation accuracy. However, if the simulator will be given other responsibilities, high simulation accuracy may need prioritization.

9.7 Further Work

This thesis has not discussed if the simulator is of any help when producing PLC-code. The claims that automated tests can save both time and money while improving the quality of the code cannot be tested without a new study taking place. Testing the simulator in a near-real situation is vital in order to make any conclusions of needed improvements or change of platform.

Automated tests have proved successful in other areas of software engineering, but it is yet to be seen how the development process at Alfa Laval fits or can be re-fit to enable convenient testing of PLC-code.

10. Summary

The main purpose of this master's thesis was to construct a simulator for testing IEC 61131-3 code for separators. The separator code is written in Structured Text and executed in a soft PLC called TwinCAT. The proposed solution consists of a hardware simulator written in C# which communicates with TwinCAT using ADS, a TwinCAT specific communication protocol.

The modeling of the separator was performed in cooperation with employees at Alfa Laval. The final model consists of a continuous model and a discrete event model which together form a hybrid model. The model is based on the idea of production and consumption of energy in the separator system which is expressed in a differential equation. Energy is added to the system by the motor and consumed by the separator bowl. The discrete event model controls the contents of the differential equation. Typical discrete events are opening and closing of valves in the separator.

The simulator executes the model and solves the differential equation numerically using the Euler method. It is possible to run the simulator in two modes, either manually with a user interface or automatically by executing pre-defined test files. The purpose of the test files is to enable automatic testing of IEC 61131-3 code executed in TwinCAT. The files are written in a script language defined by the authors giving the tester the possibility to test if a variable in TwinCAT has a certain value.

Comparisons with the real separator have been made, showing that the simulator performs well enough to enable testing of code. However, it is important to remember that the simulator is only a tool. To make full use of the tool, efficient tests must be written that thoroughly test all relevant functionality of the IEC 61131-3 code in TwinCAT. Writing tests is the most extensive part of testing. Three test files have been written by the authors and executed successfully on the simulator.

Testing has long been an important factor in software engineering, in large software companies thousands of test specialists are employed for the sole purpose of testing. While testing normally is conducted with advanced tools, testing in a PLC environment has long been a strenuous manual process responsible for unnecessarily large portions of total costs in projects. It is now time for PLC programming to step into the 21^{st} century.

11. Bibliography

Alfa Laval (1994), ‘G force in orbit’, *Alfa Laval Utbildningsmaterial* p. 4.

Alfa Laval (n.d.), ‘Separation priciples and features’, Internal trainee work book.

Beck, K. (2000), *Extreme programming explained*, Addison-Wesley.

Beckhoff Gmbh (2001), *TwinCAT System Overview.*
*ftp://ftp.beckhoff.com/Document/Software/TwinCAT/1033/

Beckhoff Gmbh (2002), *TwinCAT ADS AMS Specification*, Eisenstraße 5, D-33415 Verl.
*ftp://ftp.beckhoff.com/Document/Software/TwinCAT/1033/

Beckhoff Gmbh (2004*a*), *TwinCAT ADS*, Eisenstraße 5, D-33415 Verl.
*ftp://ftp.beckhoff.com/Document/Software/TwinCAT/1033/

Beckhoff Gmbh (2004*b*), *TwinCAT OPC Server 4*, Eisenstraße 5, D-33415 Verl.
*ftp://ftp.beckhoff.com/Document/Software/TwinCAT/1033/

Beckhoff Gmbh (2005). Accessed 7 February 2005.
*http://www.beckhoff.com

Björklund, M. & Paulsson, U. (2003), *Seminarieboken - att skriva, presentera och opponera*, Studenlitteratur, Lund.

Dynasim AB (2005), ‘Dymola’. Accessed 7 February 2005.
*http://www.dynasim.se

Eclipse Foundation (2005), ‘Eclipse’. Accessed 7 February 2005.
*http://www.eclipse.org

Freiseisen, W., Keber, R., Modetz, W., Pau, P. & Stelzmueller, D. (2002), Using `Modelica` for testing embedded systems, *in* ‘2nd International Modelica Conference’, Oberpfaffenhofen, pp. 195–201.

Graham, D. (1994), ‘Testing’, *Encyclopedia of Software engineering* .

Hassapis, G. (2000), ‘Soft-testing of industrial control systems programmed in iec 1131-3 languages’, *ISA Transactions* **39**, 345–355.

Hayes Control Systems (2001), ‘PLC on the PC with Beckhoff TwinCAT system’. Accessed 7 February 2005.
*http://www.engineeringtalk.com/news/hay/hay117.html

Hendricks, H. (1996), *The history of the PLC*, Dick Morley, R. Morley Incorporated 614 Nashua Street, Suite 56 Milford, NH 03055-4992 USA.
*http://www.barn.org/FILES/historyofplc.html

Hetzel, B. (1988), *The complete guide to software testing*, Wiley.

Holme, I. M. & Solvang, B. K. (1997), *Forskningsmetodik*, Studentlitteratur, Lund.

Howlett, M. (2005), ‘Nplot’. Accessed 10 February 2005.
*http://netcontrol.org/nplot

Jeffries, R., Anderson, A. & Hendrickson, C. (2001), *Extreme programming installed*, Addison-Wesley.

John, K.-H. & Tiegelkamp, M. (2001), *IEC 61131-3: Programming Industrial Automation Systems*, Springer-Verlag, Berlin.

Langnau, L. (1995), 'A step closer to easier plc programming', *Material Handling Engineering* p. 23.

Li, R. & Winitsky, L. (1999), A virtual rolling mill for real time control system tuning, operator training and process simulation, IEEE, pp. 592–598.

MathWorks Inc. (2005), 'Matlab/Simulink'. Accessed 7 February 2005.
*http://www.matlab.com

Microsoft Corp. (2005), 'C# .Net'. Accessed 7 February 2005.
*http://msdn.microsoft.com

Object Mentor Inc. (2005), 'Junit'. Accessed 23 February 2005.
*http://www.junit.org

Olsson, G. & Rosén, C. (2004), *Industrial automation*, Lund University, Lund.

OPC Foundation (2005). Accessed 7 February 2005.
*http://www.opcfoundation.org

Patel, R. & Danielsson, B. (1991), *Forskningsmetodikens grunder : att planera, genomföra och rapportera en undersökning*, Studentlitteratur, Lund.

PLCopen (2005), 'PLCopen'. Accessed 18 February 2005.
*http://www.plcopen.org

PLCS.net (2005). Accessed 7 February 2005.
*http://www.plcs.net

Shanthikumar, G. J. (1983), 'A unifying view of hybrid simulation/analytic models and modeling', *Operations Research* **31**, 1030–1052.

Sun Microsystems Inc. (2005), 'Java'. Accessed 7 February 2005.
*http://java.sun.com

Visual Solutions Inc. (2005), 'VisSim'. Accessed 7 February 2005.
*http://www.vissim.com

Wallén, G. (1996), *Forskningsteori och forskningsmetodik*, Studentlitteratur, Lund.

Zeigler, B. P., Kim, T. G. & Praehofer, H. (2000), *Theory of modeling and simulation : Integrating discrete event and continuos complex dynamic systems*, Academic Press.