Maintenance Free and Sustainable High-Level Control in Cement and Mining Industry

Hansen, Ole Fink; Andersen, Nils Axel; Recke, Bodil; Recke, Bodil Olesen; Ravn, Ole

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Technical University of Denmark

DTU

Ole Fink Hansen

Maintenance Free and Sustainable High-Level Control in Cement and Mining Industry

PhD thesis, December 2008

DTU Electrical Engineering
Department of Electrical Engineering

Maintenance Free and Sustainable High-Level Control in Cement and Mining Industry

Ole Fink Hansen

Ph.D. thesis
December 2008

This document was typeset with LaTeX2e using 10p. Nimbus Sans

Preface

This thesis is submitted in partial fulfillment of the requirements for the Industrial Ph.D. program (Danish: ErhvervsPhD). The project was conducted by Ole Fink Hansen from 2005 to 2008 in collaboration between Department of Electrical Engineering, Technical University of Denmark and FLSmidth Automation. University supervisors were associate Professor Nils Axel Andersen and associate Professor Ole Ravn. The company supervisors were Bodil Recke, Jørgen K.H. Knudsen and Hassan Yazdi.

Acknowledgements

First of all I would like to thank the Industrial PhD Programme Committee and FLSmidth Automation for making this project possible.

I would like to express my gratitude to both university and company supervisors. Jørgen Knudsen who initiated the project, has been an invaluable provider of insight and support especially in the business related aspects of the industry and the project. Furthermore he has arranged visits to various plants and conferences which have given invaluable insight.

Bodil Recke has shown endless patience, support and technical insight regarding high-level control systems. Hassen Yazdi has, like the other supervisors, supplemented with personal advice regarding the process of conducting a Ph.D. project.

At the university, Nils Andersen and Ole Ravn have always had their door open and provided essential help in structuring the project and the thesis.

The co-workers at FLSmidth Automation represent an enormous knowledge base and have shown great interest in the project, supplied anecdotal evidence, suggestions and feedback. My thanks go also to the people in the Danish process control community in industry as well as universities for good discussions and feedback.

Finally I would like to thank friends and family who have been neglected during times of deadlock and intensive workload.

Abstract

High-level control systems have been utilized in the process industry for decades, and also in cement production their use is well established. In comparison to manual control their ability to increase production and quality of end product, while reducing energy consumption and emission, is well recognized. Therefore, the payback time may be less than one year. It is common however, that the systems are disabled only a few months after commissioning because the process has changed in such a way that it does no longer matching the systems' tuning.

The cause of this can be raw materials changing, wear of machinery, and reconstruction of the plant etc. Therefore, in order to keep a constant, high performance, the high-level control system requires regular maintenance by means of expert personnel readjusting and modifying the algorithm, which is resource demanding.

The aim of this Ph.D. project is to minimize or eliminate the amount of resources needed to keep a high performance.

Current high-level control algorithms are sophisticated and complex software. An analysis of such algorithms shows that only 10% of the source code can be considered implementation of control theory. The remaining 90% handles other tasks but nevertheless still require maintenance.

For the 10% of the algorithm that is control related, the maintenance issue is to some extent addressed by research topics such as adaptive control, which aim at retuning the parameters of the algorithm to match the changing process. In this project however, it has been chosen to focus on the remaining 90% of the algorithm which still require manual modifications to cope with a changed process. Although this issue has gained limited attention from academia so far it is well recognized by the industry.

In the process of maintaining an algorithm it has turned out that navigating the source code, understanding the interaction of signals and tracking down the statement in the code responsible for the problem is the issues which require expert knowledge and they are very time consuming. In contrast, it is relatively simple to conduct the actual modification once the few statements to be modified have been found.

Current SCADA systems allow logging of signals while the control system is running which may be useful in the diagnostic process. However, each signal to be logged must be explicitly specified, and typically only measurements, setpoints and key performance indicators are therefore logged. Thus, critical algorithm-related information is not available post-mortem.

A number of tools and methods has therefore been developed which aim firstly at monitoring a running algorithm to raise an alarm in case the algorithm starts to

behave abnormally. Secondly, a set of tools and methods are proposed to help the human expert to diagnose and locate the part of the algorithm that is responsible for the malfunction. These steps are based on information extracted from the algorithm at runtime by means of a technique called program slicing.

Thus, instead of monitoring the process to detect changes, attention is focused on the control algorithm itself. In addition to the signals already logged by the SCADA system, enough data is collected at each execution of the control algorithm so an exact replay of its execution can be performed later. The resolution of this replay is down to single-stepping through the lines of source code, keeping track of variable values from the execution of one statement to the other.

The method has been tested on a full scale control algorithm, showing that computational cost at runtime is neglectable and that the amount of additional data to be logged is compatible with the storage capacity of current computers. Preliminary statistical analysis of the logged data shows that normal and abnormal behavior of a real-world control algorithm can be distinguished so an alarm can be raised.

The method enables backtracking of signal dependencies in the algorithm which can be used to semi-automatically guide the human expert to the responsible part of the algorithm. This feature is demonstrated in a simple control algorithm.

Dansk Resumé

Højniveau reguleringssystemer har været anvendt i processindustrien i årtier og udgør et veletableret marked også i cement produktion. Tilføjelsen af et sådant reguleringssystem medfører øget produktivitet, nedsat energiforbrug og forbedret kvalitet af slutproduktet så tilbagebetalingen typisk er under ét år. Desværre sker det ofte at systemet bliver taget ud af brug, få måneder efter installation.

Årsagen hertil er mangeartet og skyldes grundlæggende at miljøet omkring systemet ændrer sig så processen som systemet er designet til at regulere ikke længere er den samme. Det kan f.eks. være råmaterialer der ændrer sig, slidtage af maskiner osv. For at opretholde konstant, høj ydelse er det derfor nødvendigt at systemt jævnligt vedligeholdes og tilpasses de ændrede forhold hvilket er ressourcekrævende.

Formålet med nærværende PhD projekt er at minimere eller helt eliminere de resourcer det kræver at opretholde høj ydelse.

Moderne højniveau regulerings-algoritmer er omfattende og kompliceret software. Analyse af sådanne algoritmer viser at kun 10% af kildekoden kan betegnes som implementeret reguleringsteknik - de resterende 90% er kode som varetager andre opgaver men som dog stadig kræver vedligeholdelse.

Hvad angår de 10% af algoritmen som er reguleringsrelateret findes etablerede forskningsområder så som adaptiv regulering som søger at tilpasse algoritmens parametre til ændringer i processen. I dette projekt er der derimod valgt at fokusere på de resterende 90% af algoritmen som kræver manuel tilpasning til nye forhold - et problem som er anerkendt i industrien.

I forbindelse med manuel tilpasning af en reguleringsalgoritme har det vist sig at kræve ekspertindsigt og være meget tidskrævende at navigere rundt i kilde koden, forstå sammenhænge mellem signaler og finde de få linier kode som skal ændres. At udføre ændringen derimod er relativt simpelt.

Der er derfor udviklet en række koncepter og værktøjer med henblik på først at overvåge algoritmen under kørsel og detektere og alarmere i tilfælde af at algoritmen opfører sig unormalt. Dernæst hjælpes brugeren (vedligeholdelses-personale) til at lokalisere de få, relevante linier kode som er skyld i problemet, uden at det er nødvendigt at undersøge omkringliggende dele af algoritmen.

I stedet for at overvåge den regulerede proces rettes fokus således imod reguleringsalgoritmen selv.

Eksisterende SCADA systemer tillader i begrænset omfang logning af signaler hvilket er til gavn i forbindelse med vedligehold. Dog skal det konfigureres eksplicit hvilke signaler der skal logges, og interne variable i reguleringsalgoritmen er ikke tilgængelige. Derfor er det typisk kun måling og set-punkter der logges. Informatio-

ner om reguleringsalgoritmens opførsen er derfor ikke tilgængelig f.eks. når det skal undersøges hvorfor algoritmen gjorde som den gjorde for en uge siden.

Med de foreslåede løsninger bliver der logget tilstrækkelig information til at algoritmens nøjagtige eksekveringsforløb kan genskabes kode-linje for kode-linje ned til værdien af hver enkelt variabel.

Contents

Chapter 1

Introduction

It is Monday morning at Elbonia Cement Industries and plant manager John routinely opens his email to check the weekly, auto-generated report from the SCADA- and control system. Besides the usual graphs and tables showing production trends and emission indexes etc., the system has this time included a new and emphasized section which he has never seen before. It tells that the control system itself, and not just the physical plant has somehow changed its behavior. A closer look at the tonnage shows a slight drop since the previous week, and the key performance indicators together with the environmental section in the report are not as tidy as they use to be. Overall there is nothing intimidating, like reduced availability of the machinery, but just a minor indication that something is afoot.

John have chosen not to maintain any expert control system knowledge on site, so he clicks a link in the report to request service from the manufacturer who commissioned the system last year. Less than half a cup of coffee later, Mrs. Jensen from the manufacturer is on the phone, asking for permission to access the system remotely...

At her desk in Copenhagen, Mrs. Jensen is confident that she can resolve the issue, although she is not quite an expert in the kind of control algorithm running at Elbonia Cement. A vague memory tells her that Mr. Nielsen who installed and customized the system had to make some rather extensive last minute modifications to the algorithm to make it cope with the special conditions on site. Unfortunately he is on vacation and is unavailable.

The new tracing tool tells her that, while the system used to alter slowly between a handful of states, it switched to a rapid cycle between 3 new states about a week ago. A single mouse click leads her to the few lines of code to blame for the cycle. Since it is only a few lines, she understands their purpose rather quickly. The remaining thousands of lines of source code comprising the entire algorithm she does not know, and she does not care. The tool has pin pointed the piece of code that needs attention, and that is all she has to know for now.

Another mouse click brings up statistical information about the single signal responsible for this particular branching in the algorithm. It shows that the kiln temperature is fluctuating a little more than usual, just enough to trigger some threshold that causes the controller to switch to a different mode. It turns out that the latest shipment of alternative fuel the plant got is more heterogeneous in terms of calorific value and a brief chat with John ensures her that the increased fluctuation is acceptable considering the attractive price tag of the new fuel. So, only a few thresholds need adjustment to tolerate the new fuel characteristics.

Before noon the system is back at stable, high performance and John is happy. The control system issue was identified and resolved before it ever evolved to become a problem and he is now confident he will reached the targeted production and plant availability which the management have set for him this quarter. He heads to the control room to inform Joe, who just started his watch on the day shift, but hesitates... In fact, there is no reason to disturb him at all.

1.1 The Situation Today

This short story highlights some of the improvements in the maintenance of industrial control systems that have been the objective of the work in this project.

Our imaginary plant is able to detect abnormal behavior and raises a very early warning of a control algorithm that can no longer cope with the changing environment. Our control expert Mrs. Jensen is able to resolve the problem with an effort so limited that it may be conducted remotely, before the issue develops into something serious.

To emphasize the envisioned benefits, we imagine the same story once again, this time as it would typically develop with the current state of the art in industrial control:

“Nine. That is the only number missing horizontally. But, no! Wait - it is already present vertically? Bugger, there must have been an error somewhere.” It is Saturday morning around 04 o’ clock at Elbonia Cement Industries and operator Joe is alone in the control room. The room’s calm atmosphere, with classical background music and the soft snore from the coffee machine in the small adjacent kitchen makes up a sharp contrast to the inferno taking place outside the soundproof windows. Out there, powerful machinery is reducing an entire mountain into dust, heating it to temperatures that will melt steel and crushing the whole thing to dust one more time. Only to achieve what is properly one of the earliest chemical reactions man learned to master. And doing so with 500t of mountain per hour.

But for Joe, this inferno is merely perceived by means of the hundreds of silent little green numbers, symbols and figures on the large LCD panels in front of him. He has almost completed his third Soduko that night when he is interrupted by the all too well known repeating "ding!...ding!" from the SCADA system requesting his attention for the 317th time tonight. Once again it is the automatic control system for the kiln that has bailed out with some obscure error message. It started a couple of weeks ago and has been more and more common. Slightly annoyed, Joe finally chooses to switch it off and run on manual instead. He never really trusted that thing anyway...

At the staff meeting one month later, plant manager John is frustrated. They have difficulties controlling quality, power consumption has increased 5% and production has gone down, not to mention the unexpected stop they had to make last week because some parts had worn out way faster than they use to. Joe had mentioned something about problems with the control system, but that will have to wait - there is no time to worry about his fancy computer toys when production is not even stable.

When John finally contacts the manufacturer, there is another problem: The only expert control engineer available for the time being is Mrs. Jensen, and the culture in Elbonia makes females on the plant unacceptable. Another 6 month passes by before the young Mr. Petersen finally visits the plant. A task like this is complex and time consuming and is not suitable to be done remotely. It is absolutely necessary to be present on site to coordinate with people in the control room.

All in all, it takes Petersen 3 weeks to get the control algorithm up and running again and fix a number of other smaller issues. The error message mentioned does no longer show up and he is happy the day he can finally leave the hostile environment of the Elbonian desert.

What he did not realize during his stay, was that a new feeding system has been installed that gives twice the amount of material for the same control signal as the algorithm was originally tuned for. So the performance of the control algorithm is still poor after his stay and there are yet more chapters to be written about the control system at Elbonia Cement.

1.2 Background

Cement is produced by crushing and heating of limestone together with various additives. Cement plants are typically very large and complex structures with high throughput of materials and, consequently, high energy/fuel consumption and emission levels. In 2006, FLSmidth gained an order to build the world's currently largest cement production line with a capacity of 12,000 ton/day. In the minerals sector, facilities are typically larger, with capacities one order of magnitude greater.

In this section we will briefly introduce the technology involved from a control point of view, of what is considered a typical, modern cement plant (2008). However, production techniques control strategies and machinery undergoes constant development. On the other hand, the lifetime of some equipment often exceeds half a century. Therefore, it is common that the technology at a single plant range from state of the art to well beyond obsolete.

The number of IO's, e.g. temperature measurements, solenoid valves, etc. may exceed 100.000 for plants with multiple production lines. Hence, the utilization of Distributed Control Systems (DCS) and *Supervisory Control And Data Acquisition* (SCADA) systems is the norm. The SCADA system supplies a Human Machine Interface (HMI) through which the human operator can control actuators and monitor measurements from a centralized control room. The same system handles logging of selected process data for review and reporting to e.g. management and enterprise level systems.

Thus, the control system is hierarchical, with the bottom level comprising discrete sensors and actuators interfacing to Programmable Logic Controllers (PLC). Due to technological development, discrete sensors are being replaced by intelligent counterparts with embedded communication, calibration and diagnostic capabilities. The PLC's handle safety interlocks and sequential starts and shutdowns as well as stabilizing control.

Examples of safety interlocking includes ensuring that the fuel pump to a burner cannot be started if the pilot flame is out, or to automatically stop a feeder loading material on a conveyor belt if the conveyor belt stops for whatever reason. Together with the SCADA's HMI system, multiple actuators can be grouped together so they appear to the operator as only one, e.g. the starting of a burner may involve a sequence of opening valves and starting fans and pumps, but for the operator this will appear as a single operation.

Fast and stabilizing control is performed by PID loops in the PLCs. For example the power to a fan drawing air through some equipment is controlled to maintain a certain differential pressure, measured across the equipment. In this context, "fast" control is denoting time constants in the range of milliseconds to seconds.

Via the SCADA system the human operator will then control the setpoint for the differential pressure rather than the fan power directly. Communication between the PLC- and SCADA layer is usually standard Ethernet with extensive use of fiber optic cable for noise reduction.

For further reading [Wang and Liao 2004] presents a good walk-through of the current state of the art in SCADA systems, and [Mahjoub and Dandachi 2007] gives a case study of a current SCADA system commissioning.

The high-level control system is located on top of the SCADA system in the control system hierarchy, taking over the role of the operator by reading measurements and writing setpoints to the PLCs. Thus, the term “high” in “high-level control” refers to its location in the hierarchy, above the low-level control performed by the PLCs. It is important to note, that the high-level control is not necessarily more technologically advanced or more important to the overall process control than the low-level. The high-level control could in theory be simple a PID, while low-level controllers could utilize the state of the art in fault tolerance and nonlinear neural networks.

High-level control systems are generally not considered a replacement of the human operator but merely a supporting system similar to the role of the autopilot in commercial aircraft. In practice, the high-level control system(s) may be considered a detail of the entire SCADA system especially if both are supplied by the same vendor; in this case they may even share common hardware and other IT infrastructure.

High-level control systems are used extensively in the operation of cement and mining plants in order to maximize production while minimizing fuel consumption and emission as well as down-time of the plant. Due to the high complexity of the plant, control systems can, when used correctly, perform this task much better than human operators. A promotional brochure [ProcessExpert 2008] for a high-level control system for a ball mill - a mill system used to produce cement powder - promise up to 6% increase in production, 5% reduction of power consumption, and 50% reduction in quality variations compared to manual operation.

Nonetheless, plants with no automation at all still exist in some areas of the world. This emphasizes that control, and especially high-level control systems, plays a role of optional, performance increasing acquisitions for these particular kinds of process facilities. Furthermore, it emphasize the fact that the main purpose of these control systems is disturbance rejection rather than stabilization and reference tracking since cement production is a continuous process in which reference signal such as production level and quality are mostly constant.

In the context of control theory, the time constants handled by the high-level control is ranging from a few minutes to several hours. Consequently, sample times are correspondingly long allowing the high-level control algorithms to be running on standard PC hardware with computation time and the limited real time capabilities of common operating systems (2008) not being an issue. Any hard real time control is handled by dedicated hardware, i.e. PLCs, or even special safety-certified PLCs.

The control methodology behind high-level control systems have historically been dominated by Fuzzy logic. Examples include [Sheridan and Skjoth 1984], [Holmblad and Østergaard 1995] and [Yuan et al. 2008]. The tendency in the industry is that this technology is gradually being replaced by Model Predictive Control (MPC) [Maciejowski 2002; Martin et al. 2000].

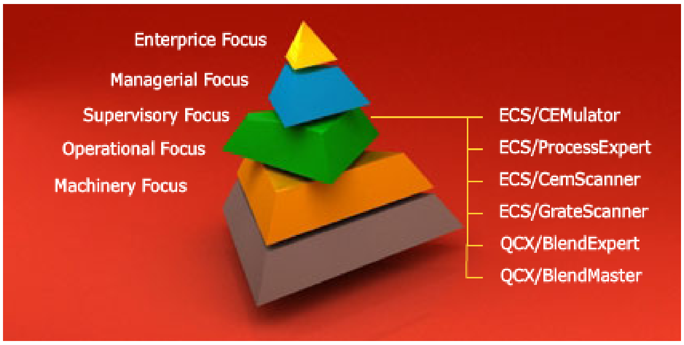


Figure 1.1: Promotional figure of the systems hierarchy with the location of the high-level control system highlighted.

Figure 1.1 shows the location of the high-level control system in the hierarchy of systems on top of the SCADA and PLC levels as discussed here. The higher levels above the high-level control primarily consists of systems to supply plant- and enterprise level management with information about productivity, equipment availability etc. and will not be discussed further.

1.2.1 Cement Production

The raw materials for cement production are lime and clay in the ratio 80% and 20% respectively. To some extent sand, iron ore, shale and various industrial waste may be added to compensate for chemical compounds lacking in the raw materials. Figure 1.2 shows the major steps in the production. The raw materials are crushed, then mixed and ground to a powder about the fineness of common wheat flour in the raw mill and stored in the raw meal silo.

In this stage it is critical to maintain the desired chemical composition of the raw meal despite variations in the raw material. Therefore several steps are taken in homogenizing the materials according to chemical analysis of their composition.

The raw meal is fed to the top of the pre-heater tower which is essentially a large counter-flow heat exchanger where the particles flow downwards and are heated to approximately 900°C in a series of cyclones by hot combustion gases flowing upwards. In the bottom chamber - the calciner - fuel is added and CO_2 is expelled from the lime by heat. The hot raw meal then enters the rotary kiln in which the main burner increases the temperature to app. 1500°C. This liquefies the metallic compounds of the material that sinter together and forms pebble sized nuggets, known as cement clinker. After the kiln, the clinker is flash cooled by cold air on moving grates in the cooler and then stored in silos.

The hot air from the cooler is recycled for combustion in the kiln and pre-heater. A single, large Induced Draft (ID)-fan draws air from the cooler, through the kiln, calciner and pre-heater cyclones and leaves the system as oxygen depleted combustion gas at the top of the pre-heater tower. This ensures that the pressure at any point inside the system is lower than the atmospheric pressure outside so there will be no dangerous hot gas/dust leaving the system in case of small leaks.

The combustion gas is cooled and dust is reclaimed and returned into the production. Depending on the chemistry of fuel and raw materials further treatment of the combustion gas may be required for environmental reasons, e.g. scrubbing to remove sulfur, before the gas is released to the atmosphere.

The cooled clinker is finally ground to the well known, grayish cement powder in the cement mill where gypsum and fillers such as fly ash may be added and the cement is ready for shipping.

Coal, heavy fuel oil and natural gas are common fuels for the kiln and calciner. These fuels are homogeneous, controllable and predictable in terms of calorific value and feeding. However, there is an increasing focus on alternative fuels ranging from wood chips, bone meal, and dead animals, to tires and various liquid and solid waste products. The very high temperature in the kiln makes cement production attractive for waste incineration and it is common that alternative fuels are introduced not for their calorific value but for the purpose of destruction. Any ash of such materials will be embedded in the final product.

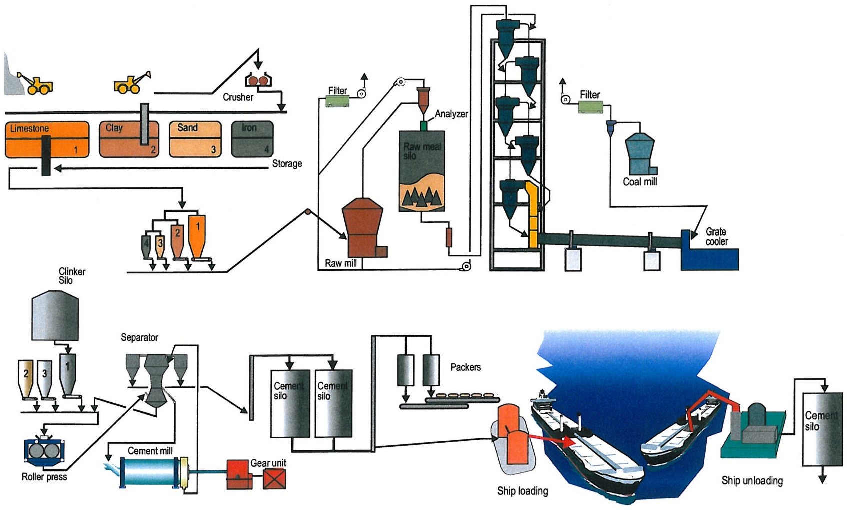


Figure 1.2: Principles of cement production.

The inhomogeneity and oftentimes unknown composition of such materials is a special challenge for the stabilization of the process and high-level control systems for optimizing the use of alternative fuels is an upcoming market.

[Maranzana 2002] and [Prouty 2006] exemplifies recent installations of control systems in the cement industry.

For a comprehensive introduction to cement production see [Duda 1977].

1.2.2 The Industry

The company engaged in this industrial Ph.D. is FLSmidth Automation which is a part of FLSmidth & Co A/S, a supplier in the cement and minerals industries ranging from complete, turn-key factories to equipment and services. Within the cement industry, the company holds a world leading position with one third of the world's cement being produced using FLSmidth equipment. The minerals sector is a focus area where the company is expanding aggressively.

The company employs about 9000 people worldwide and has a turnover of 2.5B€ (2007 estimate).

FLSmidth Automation (FLSA) supplies control systems and electrification for the industries mentioned above. About 400 people are employed worldwide, responsible

Figure 1.3: Top: operator in control room. Bottom: automated robot laboratory.

for a turnover of 100M€.

The R&D department in Valby, Copenhagen holds 40 employees and is essentially doing software development. This Ph.D. project relates mainly to a single product in the portfolio branded “Process Expert”; a high-level control system for the cement and mining industry. In case of high-level control, other suppliers include ABB[1] and Pavilion[2].

Information about 3. party high-level control systems is sparse, and largely anecdotal. Lacking such information and considering the company’s high market share, the product offered by FLSmidth Automation is therefore assumed to be representative for the state of the art in high-level control at least in the context of cement and mining.

The ability to supply high-level control systems not only with high performance, but also a very low maintenance requirement, will increase the company’s potential, compared to its international competitors.

[1] http://www.abb.com
[2] http://www.pavtech.com

The global market size of advanced optimization solutions in the cement industry counts for 25-30M$. Up until now, no specific solutions for the long-term performance of these solutions has been presented in the market. The market is a quite trend-oriented market where the great potential becomes a reality after some limited number of successful installations.

1.3 The Maintenance Problem

Use of high-level control systems in cement plants is well established. It is acknowledged that their use can improve performance by increasing production and reduce the environmental impact of the plant.

In spite of this, many systems are taken out of use after a short period due to physical changes in the plant, which are not handled by the control system. The problem is that due to the changes, the control system needs readjustment and this often requires skills that are not available on the plant. One reason for the adjustment requirements is that the emphasis of design of current control systems is on optimization of a given plant with little or no consideration of maintainability of the control system.

Even though the digital computers and software running the high-level control system may be considered time invariant, the environment surrounding the control system is not. For example the characteristics of the plant or the raw materials may slowly drift away from what the control system was adjusted for at install time.

This may be caused by general wear and tear but also poorly maintained instruments causing the drift of signal levels. Currently these issues call for maintenance of the system, such as retuning and reconfiguration of different modules. In addition to these gradual changes, the plant characteristics may change abruptly caused by process changes such as new recipes or rebuilding of the plant. All issues that may one way or another reduce the performance of the control system.

Modern process control systems are large and complex bulks of software. The development of process control systems have gradually moved from being a pure control engineering discipline, possibly with aspects of electronic, mechanical or chemical engineering to be enterprise-sized software development with a typical control system implementation representing millions of lines of source code.

Thus, today's successful control system manufacturer is a software house with certain domain knowledge (control engineering) rather than an engineering company. This introduces a number of new challenges caused by the complexity of such large software system. Some challenges may be successfully addressed with methods already developed by the software industry, while others are unique to process control systems due to their special properties and needs.

Traditionally, the performance of a control system is measured as the ability to, (when optimally tuned) increase the production yield while meeting the environmental requirements such as emission levels. However, this might be a poor indication of the long-term advantage, if the system looses performance and reliability soon after installation due to poor adaptation to the issues above. Other characteristics of the system are important as well, for instance how "gentle" the system controls the plant by reducing overall wear and tear and spending of wearing parts.

Ultimately, the human operator might choose to disable the system and run the plant manually, which causes a significant performance loss. These issues are well recognized within the industry but they have so far earned only limited attention from academia.

The exact cost of disabling the high-level control may be difficult to estimate. However as a rough indication, the fuel consumption of a typical cement kiln is reduced by 4%[3] when the high-level control is running. At a fuel consumption of e.g. 25t of coal per hour at a price of 150USD/t the annual loss is 1.3MUSD in fuel cost. In addition to this there is the loss associated with lost production and quality variance, for the kiln alone.

1.3.1 Scope and Limitation

The motivation for this project is that a high-level control system should perform just as well after a year of service as it did right after commissioning. The scope of this project though is limited to maintenance of the high-level control algorithm which may be defined at an arbitrary abstraction level by means of source-code, configuration data and the like. If a high level control algorithm is not maintained at regular intervals, its performance is expected to develop as depicted in Figure 1.4. If the performance drops below a certain level determined by the users (dotted line), the system may ultimately be switched off.

In this context, the aim of this project is therefore to attain a constant high performance (dashed line) with no or significantly reduced maintenance needs.

Obviously, the high-level control and SCADA system are subject to maintenance issues beyond this. At the bottom of the hierarchy sensors and actuators are subject to wear and communication channels including cables and wireless links may break and wear as well in the hostile environment. At the top of the hierarchy the entire

[3]Stated in FLSmidth advertising,
http://attachments.flsmidth.com/FLSA/Brochure/ProcessExpert_Kiln.pdf

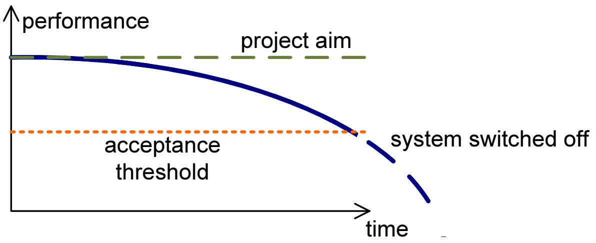


Figure 1.4: Predicted performance of high level control system without proper maintenance.

SCADA computer platform may currently (2008) consist of tens of servers, clients, network switches, storage, Uninterruptible Power Supply (UPS) etc. very similar to the IT infrastructure found in any other medium sized office facility. Such an installation naturally requires proper maintenance to keep the high-level control system runnable. However, these issues are considered beyond the scope of this project so focus is narrowed to the control algorithm alone which is the application specific software part in the context of control engineering.

Chapter 2

Analysis of the Maintenance Problem

The detailed causes of malfunction for the high level control system (HCS) remain largely undocumented. Only anecdotal evidence is available from personnel involved in the commissioning and maintenance of control systems.

The SCADA system is logging all measurements and actuator setpoints and such log files are available from a number of sites. For example from such a log file it is possible to deduce when the HCS has been running the last year. If, in addition, a human-maintained log existed, reporting all events on site such as rebuilding, changes in fuel type or raw material, unplanned downtime, and, moreover, a reason was available for each time the operator chose to switch the HCS off, then it may be possible to correlate the process data and the human log to determine the events which affect HCS performance.

But since cement is a low-value bulk product and its production is not associated with any significant danger it is not subject to particular regulations or audit that could enforce careful logging of anomalies in the production. One should note that it might not be obvious for the average customer that replacing e.g. a pump or valve with a different model is likely to affect the control system, and that information about such a modification is not logged in any way.

However, the control system as introduced in Section 1.2 consisting of DCS, SCADA and HCS, is similar to those found in other continuous processes, e.g. power plants, waste incineration and refineries. It has therefore been considered to obtain such a log from these industries. In particular nuclear power plants are known to keep such logs due to the tight safety regulations but because of the sensitive nature of unplanned downtime and equipment failure in such a facility, it has not been possible to obtain such logs.

Available data is therefore limited to SCADA log files and sparse, oral accounts from maintenance personnel. SCADA log files are generated with focus on process analysis and generally contain all measurements and setpoints. Additionally, a few calculated performance indicators such as average availability of machinery, etc. may be included.

Signals generated by the control algorithm are only logged if explicitly specified. Due to storage limitations, the resolution of the data is oftentimes reduced e.g. all signals are logged every second the last week, every minute the last month, and hourly average the last year.

Thus, if a particular incident is to be investigated for which the sequence of events is of interest, these log files may be insufficient if signals are only available as hourly or even daily average.

Initially, various initiatives have been considered briefly for producing log files with a higher level of detail dedicated to identifying maintenance issues. However, this would turn out to be particularly expensive and have a time frame of two or more years for practical reasons, and have therefore been deemed unworkable.

2.1 Causes of Malfunction

The sources of reduced control system performance that have been identified are based on oral accounts from maintenance personnel. A number of examples of such accounts are cited here:

> “The single blade damper in the tertiary air duct is worn down by clinker dust so only the axle remains. That means this actuator is effectually disabled and the HCS is no longer able to affect the air flow.”

> “A damper has no position sensor. Rather, the torque (motor current) is used to detect when the damper has reached an end point. The damper gets jammed by accumulated material in an unknown position, fooling the HCS to believe it is at an endpoint.”

> “The burning zone temperature is controlled poorly because the fuel flow can only be changed in steps of 0.2t/h (which is coarse). A closer look reveals that this is not due to mechanical limitations but a dead band that has mistakenly been added to the feeder setpoint signal.”

> Note: It remains unknown why this has not manifested itself immediately after commissioning. It is apparently common though, that deadbands are applied to setpoints in order to reduce actuator wear.
>
> “Cement dust has entered the computer casing of the HCS server damaging its hardware.”

It seem to be a general tendency that performance issues are related to the level of abstraction associated with a hierarchical control system architecture: The HCS is (as intended) unaware of the underlying implementation of actuators and any nested control loops. For example in case of the jammed damper the HCS is unaware of the particular realization of the damper’s open/closed status because the estimation of this signal based on motor torque is done at the PLC level. This ensures that the HCS can remain unchanged in case the damper is replaced by a different model equipped with an absolute position sensor.

Generally, process controller performance issues relate to one or more of the following factor which are not limited to the cement and mining industries only:

- wear and tear of machinery
- blocking and material build up
- change in raw materials and fuel
- replaced machinery or wearing parts
- general drift of process parameters
- rebuilding or addition to plant
- soft- and hardware failure

In the following sections we consider methods of addressing each of these issues.

2.1.1 Raw Material Changes

In the cement and minerals industry, the chemical and mechanical properties of the raw materials will vary over time due to local geological variations in the quarry. For example changes in hardness will affect the working point of the raw mill and changed chemistry will affect the kiln and therefore the control system. If the properties of the raw material are known compensation can be performed, e.g. material with low lime content from one end of the quarry can be mixed with high lime material from the other end of the quarry to achieve a uniform chemistry.

Methods for addressing such issues are well established. The composition of the quarry can be mapped by drill samples, and/or by the combination of on stream analysis of each truckload of material with GPS tracking of the truck. Based on the on stream analysis of the material, a data terminal in the truck can instruct the driver where to obtain the next load of material in order to maintain a uniform mixture. [Collins et al. 1995] shows an example of the gains associated with the installation of an on stream analyzer.

After initial crushing the material is stacked (See Figure 2.1) in a manner that further serves to increase homogenization.

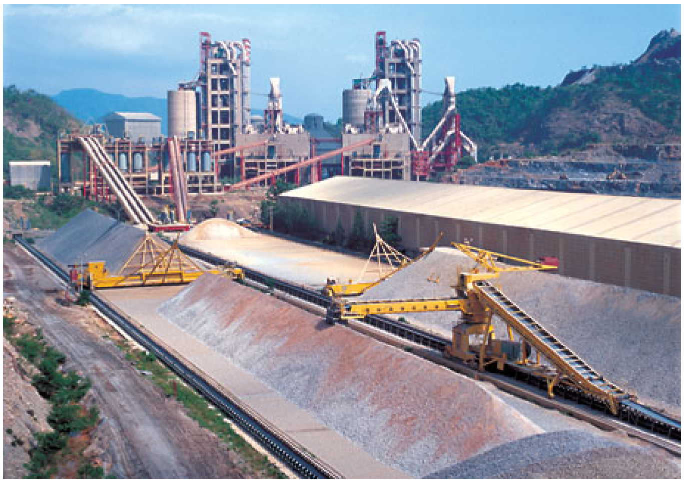


Figure 2.1: Stackers and reclaimers in operation.

Throughout the production facility, materials are sampled automatically and send by air tube to automated laboratories for analysis and the result is used in the HCS.

These steps to reduce raw material changes are extensive cf. the size of machinery (See Figure 2.1), and aim at controlling the raw material composition and thus, quality variations disregarding the HCS's used downstream in the production. Any positive influence on HCS performance and maintenance is therefore merely a side effect. Nonetheless, some potential of improvement remains for the HCS. The sample frequency and sample delay are critical and may both be close to the time constant of the process. Currently Kalman filters are used to estimate the value between samples and any improvement to the accuracy of these estimates will naturally increase the tolerance to raw material changes.

2.1.2 Fuel Changes

The rate of adding fuel to the burning process is an important control signal and any changes in the fuel, such as calorific value therefore strongly affects the HCS performance. Preferably, fuel should be homogeneous and possible to be dosed accurately. Coal, Pet coke, fuel oil and gas tend to be homogeneous and easy to dose, but calorific value may still vary from one batch to the other. More importantly the increasing use of alternative fuels emphasize these issues [Kääntee et al. 2004; Keefe and Shenk 2007; Zabaniotou and Theofilou 2008]. One popular fuel is scrap tires that are often added whole. Since sizes of the tires may vary the use of whole tires may impose both significant quantizations of the control signal and dosage problems. A solution to this is shredding the tires which however, require additional energy.

Other fuels may either be so inhomogeneous that the calorific value is unknown. HCS dedicated to handling alternative fuels are upcoming [Recke 2006]. The user can be forced to enter at least a qualified guess of the calorific value of the fuel. If extra information such as monetary cost and content of contaminants such as sulphur and chlorine are entered such a control system can supply added value like the ability to choose the cheapest fuel while maintaining product quality and environmental restrictions.

2.1.3 Diagnosis and Fault Tolerance

Accumulating material may block sensors, limit the movement of actuators and/or affect the dynamics of the process causing a mismatch between the controller's tuning and the process.

Intelligent or “smart” sensors/actuators [Song and Lee 2008; Eccles 2008] are addressing some of these issues by implementing self-monitoring, calibration functionality etc. The IEEE 1451 standard[1] describes one architecture for such transducers.

A traditional sensor of e.g. temperature generates a voltage or 4-20mA signal which is then to be converted to a temperature by a PLC. In contrast, a smart sensor may perform signal validation, internal calibration and supply meaningful error reporting rather than supplying a incorrect measurement in case of a fault.

This means that the control system, including the HCS, can be informed explicitly about many kinds of in-transducer faults and be designed to act appropriately.

Diagnosis and fault tolerant control are well established topics in academia [Simon et al. 2002; Blanke et al. 2003; Perrier and Kalwa 2005] which aim at designing

[1] http://ieee1451.nist.gov/

control systems that degrade gracefully in case of a fault. For example if a model of the process during what is considered normal operation is available, process outputs can be continuously compared to simulated outputs from this model. In the ideal case the two signals should be nearly identical. Divergence between the real process and the model outputs will indicate a change in the process - a fault. By means of the known model the control system may handle the fault in a formal manner and continue operation with degraded performance and possibly even diagnose the problem.

A popular example of fault tolerant control is that of a commercial airliner loosing the ability to move the rudder and thus control of the craft's yaw rate. A fault tolerant control system will detect the mismatch between the input (pilot's movement of flight controls) and output (measurements of aircraft's state) according to a known model. Using the methods of fault tolerant control, the controller will automatically reconfigure so that the flight controls that would normally move the rudder will now apply different power to the starboard and port thrusters to control yaw. Thus, maneuverability is maintained although at reduced performance.

While smart transducers may be able to handle a number of known predictable faults, fault tolerant control would address issues like wear, blocking, changes in raw material etc. in a general, formalized framework. Furthermore, even limited diagnosis capabilities would supply added value to a HCS product, e.g. simply the ability to state that something physical in the process has caused the problem and not the HCS itself could be beneficial.

Finally the use of observers [Hendricks et al. 2005] (Kalman filters) or "soft sensors" can further increase the tolerance to failing sensors and even substitute physical sensors.

2.1.4 Adaptive Control and Auto Tuning

The low level PID loops inherently hides the effect of some of the issues listed earlier in this section minimizing their effect on HCS performance. However, a gain or a major process time constant may eventually drift so far away from what the HCS was tuned for, such that performance becomes unacceptable.

An especially "forgiving" tuning of the HCS could alleviate this issue. In practice, close-to optimal tuning may very well be impossible why a forgiving tuning could in fact result in a better performance at the bottom line as illustrated in Figure 2.2. Robust Control [Skogestad and Postlethwaite 2005] is formalizing this by defining an envelope of process changes within which the controller is guaranteed to perform.

The meaning of the term *performance* is discussed further in Section 2.4. Adaptive control [Åström and Wittenmark 1994] is in a way an opposite approach. By

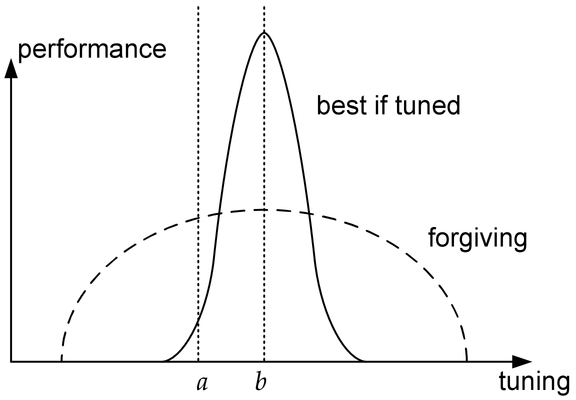


Figure 2.2: Forgiving versus optimal tuning. An optimal controller (solid) only performs if the controller/process mismatch is small (b). Therefore, a sluggish, forgiving controller (dashed) may outperform the optimal controller in practice where controller/process mismatch may be significant (a).

continuously identifying a model of the process by means of system identification [Ljung 1986; Bishop 1995] and adjusting controller parameters accordingly, constant closed loop behavior is maintained.

Alternatively, auto tuning may be performed only occasionally initiated either manually or automatically when needed [Åström and Hägglund 1984]. However, some changes remain that requires nonparametric modifications of the control algorithm such as the addition of a suddenly needed feature to a running algorithm. More importantly, changes in the process or operation may drive the control system into an operational state or "behavior" different from the ones of concern during install time.

In turn, this could trigger a bug or a missing feature that, though having no influence on the controller performance at install time, could be fatal for the control system's operation. Furthermore, such a malfunction may very well be caused by a part of the algorithm that is unrelated to the implemented control theory and consequently further beyond the scope of these research topics.

2.2 The High-level Control System

Despite the control methodologies and design principles involved, the HCS algorithm is eventually implemented by means of software and this software is executed on suitable hardware. The HCS may more or less share the hardware platform of the SCADA system. Presently, this platform consists of off-the shelf servers with commercial operating systems, network equipment etc. and represents maintenance issues like any other “server room”. Technological advances in this area may lessen the need for maintenance as well as decentralized, grid-based control system architecture may add redundancy.

In case of excessive process changes or rebuilding of the plant the algorithm may need adjustment to cope with the new control task. Since the time is not ripe for large scale, self programming computers human redesign of the algorithm and possibly of the hardware will eventually be needed. An example of such a scenario is shown in the following Section.

Nonetheless, concepts such as Plug and Play Control [Bendtsen et al. 2008; Michelsen et al. 2008] could alleviate this issue to some extent by allowing the addition and removal of transducers without human reconfiguration of the control system.

2.3 Maintenance Example

In 2005 a high-level fuzzy control was commissioned for a kiln. Less than one year later, it is realized that the controller is abandoned. The question is what has caused the operators to prefer manual operation and to discontinue the use of this controller? The answer is found by investigating the SCADA log files from the period together with the source code of the control algorithm. Figure 2.3 shows the log of some performance indicators from the HCS which was installed or re-tuned by expert personnel until about 10 month later.

The system is set up so that operators in the control room are allowed to switch on the system once the process state and other conditions are within the design limitations of the system. The algorithm will then control the process fully autonomously. If suddenly the conditions needed are no longer satisfied, an alarm is sounded and the control algorithm disables itself, allowing the operators to manually bring the process back into a state from which the control algorithm can take over.

In any case, 2 hours must pass before the control algorithm is re-enabled and can be switched back on by the operator. This timeout is roughly matching the largest time constant of the process, allowing the process to return to a near-steady state before the control algorithm is engaged.

The cause of such unintended behavior could be e.g. break-down of machinery, failing actuators, excessive disturbance, or even the control algorithm itself e.g. by means of a bug causing it to execute a fatal control action.

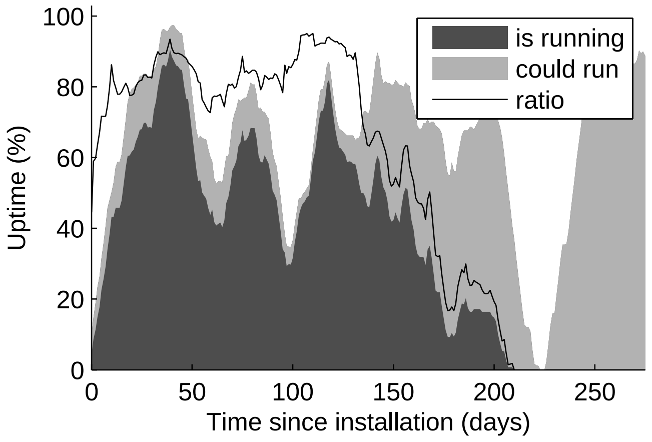


Figure 2.3: Development of uptime for typical control algorithm after installation.

Figure 2.3 shows a moving average of the uptime of the control algorithm. The light shaded area shows when the above mentioned conditions area satisfied, and the dark shaded area when the operator have actually engaged the algorithm. The black line shows the run factor being the ratio between the two, i.e. to what extent the operator have chosen to engage the algorithm when having the opportunity to do so.

The relatively high uptime during the first months after commissioning indicates not only the control algorithm's ability to control the process, but also the operators' motivation to switch on the control algorithm whenever possible. However, from approximately day 125 to 200 its uptime is decreasing gradually, indicating increasing problems for the algorithm to control the process and/or the operator's increasing dissatisfaction with the product. Eventually, after an extended downtime of the entire plant around day 200-230, the control algorithm is no longer used. Note that the development of the uptime in this example is very similar to the performance initially predicted in Figure 1.4.

Whether the cause of the decreased uptime of the control algorithm is to be found internally or externally is irrelevant. Fact is that the utilization factor of the advanced control system have dropped to zero in just 200 days, and the system needs attention from expert maintenance personnel in order to regain its initial performance.

By carefully inspecting the log files and control algorithm source code it turns out that the latter contains the following statements of interest:

```
1: if ID fan stops then
2:     stop kiln drive
3:     stop kiln feed
4: end if
5: if kiln feed and kiln motor have been running uninterrupted for 2 hours then
6:     enable fuzzy control
7: else
8:     disable fuzzy control
9: end if
```

Thus, if just one of the 3 signals ID fan, kiln feed or drive indicates a stop, the fuzzy control stops immediately and switches to manual mode. A minimum of 2 hours must pass before the operator is allowed to switch the controller back on.

The SCADA log files shows, that tiny glitches appeared increasingly frequent in the kiln feed measurement at the time the uptime in Figure 2.3 started to drop. While the duration of each glitch is only a single sample and therefore insignificant from a process point of view, it is enough to disable the entire control algorithm. The cause of the glitches was not investigated.

The fuzzy controller is designed to run the kiln system only in its near-steady state and not the nonlinear transient mode that is experienced after a full stop. The intention of this rule set is therefore to ensure that the controller is not switched on during the first hours of a full startup where the process is in transient mode. However, a drawback of this simple design is that the controller becomes sensitive to dropouts that are too short to cause significant transient response. Furthermore, these short dropouts might even be related to the sensors and signal processing rather than the physical actuator.

A problem like this could have been avoided at design time, e.g. by following best practices of never making a decision based on a single measurement/sample. Nevertheless the design works well on other plants as well as this plant at commissioning time. Only at this particular plant, due to changes in feed, wear or other, a problem arises some time after installation. It must therefore be considered a *maintenance* issue.

Currently, these issues are addressed by regular maintenance, and service contracts. At regular intervals an expert will visit the plant and work on issues like this according to customer needs.

The time needed to realize this particular problem was approximately two man weeks predominantly used on understanding the complex algorithm. In contrast, the effort needed to make the algorithm robust towards the mentioned glitches is relatively small.

2.4 Discussion

The term *performance* of a control system is traditionally denoting some kind of specific metric such as the mean squared error between two signals. In case of HCS this could include the deviation from target end product quality, total daily production and so forth. In the context of maintenance however, we will expand the term to include all written and unwritten requirements of the HCS including aspects such as user experience for the operator and CPU load etc. That is to say, all aspects of the control system's behavior that could affect its acceptance and amount of added value to the customer. For example as seen in Section 2.3, poor operator perception can contribute to the HCS being permanently disabled.

Next, we can in this context define maintenance as:

Maintenance of high-level control systems: post-install modifications of the system in order to keep a high performance.

and the aim of this project is to eliminate or reduce the amount of maintenance needed by the HCS.

The low level control loops are predominantly of the PI type and are often poorly tuned. In practice the control loops may only be given attention during commissioning if their behavior prevents starting up the process. Thus, controllers may even be left with default parameters. Once the process is running continuously, any reconfiguration of these loops may be banned due to fear of causing a production stop which is associated with great monetary loss.

Adaptive, robust and fault tolerant control as well as auto tuning as discussed earlier in this chapter have all been demonstrated successfully on such loops of relatively low complexity, especially if noise is low and the dynamics can be modeled in sufficiently accurate way. It is therefore our belief that the introduction of these techniques in the low level control loops could contribute significantly to the overall process performance. Possibly even more than the gains achievable by the application of a HCS as discussed in Section 1.2.

However, to our knowledge, there has not been any successful implementation of these technologies in industrial high-level control where systems are generally of high dimension and accurate models are not available. Most importantly, these techniques themselves will represent additional software/algorithms and parameters within the HCS that will require maintenance.

These advanced control methods may also require that the controller is implemented or specified in a dedicated framework or that the controlled process is modeled in a special manner. This requires additional man-hours in the design phase which can be acceptable if for instance the controller is to be designed only once and then embedded in a mass produced product. [Sánchez 2003] is an example of such an application in which a robust and fault tolerant controller is designed for a Compact Disk player. Low level control loops in process control as mentioned above may also fulfill these criteria.

Currently it is unattractive from a HCS point of view though, since the control algorithm is customized significantly for each installation and any additional design and tuning is costly.

In this project we will therefore focus on, and alleviate the maintenance, of the control algorithm which requires human interaction. Even if the techniques discussed so far are successfully applied a subset of maintenance scenarios, such as plant reconstruction, will remain which requires algorithm modifications.

2.5 Summary

The data available to document the extent of the maintenance issue is limited to SCADA log files and anecdotal evidence. While downtime of controller and machinery is registered in these logs, the root cause of the downtime is not. Oral accounts show that downtime or malfunction of an unmaintained control algorithm is related to issues such as wear and tear of machinery, changes in fuel and raw materials, material build-up and reconstruction or replacement of machinery. In this way the characteristics of the process to be controlled is changing from the requirements specification that the control algorithm has originally been designed for.

A significant effort is already put on reducing raw material changes by means of homogenization and control of chemical composition, and these steps are taken disregarding any high level control systems (HCS) downstream on the production line. Regarding fuel changes the increasing utilization of alternative fuels has given rise to handle the use of multiple and possibly heterogeneous fuels.

Advanced control concepts, such as fault tolerant control and auto tuning, could potentially address all of these issues but it is concluded that for the time being, the greatest benefit from these methods will be in the low level control loops and not in the HCS. It is therefore decided to focus the attention towards the remaining subset of maintenance issues that will inevitably require human expertise to redesign and adjust the control algorithm. Rather than attempting to eliminate the need of human interaction, the aim will be to reduce the amount of man hours when needed.

Chapter 3

Tools and Algorithms

Anecdotal evidence suggest that issues related to operator's perception and interaction with the control system is sometimes wrongfully seen as reduced performance of the control algorithm. Poor situational awareness may lead to interventions that are incompatible with automatic control, or the algorithm's control actions may be wrongfully perceived as harmful so the operator choose to disabled automatic control. For example it has been observed that Fuzzy type controllers gain acceptance among operators easily because they tend to alter one actuator at a time, similar to manual operation. Modern MIMO type controllers though can be met with skepticism because they tend to apply frequent, small adjustments which may be counter intuitive to the operator.

Indeed, the human machine interface (HMI) of the control room may have potential of much improvement [Lau et al. 2008] that will enhance situational awareness, as well as educating operators may reduce wrongful disabling of control algorithms. However, these subjects are considered beyond the scope of this project.

Nonetheless, these observations have served as inspiration to investigate the user interface (UI) involved in developing, implementing, and maintaining the control algorithm. Once the performance of an algorithm has dropped so significantly that maintenance is needed, this is the UI that expert personnel must use. So, if performance degradation is unavoidable, perhaps there is at least a chance to improve the available tools, which will ultimately reduce the number of man hours used for maintenance and hence the overall downtime of the algorithm.

Moreover, such improvements may induce that the algorithm is designed in a more consistent and verifiable manner from the beginning, and therefore easier to maintain in the future. It is not the aim of this project though, to consider general usability aspects of the software.

3.1 The Algorithm Editor

The control algorithm is implemented by means of source code. Rather than using a standard development environment during implementation of each algorithm FLSmidth have created an environment dedicated to high-level control and integrated with the rest of the control (SCADA) system.

Within this environment an algorithm is constructed as a network of interconnected objects as depicted in Figure 3.1. Measurements are read by input objects to the left, and actuator setpoints are written by output objects to the right. Lines indicate data flow which is always from left to right. Each object is encapsulated and have defined inputs and outputs on which some sort of signal processing may be conducted. Objects may be grouped together to further enhance encapsulation and modularity. This means that some of the objects as seen in Figure 3.1 contains a plurality of sub-objects which may again hold a number of sub-objects and so on.

Hard coded, configurable objects for which the source code cannot be modified are available for common functionality such as a Kalman filter, simulation of a LTI system, Fuzzy rule engine and the calculation of various performance indexes. But most importantly, a general purpose utility object is available, in which its business logic can be implemented by means of custom source code. Currently Visual Basic and Matlab are the supported languages.

The editor for such an object is seen in Figure 3.2. The source code is edited to the left whereas the table to the right contains internal variables, inputs and outputs. These variables are persisted between executions, thus representing the algorithms internal state.

When maintenance is performed on the algorithm, the work is done in these objects.

3.1.1 Control Algorithm Framework

The editor projects a direct mapping of the underlaying network of interconnected algorithm blocks so its description also covers this choice of architecture for control algorithms to be implemented. This architecture is a hybrid between data flow programming as known from products such as Simulink and Labview and traditional text based functional and imperative programming.

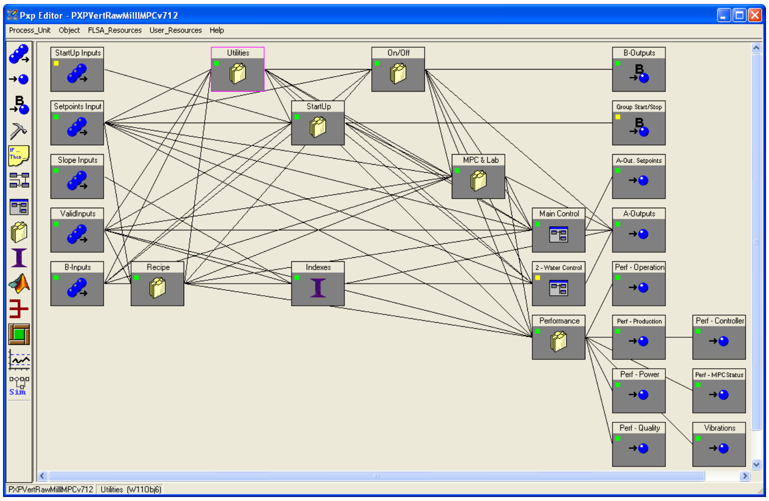
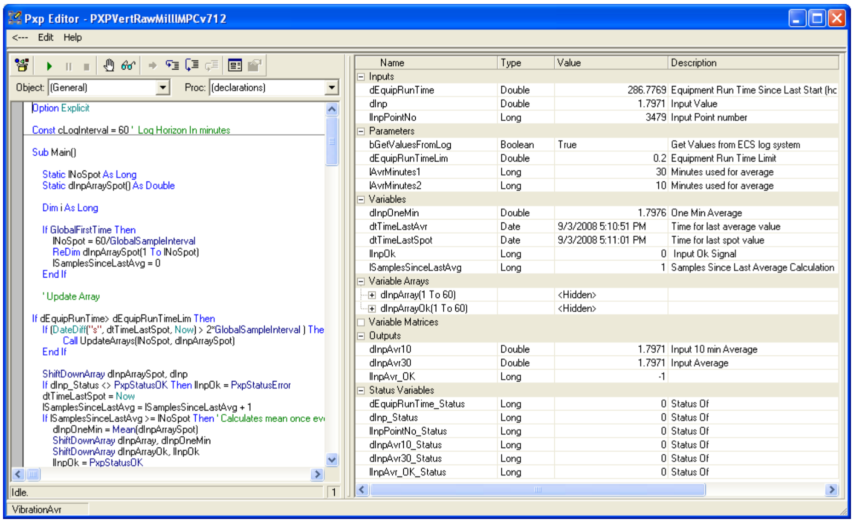


Figure 3.1: Algorithm editor viewing the structure of an algorithm controlling a vertical raw mill.



```
Option Explicit

Const cLogInterval = 60 '  Log Horizon In minutes

Sub Main()

   Static lNoSpot As Long
   Static dInpArraySpot() As Double

   Dim i As Long

   If GlobalFirstTime Then
      lNoSpot = 60/GlobalSampleInterval
      ReDim dInpArraySpot(1 To lNoSpot)
      lSamplesSinceLastAvg = 0
   End If

   ' Update Array

If dEquipRunTime> dEquipRunTimeLim Then
   If (DateDiff("s", dtTimeLastSpot, Now) > 2*GlobalSampleInterval ) The
         Call UpdateArrays(lNoSpot, dInpArraySpot)
   End If

   ShiftDownArray dInpArraySpot, dInp
   If dInp_Status <> PxpStatusOK Then lInpOk = PxpStatusError
   dtTimeLastSpot = Now
   lSamplesSinceLastAvg = lSamplesSinceLastAvg + 1
   If lSamplesSinceLastAvg >= lNoSpot Then ' Calculates mean once ev
      dInpOneMin = Mean(dInpArraySpot)
      ShiftDownArray dInpArray, dInpOneMin
      ShiftDownArray dInpArrayOk, lInpOk
      lInpOk = PxpStatusOK
```

Name	Type	Value	Description
⊟ Inputs			
dEquipRunTime	Double	286.7769	Equipment Run Time Since Last Start (hc
dInp	Double	1.7971	Input Value
lInpPointNo	Long	3479	Input Point number
⊟ Parameters			
bGetValuesFromLog	Boolean	True	Get Values from ECS log system
dEquipRunTimeLim	Double	0.2	Equipment Run Time Limit
lAvrMinutes1	Long	30	Minutes used for average
lAvrMinutes2	Long	10	Minutes used for average
⊟ Variables			
dInpOneMin	Double	1.7976	One Min Average
dtTimeLastAvr	Date	9/3/2008 5:10:51 PM	Time for last average value
dtTimeLastSpot	Date	9/3/2008 5:11:01 PM	Time for last spot value
lInpOk	Long	0	Input Ok Signal
lSamplesSinceLastAvg	Long	1	Samples Since Last Average Calculation
⊟ Variable Arrays			
⊞ dInpArray(1 To 60)		<Hidden>	
⊞ dInpArrayOk(1 To 60)		<Hidden>	
Variable Matrices			
⊟ Outputs			
dInpAvr10	Double	1.7971	Input 10 min Average
dInpAvr30	Double	1.7971	Input Average
lInpAvr_OK	Long	-1	
⊟ Status Variables			
dEquipRunTime_Status	Long	0	Status Of
dInp_Status	Long	0	Status Of
lInpPointNo_Status	Long	0	Status Of
dInpAvr10_Status	Long	0	Status Of
dInpAvr30_Status	Long	0	Status Of
lInpAvr_OK_Status	Long	0	Status Of

Figure 3.2: The utility object editor.

Data flow programming is attractive for control theory implementation due to the similarity with the block diagrams commonly used to depict signal processing in general while the general purpose imperative programming adds flexibility. Whether a control object is hard coded or not, is just a matter of allowing customization of that particular object.

However, the data flow paradigm is only realized on a macroscopic level for directing signals from one object to another. Apparently it has been considered too complex for primitive operations on signals since this is exclusively handled by imperative languages, hard coded or not.

Although the product is extensively used for editing common source code, it does not include any of the supporting tools associated with modern software development such as version control, code style enforcing and unit testing.

Information about other similar products within the industry is not available. It will therefore be assumed that the product described above represents the state of the art in the industry. However, the data flow and encapsulation specific for this particular editor and/or the algorithms implemented in it can be seen as merely a graphical presentation of the encapsulation and abstraction offered by traditional software paradigms such as methods, classes, interfaces, network services etc. The findings discussed in this chapter is therefore relevant for any control algorithm design involving multi-paradigm programming and encapsulation.

3.1.2 The Control Algorithms

A number of control algorithm implementations that have subjectively pointed out as being representative, have been inspected in order to understand needs and identify common operations and shortcomings. Immediate observations include that large amounts of the source code performs operations which can hardly be expressed in terms of classical control theory by means of transfer functions, LTI systems etc. It has been found particularly challenging to track the way signals, carried as the value of variables, are routed through the algorithm. To some extent, this is a trade-off from the concept of encapsulation and abstraction. Wherever a signal crosses the boundary of an encapsulation, the name of the carrying variable may change

Many common, but small operations are not standardized probably because it is often non-obvious that two different implementations are conceptually performing the same operation. For example validating a sample of a signal according to some criteria and storing this value in a history (array) of samples, a history that must be reset in certain situations, e.g. restart. Such functionality is implemented in multiple different ways which makes an algorithm less readable and prone to bugs. An operation in the nominal case is typically accompanied with alternative operations to be conducted in a plurality of non-nominal situations. One typical non-nominal situation is the first execution of the algorithm after a reset in which various buffers, time spans and integrators have to be reset. This induces a large number of, sometimes nested, conditional branches in the code which is not handled in a standardized manner.

A considerable amount of the internal signals are boolean which lead to more conditional branching. Loops are rare, and are mainly used for iterating through an array of values, which is typically storing a short history of some signal.

Another common operation is the testing whether a signal is within a given interval, or above or below some threshold. Although such operations are conceptually simple their implementation turns out to be complex because each test may have to be supplemented with some kind of filtering and hysteresis etc. This problem is illustrated by Example 1:

Example 1

Consider the situation that, at design time it is decided that if the signal x is above the threshold H then the algorithm must perform operation $\mathbf{a}$ and otherwise $\mathbf{b}$. In code, this is implemented by the simple lines

```
if $x > H$ then
    do $\mathbf{a}$
else
    do $\mathbf{b}$
end if
```

but once the algorithm is to be commissioned on site, it turns out that x is particularly noisy so a hysteresis must be introduced. Therefore the programmer reformulates the problem to "if x have been greater than H_h for longer than T_h samples perform $\mathbf{a}$ or if x have been less than H_l for longer than T_l samples perform $\mathbf{b}$". In code, one implementation of this could be

```
if $x > H_h$ then
    $t_h \leftarrow t_h + T_s$
    if $t_h < T_h$ then
        $s \leftarrow$ high
    end if
```

```
else
    $t_h \leftarrow 0$
end if
if $x < H_l$ then
    $t_l \leftarrow t_l + T_s$
    if $t_l < T_l$ then
        $s \leftarrow$ low
    end if
else
    $t_l \leftarrow 0$
end if
if $s =$ high then
    do **a**
else
    do **b**
end if
```

The functionality is achieved by introducing 3 more variables (t_h, t_l, s) and the number of parameters are in creased from 1 (H) to 4 (H_h,H_l,T_h,T_l). T_s is the sampling time.
The amount of code and its complexity has increased considerably but most importantly the same functionality can be achieved in a variety of ways.

The fact that what is essentially the same but complex operation is implemented in a variety of ways throughout the algorithm makes it prone to bugs and, in the context of this project, difficult to understand and maintain.

The typical working method is that a template control algorithm is developed, maintained for any new type of industrial process and tested against a simulation of said process in-house. Whenever an algorithm is to be commissioned for a real life process, the template is used as a starting point on which the needed customization is applied - often on-site.

The carefulness that is necessary for such modifications may not be in concordance with the large variations in on-site working conditions and testing opportunities may be limited. Specifically, in one algorithm a trivial copy-paste error was found in which variables had not been renamed as intended. Although this may be considered a software bug, to be resolved accordingly, a bug may reside in a part of the algorithm that, tested or not, does not executed during normal operation and then, years later when operation conditions change, it surfaces as a maintenance issue.

	Count	Example
Objects	117	
Lines, total	11845	
Executable code lines	8198	
Executable lines, ex. equal objects	2399	
Loops	183	**for**, **while**
Branching keywords	2444	**if ... then**, **else**, **elseif**, **endif**
Conditional branches	1510	**if ... then**, **elseif** ... **then**, **case** ...
Equations	819	$a \leftarrow b + c$
Assignments, no equation	1804	$a \leftarrow b$
Assignments to 0 or 1	383	$a \leftarrow 0$

Table 3.1: Composition of a typical control algorthm.

Once detected, even a self deploying bug fix may not be appropriate, since a workaround may have been established, either intended or unintended, which may be disrupted by a bug fix. Furthermore, it may be unclear on which sites said bug has been deployed.

In order to quantify some of the findings above, the source code of the algorithms have been analyzed using regular expressions to find and count constructs of interest. The result for a fuzzy kiln control algorithm is shown in Table 3.1. The analysis does not include hard coded objects, such as the fuzzy rules and engine that execute these control laws. In some sense, the algorithm analyzed can therefore be categorized as validation plus pre- and post-processing of signals.

The algorithm at hand contains a total of 11845 lines. Some of the 117 objects though contain code which is 100% identical. By counting these only once the number of lines is reduced to 3544 of which 2399 are executable code, exluding comments, whitespace etc. Of course this indicates that objects are reused to a large degree, but multiple copies of the same code do not improve the maintainability of the algorithm.

The observations listed in Table 3.1 may to some degree be dependent on the language used, in this case Visual Basic. However, it is evident that the amount of branching is large. 1510 conditional branches is a significant part of the total 8198 lines. Moreover, if we assume classical control theory to be implemented by means of equations, we see that only about 10% of the code is implemented equations. We have defined an equation as an assignment statement that includes one or more of the characters “+-*/” on the right hand side.

Additionally, the number of assignments to a value that does not involve an equation is significant. This includes simply assigning the value of one variable to another, effectively routing an untouched signal through the algorithm.

The analysis of other control algorithms shows results identical to those in Table 3.1.

3.2 Control Algorithms as Software

High level control algorithms, regardless of the control paradigm and theory being utilized, are implemented by means of source code. Eventually, this source code is to be inspected and modified during a maintenance scenario. Although a HCS like the one analyzed in the previous section is an enterprise sized application comprising in the magnitude of millions of lines of source code depending on to what extend the remaining SCADA system is included, the actual control algorithm that is customized for the particular process is only 1000-10.000 lines of code.

This code may utilize large external resources such as a Kalman filter or the solver for a Model Predictive Control problem, but such resources are considered black boxes during a maintenance scenario. Eventually, workarounds may even be added to the custom code in order to compensate for unintended behavior of such resources.

From the number of source code lines and their complexity by means of conditional branching and routing of signals, it is evident that a control algorithm represents a significant amount of software, which must be developed and maintained as such. Consequently, much of the work produced in the area of software engineering concerning software maintainability is undoubtedly directly applicable. This could range from concrete technical tools such as coding style enforcers and strict version control, through engineering methods like extreme programming and unit testing to management methods like agile development which all contributes to the development of software with better maintainability. However it is not the aim of this project to research software maintainability in general.

But since control algorithms contains no strings, no user interface, no web clients or database queries etc. they make up only a limited subset of general software. It might therefore be imagined, that some set of methodologies and tools can contribute significantly to solving the maintenance issue with respect to control algorithms while being inapplicable or inefficient for general software. Such contributions may therefore have gained only limited attention from the software society whereas they may represent great potential for the area of control applications.

3.2.1 The Maintenance Process

A typical maintenance scenario involves the following steps:

Listing 3.1: The four steps of control system maintenance.

```
1. Realizing that a problem is present
2. Identifying symptom(s)
3. Tracing cause(s) of each symptom
4. Resolving issue
```

Once the problem is realized, the remaining steps 2-4 are very similar to the process of ordinary debugging. However, since the code is limited to handling signals, the process always involves backtracking the dependency of some variable(s). In case of the example in Section 2.3 this would include answering the following questions, once it has been identified that the *enable fuzzy controller* flag went false:

1. What piece(s) of code changed its value?
2. Which logic conditions was responsible for executing this piece of code?
3. Which signals/variables caused the change of these conditions?
4. Repeat from point 1. until the root cause is found.

This process is far the most difficult and time consuming of the whole process. Resolving the issue once located is on the other hand often limited to adding or modifying a few lines of code. The process of backtracking of signals is investigated further in Chapter 4.

3.2.2 Maintenance vs. Debugging

The time varying behavior of the systems interfacing the control system might activate a part of the algorithm which turns out to have an obvious poor behavior. The issue with this particular part of the algorithm might even have been recognized on other installations after the commissioning and resolved so that future implementations do not include it - i.e. a “bug”.

It might be tempting to explain such maintenance issues as the result of insufficiently debugged software. However, if a bug is defined as failure to meet the software’s requirements specification, the afore mentioned changing environment can be seen as constantly changing requirements. Thus, the term “bug” is rendered ambiguous.

In the case of control algorithms that are customized for every installation, bugs should not be patched straight away as might be the case with other kinds of software. For example traditional office applications can be patched automatically since all copies of the software are identical; hence the consequences of a patch are easily predictable. In contrast, for control systems that are customized for every installation a standard patch might lead to a new and potentially unintended behavior. A work around might have been implemented at commissioning time, intentionally or not, or simply the unique environment of the particular installation will cause new issues.

Thus, the task of performing maintenance remains the same, disregarding whether the issue it originates from can be considered a bug or not.

It is not the intention to question that well designed, well tested, robust software is important in the strive toward reducing maintenance, but it is important to note that once a maintenance issue have arisen in a control system, it does not affect the resolving of the issue whether the cause may be considered a bug or not.

3.3 Improved Tools and Framework

Summarizing, the following shortcomings have been identified for typical control algorithms and the framework in which they are developed:

- Lack of standardized operations;
- Every code block designated to some nominal behavior is accompanied by blocks to handle one or more non-nominal cases;
- Consequently, conditional branching is extensive;
- The mental process of tracing signals is challenging.

It seems that control algorithms have reached a level of complexity that give rise to problems which are well known in the area of software engineering. Some solutions developed in this area may therefore be directly applicable.

Concerning the complex branching and the pattern that a nominal behavior is usually paired with some fall back strategy some solutions have been investigated in order to ease the mental process of tracing dependencies and execution flow. To clarify the branching some sort of graphical presentation of the source code seems reasonable, however, the flexibility of the general purpose programming language must not be obstructed.

Therefore a merge of graphical flowchart-like and text based programming have been suggested. This principle is not fundamentally new though and is offered to some extent by e.g. the product Microsoft Windows Workflow Foundation[1]. Keeping in mind the identified shortcomings, the following concept have been suggested for improvement of the editor at the coding level exemplified by Figure 3.3 which shows an implementation of a simple PID controller. The algorithm is imagined to control the variable `temperature` by adjusting the output `fuelFlow` if the `fuelEnabled` flag is set. Additionally, a performance indicator is calculated by means of the mean squared error during the latest hour.

Disregarding the way of implementation, any variable in a control algorithm can be classified as belonging to one of these 5 categories:

- *input* which is read and cached prior to execution
- *output* which is made publicly available to other systems
- *internal state* which is persisted between executions
- *parameter* which is read-only and may be modified from the outside only
- *field* which is instantiated and reset to a default value prior to execution

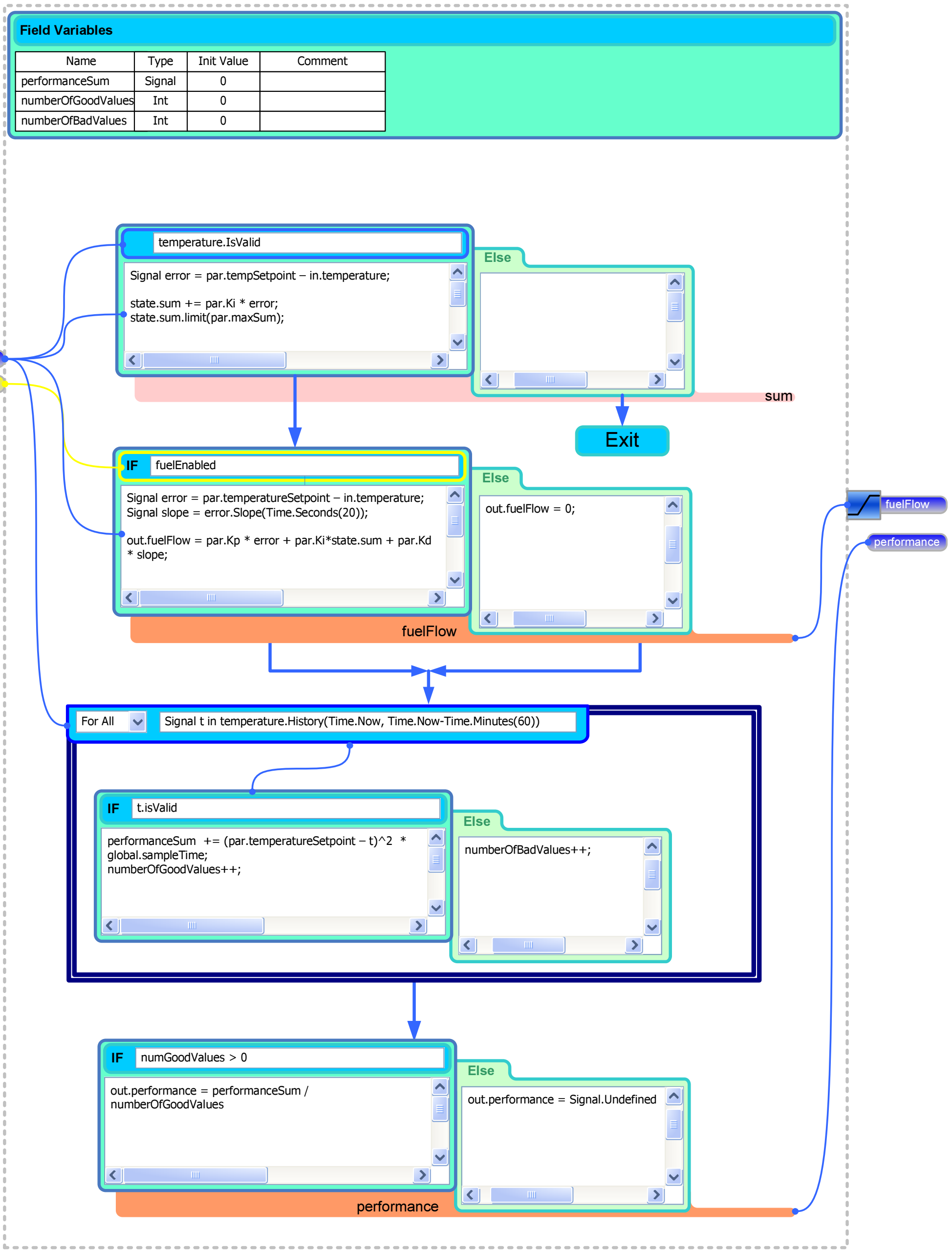
With the suggested concept, a variable's category is stated explicitly. Outputs are persisted and thus represent internal state, the only difference being the public accessibility. Input, output, and internal state variable declarations are not included in Figure 3.3 but are imagined to be similar to Figure 3.2. Field variables are declared in the green box at the top of Figure 3.3.

Programmatically, input, output, parameter and state variables are organized in namespaces so e.g. the state variable `sum` can be accessed as `state.sum`. Execution starts at the top-most code block and flows downwards following the thick blue lines. Each code block features a condition (`If`) that must be satisfied for the block to be executed, `Else` the alternative, right-most block is executed.

Variables declared within a code block are local, i.e. only field variables are shared between code blocks. If an input variable is used in a particular code block, this is indicated graphically by a line drawn from the input to the block and likewise if the input is used in a conditional statement. If a state or output variable is assigned within a block, this is indicated graphically by horizontal outriggers below the block. Input and outputs are color coded according to their type, e.g. boolean, scalar float, matrix etc. Loops of various kinds are presented by a frame around the blocks within the loop. Common post-processing such as slew-rate limiting can be added directly as properties on outputs as illustrated on `fuelFlow`.

During development the programmer must position blocks and connect the lines directing execution flow manually, but lines connecting inputs and outputs and the out-

[1] http://msdn.microsoft.com/en-us/library/ms734631.aspx



Field Variables

Name	Type	Init Value	Comment
performanceSum	Signal	0	
numberOfGoodValues	Int	0	
numberOfBadValues	Int	0	

```
temperature.IsValid
Signal error = par.tempSetpoint – in.temperature;

state.sum += par.Ki * error;
state.sum.limit(par.maxSum);
```

```
IF fuelEnabled
Signal error = par.temperatureSetpoint – in.temperature;
Signal slope = error.Slope(Time.Seconds(20));

out.fuelFlow = par.Kp * error + par.Ki*state.sum + par.Kd
* slope;
```

```
Else
out.fuelFlow = 0;
```

```
For All   Signal t in temperature.History(Time.Now, Time.Now-Time.Minutes(60))
```

```
IF t.isValid
performanceSum  += (par.temperatureSetpoint – t)^2  *
global.sampleTime;
numberOfGoodValues++;
```

```
Else
numberOfBadValues++;
```

```
IF numGoodValues > 0
out.performance = performanceSum /
numberOfGoodValues
```

```
Else
out.performance = Signal.Undefined
```

Figure 3.3: Concept for improved algorithm editor.

riggers indicating state variable assignments are generated automatically by source code analysis.

The framework is extendable, given that specialized blocks can be introduced as needed. For example a suggestion to a block that would solve the problem in Example 1 is shown in Figure 3.5.

A (scalar) input signal is wired to its left side, through an optional, configurable prefilter. At every execution, the execution flow is directed to one of the 3 possibilities (right) depending on the signal being above, within or below specified interval which in this case is a temperature.

Excepted is the case that the signal is either entering or leaving the interval, in which case the execution flow is directed to the corresponding outlet (bottom). This means that bottom outlets are fired only once at the single execution the signal is passing the threshold, while the leftmost outlets are fired continuously at every execution. All parameters, as stated in Example 1, are configured through an appropriate UI.

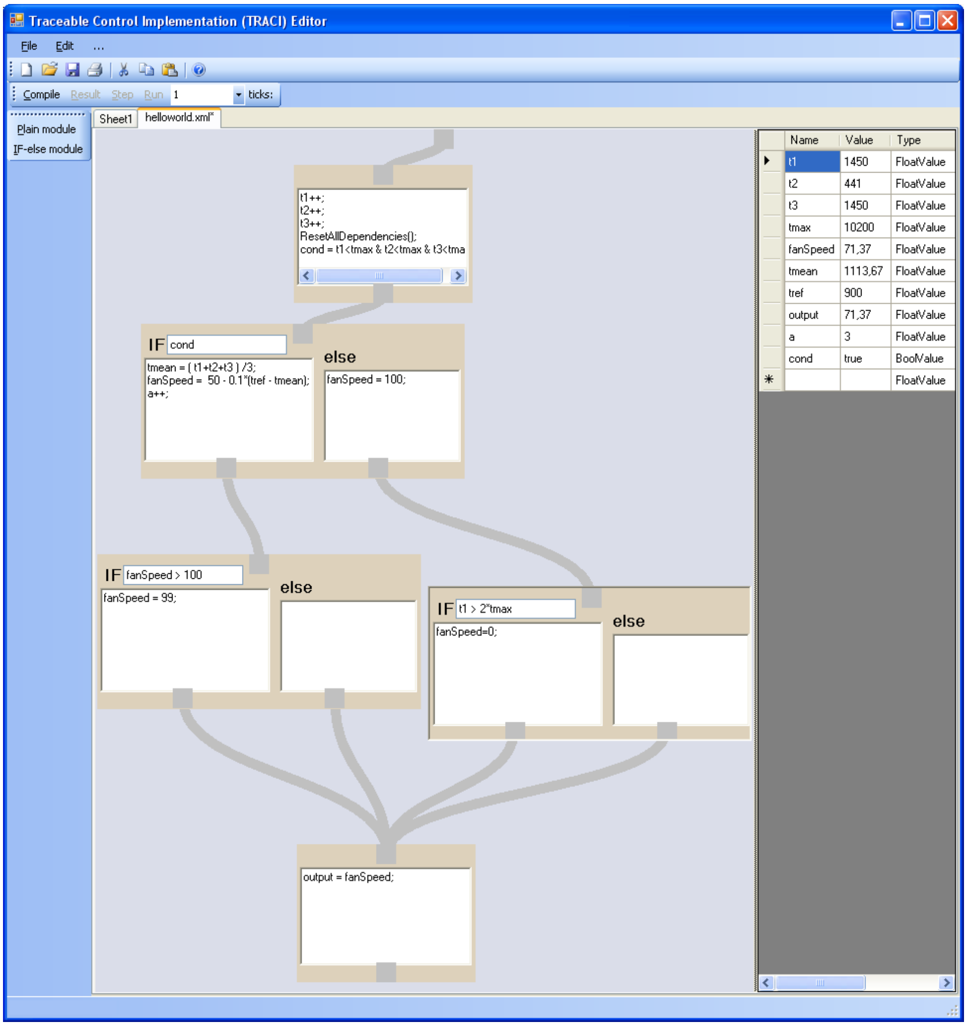
A feature limited prototype including only a basic condition free block and the `if-else` block have been created (See Figure 3.4) and successfully demonstrated to company-internal algorithm experts and commissioning personnel. The prototype allows code blocks to be created and interconnected, proves the resulting code can be compiled and executed, and internal state and output variables persisted between executions.

3.4 Summary

Current frameworks for algorithm implementation are based on a combination of dataflow programming, which is well suited for the signal flow oriented nature of control theory, and more traditional text based multi-paradigm programming which adds the needed flexibility of a general purpose programming language.

Analysis of existing algorithms show that no more than 10% of the source code can be considered implementation of traditional control theory, which could in theory benefit from concepts like robust control and adaptive control to reduce maintenance. The remaining 90% however also needs maintenance. Execution wise, the algorithms use conditional branching extensively. For instance, any operation on a signal is usually associated with one or more conditions to be satisfied for the operations to be executed, and a set of alternative operations if the conditions are not satisfied.

Subalgorithms are encapsulated at several levels in order to abstract from their implementation similar to declarative programming. A negative side effect of this is



```
t1++;
t2++;
t3++;
ResetAllDependencies();
cond = t1<tmax & t2<tmax & t3<tma
```

IF cond

```
tmean = ( t1+t2+t3 ) /3;
fanSpeed =  50 - 0.1*(tref - tmean);
a++;
```

else

```
fanSpeed = 100;
```

IF fanSpeed > 100

```
fanSpeed = 99;
```

IF t1 > 2*tmax

```
fanSpeed=0;
```

```
output = fanSpeed;
```

Name	Value	Type
t1	1450	FloatValue
t2	441	FloatValue
t3	1450	FloatValue
tmax	10200	FloatValue
fanSpeed	71,37	FloatValue
tmean	1113,67	FloatValue
tref	900	FloatValue
output	71,37	FloatValue
a	3	FloatValue
cond	true	BoolValue
		FloatValue

Figure 3.4: Proof of concept for the suggested algorithm implementation environment.

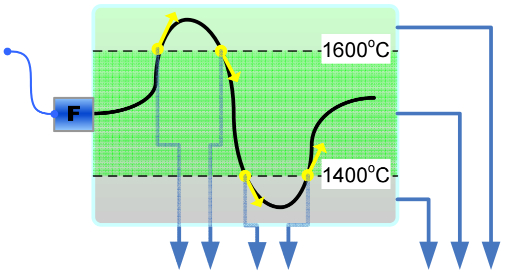


Figure 3.5: Standardized, configurable block for handling interval check of a scalar signal.

that every time a signal crosses the boundary to such an encapsulation the variable carrying its value is likely to have different names on each side of the boundary.

Altogether this means, that for a maintenance expert, it is very difficult to backtrack dependencies between variables/signals since signal identifiers often change, and every operation introduces not only new dependencies but also a conditional dependency that must be tracked down. The backtracking of signals in order to answer questions like "what caused output x to change?" has therefore been identified as the most challenging and time consuming mental act during a maintenance task.

Different implementations of the same common operations have been identified in existing algorithms which has a negative influence on maintainability. Although the enforcing of well established best practices for software development could minimize such maintainability issues, control algorithms pose special characteristics that require specialized tools.

A conceptual algorithm implementation environment has therefore been designed to address these findings. This should enable development of algorithms of increased consistency and maintainability but also reduce the man hours involved when maintenance of the algorithm is eventually needed. A hybrid flow chart style of coding, standardized blocks and limited visualization of signal flow and execution flow will reduce the time needed for the mental process of understanding the algorithm and tracing down the code that need modification. The ubiquitous conditional branching pairing standard operations with alternative operations are handled in a consistent manner.

The concept so far mainly emphasizes the execution flow and the ability to track it. Independent of the execution flow is the signal flow, for which the exact order of operations being performed and the source code performing each operation is uninteresting. For example the concept cannot answer a question like "which variables affect output x?”. However, it shows implicitly that only the source code in blocks upstream of block number two can be responsible, since this is where the variable is last assigned. This issue will be covered in Chapter 4.

Chapter 4

Assisted Monitoring and Maintenance

In Section 3.3 steps were taken to improve the development environment in which the control algorithms are implemented. It emphasizes the execution flow of the algorithm and it enforces common operations to be handled in a standardized manner.

However, it draws only superficial attention to signal flow, which has been identified to cause the most challenging mental act during a maintenance task. This tracking of signal flow is the main topic of this chapter.

Generally, control algorithms are signal processing systems. Measurements are fed as inputs, their values are filtered and validated along with status information, and control actions are decided and written to the output.

An algorithm may be hierarchically structured by well-defined subsystems with explicit inputs and outputs and containing lower level subsystems within higher level systems as illustrated in Figure 4.1. In practice, depending on the programming language, subsystems may be realized by classes, methods or other means of encapsulation.

For the sake of modularity each subsystem has its own name space in order to hide internal variables and only expose inputs and outputs. This means that a single signal, when it is routed through an input of a subsystem into a new name space, or through an output to another name space, will alter its identifier.

This has shown to cause a major challenge for maintenance personnel when backtracking signals trough the system for the purpose of finding the inputs that affect

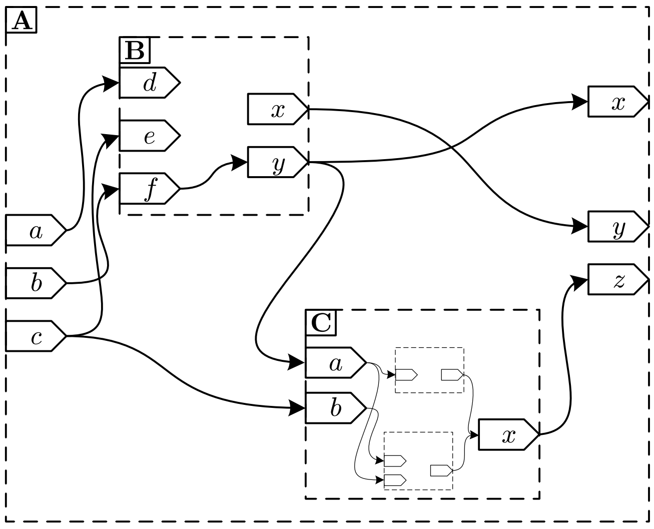


Figure 4.1: Example of signal flow through a control algorithm, with inputs a,b,c and outputs x,y,z. It consists of subsystems $\mathbf{B}$ and $\mathbf{C}$. $\mathbf{C}$ is further comprising two lower level subsystems.

a certain signal e.g., locating the part of the algorithm that modifies a signal, or simply understanding the implementation of an unfamiliar algorithm.

Figure 4.2 shows a simplified example. Consider the output identified by x of the entire control system defining the name space $\mathbf{A}$ in Figure 4.1. Backtracking the signal leads to the output y and input f in $\mathbf{B}$ to input b back in name space $\mathbf{A}$ as illustrated in Figure 4.2. That is to say the process of realizing that x is merely an untouched copy of the input b is a complex mental act.

$$\begin{tikzcd} \boxed{b_{\mathbf{A}}} \arrow[r] & \boxed{f_{\mathbf{B}}} \arrow[r] & \boxed{y_{\mathbf{B}}} \arrow[r] & \boxed{x_{\mathbf{A}}} \end{tikzcd}$$

Figure 4.2: Backtracking output x in Figure 4.1.

The example given here is over-simplified. In case of a real control algorithm a signal may pass through ten or more name spaces, signal processing is performed on the way and generates new signals. Forks may lead a signal in multiple directions and frequently the value of a signal determines executional branches in the algorithm, choosing which sub-algorithms to execute.

The mental process of back- and forward tracking such dependencies may be similar to the one performed during general debugging of software, but it is particularly complex in the case of control algorithms due to their special structure.

4.1 Dependency Graphs

Tracking of a signal, including the operations performed on the signal, can be formalized by means of a dependency graph. Consider Algorithm 1. After execution, c will be *depending* on a and b via the operator "$+$" which in turn is depending on the constants 1 and 2. This can be presented by a graph as shown in Figure 4.3.

Conditional branches in the algorithm affect the dependency graph since statements executed in one branch may produce graphs different from those produced in another branch. Hence, a statement is control-dependent on the condition responsible for its execution. Observing variable y in Algorithm 2 and executing the algorithm with $den \neq 0$ yields the dependency graph in Figure 4.4. The assignment of x in line 2 is control-dependent on the condition in line 1.

Algorithm 1

```
1: $a \leftarrow 1$
2: $b \leftarrow 2$
3: $c \leftarrow a + b$
```

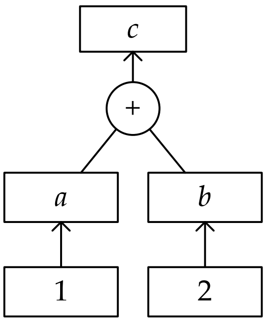


Figure 4.3: Dependency graph for variable c after execution of Algorithm 1.

A change in the branching i.e. $den = 0$, yields a different dependency graph for y. Formally, for any variable y, there exist a finite set of possible dependency graphs

Algorithm 2

```
1: if $den \neq 0$ then
2:     $x \leftarrow \frac{num}{den}$
3: else
4:     $x \leftarrow 0$
5: end if
6: $y \leftarrow x + 1$
```

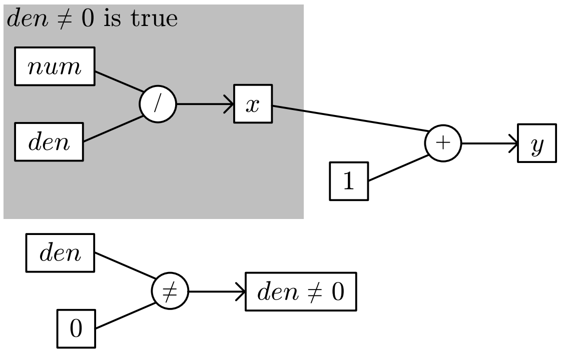


Figure 4.4: Dependency graph for variable y in Algorithm 2 for $den \neq 0$. Shaded area denotes control-dependency on the condition $den \neq 0$.

limited by the number of branching combinations implicitly affecting y by control-dependency.

4.1.1 Dynamic vs. Static Dependency Graphs

Executing the algorithm multiple times with different input and/or internal state, e.g. in a sampled system, may produce different dependency graphs from the fixed set of possible graphs, only representing the active dependencies during the particular execution. Thus, the dependencies are denoted *dynamic*.

Vice versa, a graph including all dependencies that may possibly affect a variable via all possible branching combinations in the algorithm is denoted *static* and equals the sum of the set of possible dynamic dependency graphs.

Dynamic dependency graphs tend to be smaller than the static counterpart and can serve to minimize the amount of data. Additionally, by containing information about the particular execution, such as variable values, dynamic graphs can be used for comparison of executions.

4.1.2 Program Slicing

Program Slicing is a technique that can extract dependency graphs from source code. Static program slicing was introduced by [Weiser 1982; Weiser 1984] in order to identify all possible sources that may affect a variable. [Korel and Laski 1988] introduced dynamic program slicing to produce dependency information only relevant for the actual execution of the algorithm.

Applying this technique to Algorithm 2 we obtain Algorithm 3 which shows the execution path produced from Algorithm 2 when executed with $den \neq 0$. It contains each executed statement together with the line number of its location in Algorithm 2.

Algorithm 3

```
1: $den \neq 0$
2: $x \leftarrow \frac{num}{den}$
6: $y \leftarrow x + 1$
```

4.1.3 Motivation for Program Slicing

Program slicing has been studied since the early 1980's. However, it seems that the number of publications has peaked, and that the technique has not gained widespread application despite its potentials in software maintenance.

Some general purpose development environments, such as the Microsoft Visual Studio series[1], offer debugging and refactoring features which seem to make use of static program slicing. However, commercial applications of program slicing and especially dynamic program slicing are obviously absent and as already mentioned, the publication activity in the field seems to have decreased.

This could indicate that the prospects of any benefit is depleted, which may or may not be true in the case of software in general. Development and maintenance of general software faces a wide spectrum of challenges for which program slicing is of no use, or there may be patterns and practices [Gamma et al. 1995] that break the concept of program slicing. For example, technologies such as late binding, Service Oriented Architecture (SOA) and others which cause much of the code to be

[1] http://msdn.microsoft.com/en-us/vstudio/default.aspx

unknown until runtime will to some extent violate the concept of static program slicing. Furthermore, the tracking of signals and the execution path may be irrelevant for most maintenance tasks.

But control algorithms are, from a software point of view, relatively simple as seen in Chapter 3. While any software patterns may principally be utilized in a control algorithm this does not seem to be the norm. There is no such thing as late binding and SOA, and no strings, parallel execution, user interface, etc.

Rather, the source code consists of only calculations, branching, and method calls and the data types used are mainly scalar and to some extent matrices. Thus, control algorithms comprise only a small subset of general software as depicted in Figure 4.5. This small subset is almost, if not completely, in turn a subset of software

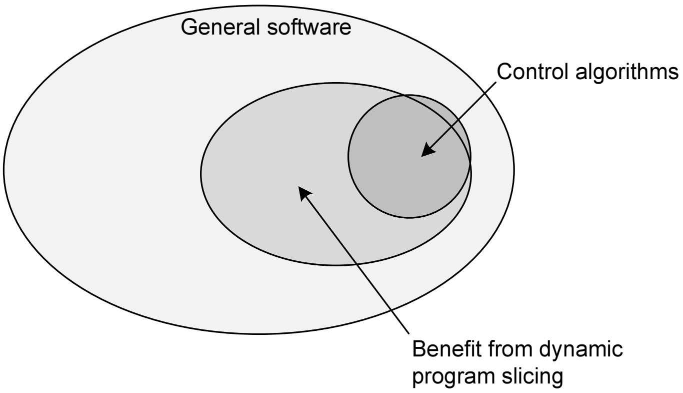


Figure 4.5: Control algorithms utilize only a small subset of patterns in general software, most of which are covered by program slicing.

suitable for program slicing. Moreover, control algorithms have the unique characteristic of being executed periodically as part of a sampled system.

Therefore, while program slicing may provide limited benefit in the maintenance of software in general it is believed to have great potential for the very limited and specialized subset of software that control algorithms represent.

4.2 Assisted Maintenance

It is the intent that the introduction of dependency graphs in the maintenance of process control algorithms should help personnel to locate the failing part of the algorithm and eventually semi-automate the maintenance task.

4.2.1 Monitoring and Detection

Any algorithm will have a limited although potentially large set of possible execution paths. Now let us assume that the algorithm is to be executed periodically and let us then consider each execution path, i.e. branching combination producing a unique sequence of executed line numbers, to be a state in a state machine. During normal operation with high performance the algorithm will stay in a single state or cycle through a limited subset of states as illustrated in Figure 4.6 by the states $n_{1...i}$. In case of an unintended situation the algorithm will likely diverge from this behavior, e.g. move to rarely used states f_{1-2}.

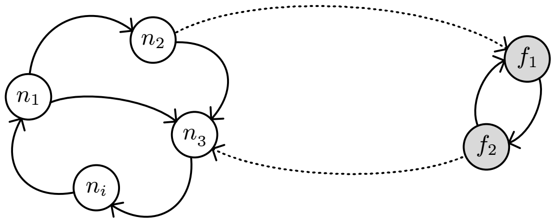


Figure 4.6: State machine representation of execution paths for algorithm during normal operation $n_{1...i}$, and abnormal situation f_{1-2}.

Thus, normal behavior can be identified statistically by means of sequences of states, time used in each state, etc. An alarm can be raised in case of deviation from normal behavior of the algorithm by continuously monitoring the state machine and comparing the behavior to the identified normal behavior.

In the following maintenance scenario, the single conditional branch in the algorithm responsible for producing the abnormal execution path is a natural point of interest, and it is easily located by comparing the execution path to the ones classified as normal. Once located, the dependency graph for the condition can be generated.

The root cause and/or the single statement in the algorithm causing said conditional branch to change will be present within this dependency graph. Finally, a comparison of this graph and one obtained during normal operation will pinpoint a single or few specific statements in the algorithm to be of interest for the maintenance crew.

Thus, the entire process from monitoring, detection and identification of the relevant conditional branch to locating the statements of interest is automated leaving the maintenance crew only the task of verifying the found location of the problem and conducting the proper modifications to the algorithm.

4.2.2 Dependency Graphs and General Assistance

The presented dynamic dependency graph represents a new way of exploring and visualizing existing control algorithms. The dependency graph in Figure 4.2, if generated automatically, may be used to highlight the corresponding signal path in Figure 4.1 instantly visualizing the signal flow after a particular execution.

The raw dependency graph may be utilized with any level of detail, e.g. the graph of Figure 4.4 may be reduced to show only that y depends on the inputs num and den. Being able to instantly identify inputs that affect a variable is particularly useful with control systems having tens or hundreds of inputs. Having recognized a variable behaving unintended, only the affecting inputs/measurements have to be suspected and investigated.

Similarly, entire subsystems may be considered black boxes and incorporated in the dependency graph as such.

If the subsystems of an algorithm are classified according to their purpose, e.g. “normal operation”, “initialization”, “upset recovery” etc. various long term statistics on dependency together with execution paths will yield useful information about the health of the algorithm. For example, if output y is predominantly affected by subsystems classified as “upset recovery” and rarely by “normal operation” subsystems, this will indicate improper tuning of the algorithm and serve as an early warning of reduced performance.

Statistics on dependency graphs for all outputs of a system will yield other useful information such as the most active signal paths and rarely used signals and inputs, indicating dead code.

4.2.3 Comparing Dependency Graphs

Comparing dependency graphs from two or more executions may help to pinpoint statements in the algorithm being responsible for certain actions. Suppose that a discontinuity in the history of an output value m has caught attention during a routine inspection of the trend plots produced during the last two weeks by a running algorithm, as shown in Figure 4.7. The cause of such an event can be investigated by inspecting and comparing the dependency graphs for m at the two executions at sample $k-1$ and k before and after the event. In this example, m is intended to be the mean of 3 input signals.

Suppose the comparison yields the graphs illustrated in Figure 4.8. It is evident that while m is calculated on the basis of a,b and c at sample $k-1$, only c is used at sample k.

From the two dependency graphs the associated snippet of the algorithm responsible for producing the graphs can be located and is shown in Algorithm 4. For instance the difference between the two graphs, being the statements executed at $k-1$ but not k, can be highlighted. It is now clear that the discontinuity in Figure 4.7 was caused by input a and b becoming invalid, changing m from being a mean of 3 inputs to a direct copy of input c.

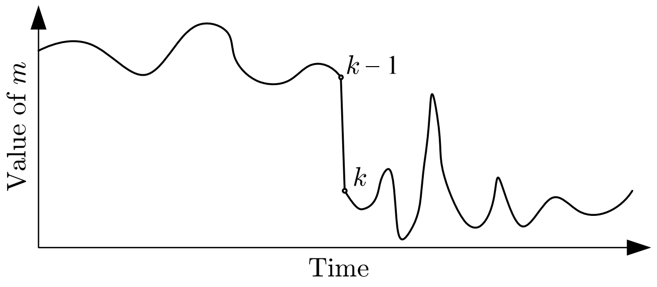


Figure 4.7: Discontinuity in the output m between two executions of the algorithm at sample $k-1$ and k.

The sequence of tasks from recognizing an interesting feature in the history of a signal in Figure 4.7, through generating the dependency graph comparison in Figure 4.8 to displaying the algorithm snippet with highlights or other information like Algorithm 4, can be automated.

Although this example is artificial, the ability to browse dynamic dependency graphs for any variable and to compare graphs recorded during previous executions presents a new way of navigating control algorithms and quickly locate the source code responsible for chosen control actions.

Thus, investigation of cause-effect relations, which was otherwise to be performed manually, can now be semi-automated.

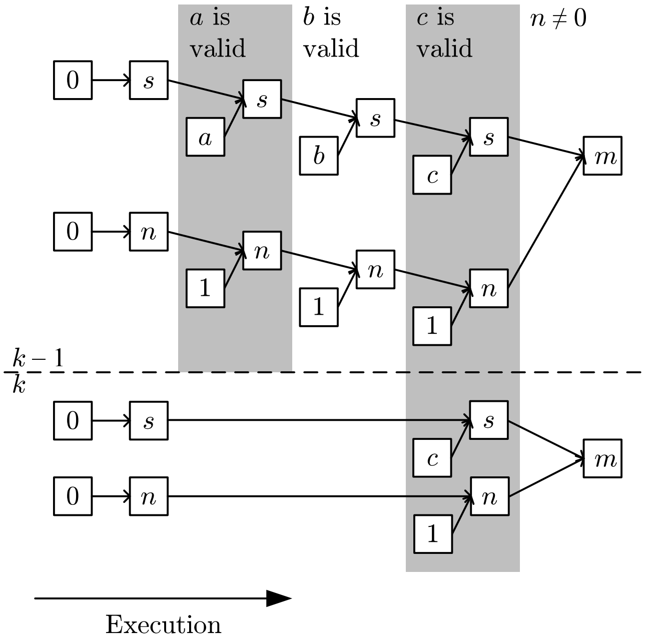


Figure 4.8: Comparison of dependency graphs of m for executions $k-1$ and k. Operators are omitted for the sake of readability.

4.2.4 Limitations

The concept proposed so far of using program slicing to assist maintenance is based on the approach that control algorithms are recognized as software systems.

As discussed in Chapter 2 many control system related issues may be addressed by dedicated solutions. For instance, consider an algorithm containing a P-controller:

Algorithm 4 Algorithm-snippet associated with the dependency graphs in Figure 4.8. Difference between $k-1$ and k is highlighted.

```
1: $s \leftarrow 0$
2: $n \leftarrow 0$
3: if $a$ is valid then
4:     $s \leftarrow a$
5:     $n \leftarrow 1$
6: end if
7: if $b$ is valid then
8:     $s \leftarrow s + b$
9:     $n \leftarrow n + 1$
10: end if
11: if $c$ is valid then
12:     $s \leftarrow s + c$
13:     $n \leftarrow n + 1$
14: end if
15: if $n \neq 0$ then
16:     $m \leftarrow s/n$
17: else
18:     $m \leftarrow m$
19: end if
```

```
1: ...
2: $u \leftarrow K_p \cdot (r - y)$
3: ...
```

with reference r, gain K_p, control signal u and controlled variable y. Under certain circumstances the maintenance issue of tuning K_p can be resolved by the addition of an adaptive tuning algorithm ensuring that K_p is always within an interval of acceptable controller performance. In comparison, the assisted maintenance concept proposed so far would merely lead the attention to line 2 in the above algorithm and leave the control expert to manually adjust the parameter *after* K_p has left the acceptable interval.

However, the source code of the adaptive control algorithm itself may be subject to maintenance too, in which case assisted maintenance will supply the same help in locating the responsible code lines. Thus, the proposed assisted maintenance concept is disregarding the kind of control theory implemented.

The benefit is expected to be high when the challenge in performing maintenance is related to signal flow tracking, which is the case when the control algorithm source code is within the set depicted in Figure 4.5. As the source code of the control algorithm at hand is leaving this set, the benefit must be expected to decrease. Furthermore program slicing is addressing the maintenance issue at the source code level. If the maintenance problem does not originate within this domain the proposed concept will be less useful.

4.3 Proof of Concept

The existing HCS has been extended to generate dynamic dependency graphs for variables in existing algorithms based on program slicing. Dynamic program slicing is implemented in order to explore the possibilities of comparing dependency graphs as suggested in Section 4.2.2.

In addition to a dependency graph for all variables at every execution, the execution path, i.e. the sequence of statements executed, is recorded. While a real control algorithm is controlling a simulated process, data generated by the program slicing is stored for later analysis in order to investigate the proposed maintenance tools.

Continuously improving slicing algorithms have been developed, e.g. [Korel and Laski 1988; Agrawal and Horgan 1990; Mund et al. 2002; Mohapatra et al. 2006; Korel and Rilling 1998; Binkley et al. 2006]. Generally, algorithms may have different computational complexity in terms of CPU load and memory consumption both at runtime while the program is executing and slicing data is collected, and at extracting dependency graphs from this collected data.

In the case of applying program slicing to existing control algorithms the computational complexity must be least affected at runtime since an increased computational delay will affect the closed loop performance. The cost of later, possibly offline, analysis of the slicing data is less important. The memory consumption at runtime is also of little concern. What particular slicing algorithm to apply will depend on the programming language used for implementing the control algorithm and possibly other practical factors. For example [Mohapatra et al. 2005] propose an algorithm to handle concurrent, object oriented programs; [Mohapatra et al. 2006] an algorithm dedicated to Java[2] and [Pócza et al. 2006] a method for the Microsoft .NET framework[3].

4.3.1 Implementation

Theoretically, the runtime collection of variable values etc. can be implemented as a part of the engine executing the source code, which may require compilation or not; otherwise the output of the compiler may be modified to collect these data as demonstrated by [Pócza et al. 2005].

However, in case of the control algorithms and HCS at hand this is impractical. Instead, the source code of the control algorithm is modified in a manner that probes but does not affect the behavior of the original algorithm. At selected locations in the source code, statements are inserted to store the value of selected variables at that particular step in execution. For a detailed example see Section 5.1.

Figure 4.9 shows a simplified layout of the extension. The control algorithms at hand are implemented in a Visual Basic like language (VB) being executed by a third party proprietary engine. During initialization of the HCS the various object in an algorithm (See Figure 3.1) are loaded from storage, compiled and kept in memory for execution at every sample.

Under normal circumstances the loaded source code is passed on to the compiler. This is bypassed by the extension which inserts additional statements in the code before passing it on to the compiler. When the modified code is executed the inserted statements will send slicing relevant data for the particular execution at sample i to storage.

If the extension is not present or if the code modification should somehow fail, the original code is compiled as usual. Thus, the extension is applied in a transparent and minimally invasive manner and only the loader module is aware of its existence.

[2] http://www.java.com

[3] http://www.microsoft.com/NET/

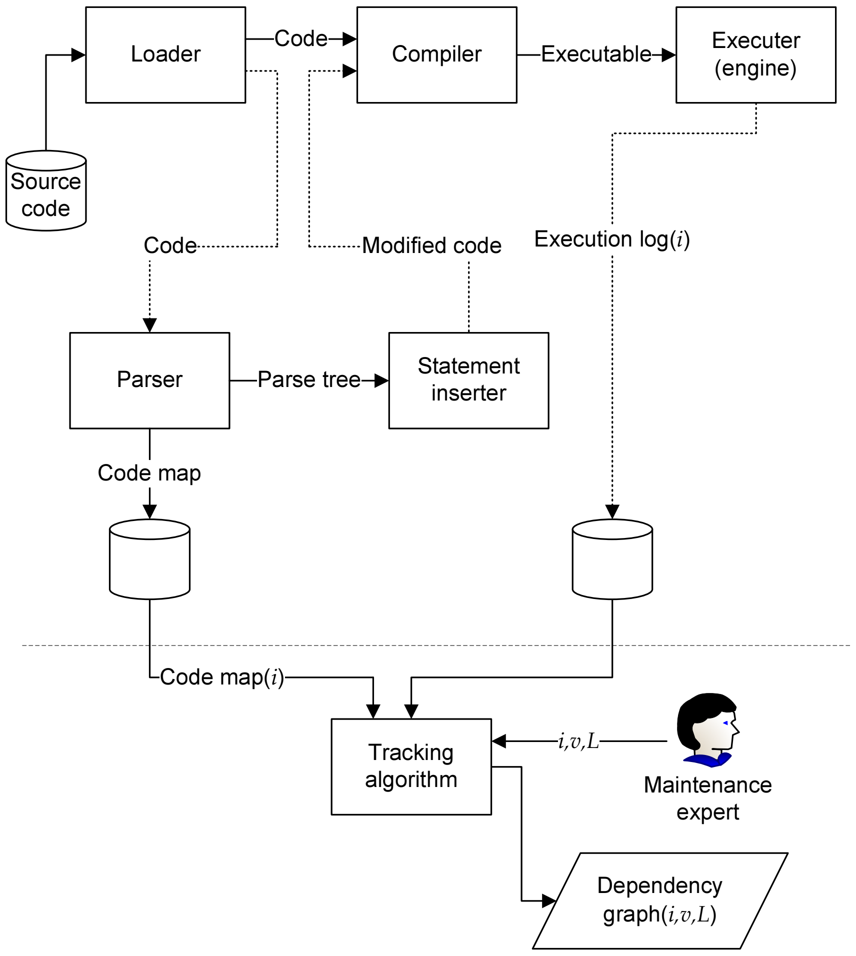


Figure 4.9: Simplified structure of the proof of concept extension to the HCS.

A parser for the used language is needed to locate statements in the source code. Surprisingly, no off-the-shelf parser was available for VB despite its widespread use. Therefore a custom parser has been made. Beside the usual parse tree, the parser creates a *code map* including a graph of the static inter-variable dependencies of the source code. Since the source code may principally be modified and re-compiled between executions it must be possible to later associate an execution log with its matching code map.

From the stored code maps and execution logs dependency graphs can be extracted by a tracking algorithm by supplying the variable name in question v, the location (step in the execution path) in the algorithm L as well as the sample number i. This information may be supplied either by the maintenance expert through a proper user interface (See Chapter 5) or by external software for further analysis.

The application extension has been named DynaTrace and is written in .NET C# and comprises approximately 26,000 lines of source code.

4.3.2 Usage

This proof of concept demonstrates the possibility of logging sufficient slicing information in order to generate a dynamic dependency graph for every variable at every execution of the control algorithm within the logging horizon. With the extension such logs can be generated while real, full scale control algorithms are running and the logs made available for further analysis. A positive side effect of this is the ability to replay the exact execution of the algorithm since enough information is stored to recreate the value of all variables at any step in the execution path.

Two conceptual uses of the logs have been investigated. First, a simple user interface enables visualization of dependency graphs in a variety of ways which is shown in Chapter 5. These visualizations are intended to demonstrate the use of dependency graphs as an assisting tool during maintenance as presented in Section 4.2.2.

Secondly, the possibility of online monitoring and detection, as proposed in Section 4.2.1, can be investigated further which is discussed in Chapter 6.

Ultimately this is addressing the identified issues of control algorithm maintenance by enabling early warning and detection of a fault and then assisted maintenance by means of guidance to locate the source code lines responsible for the fault.

4.4 Replay of Execution Without Program Slicing

The above presented implementation of program slicing can be relatively complex. However, the ability to replay the execution of the algorithm can be achieved in a much simpler manner. In order to use this method the algorithm must fulfill two requirements: it must be deterministic and time-invariant. If so, the output of the algorithm will depend on only 3 quantities: Inputs, parameters, and internal state as illustrated in Figure 4.10.

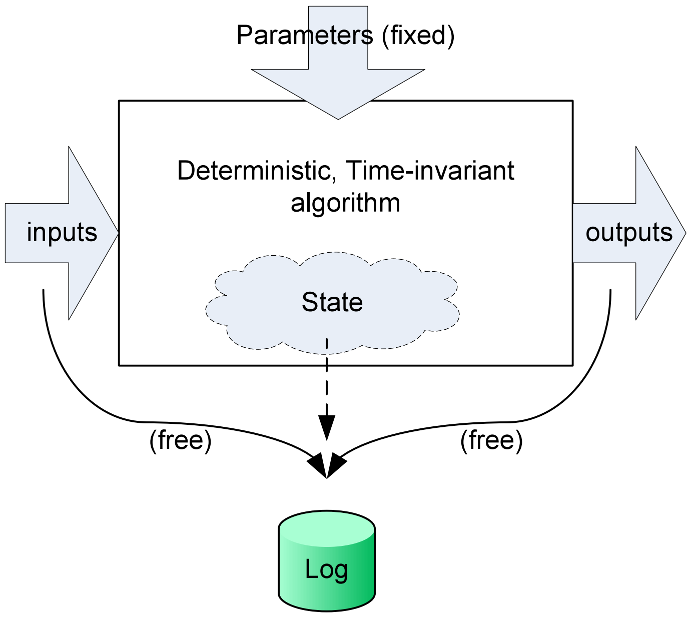


Figure 4.10: The output of a deterministic and time invariant algorithm will depend only on its input, parameters and internal state.

Thus, if these quantities are logged at each sample after execution of the algorithm, later replay is possible for investigating a previous event or even post mortem after a catastrophic failure. This method does not enable single stepping replay explicitly however, such functionality may be supplied by third party development environments depending on the programming language used.

The Inputs to the control algorithm should already be logged by the SCADA system and therefore they come at little or no cost. Likewise, the outputs (setpoints) are logged as depicted in Figure 4.10 and enables verification of the replay in that it should produce the same output values as the original execution. The parameters of the algorithms are fixed per definition so no logging is needed. Therefore, only the internal state remains. An example of internal state could be the value of the integrator in a PID controller.

Since it is desirable that such a value is persisted during a restart of the algorithm or the computer, it may already be the case that the internal state is, if not logged, then at least persisted in storage after each execution. Depending on the control system framework at hand the additional cost of enabling this kind of replay will therefore be minimal. There is no computational cost at runtime and only little or no extra data must be stored after execution.

Practical violations of the requirements of determinism and time invariance include any use of the computer's internal clock and the use of pseudo random number generators [Knuth 1997] (e.g. for dither generation) unless its seed is considered to be e.g. a parameter or an input which is logged. This also implies that e.g. a successive optimization sub-algorithm must not be made to bail out after a certain timeout. Rather, a maximum number of iterations or other means that are not depending on processor speed or load must be used. Any sound control algorithm should fulfill these requirements though since it is desirable in many other aspects that an algorithm behaves consistently and independent of hardware and for instance is executable in non-real time.

Ultimately, static program slicing could be utilized to identify variables that are members of the state automatically and verify the determinism and time invariance requirements. Since the algorithm output depends only on input, parameters and state, the leafs of the static dependency graph for each algorithm output will belong to one of these groups. By enumerating all leafs and excluding inputs and parameters, only two possibilities remain: either state variable or a violation of the determinism and time invariance requirements.

In contrast to the program slicing based replay, this method is conceptually different since it *reproduces* an execution of the algorithm while the former records and replays the actual execution. Moreover it does not enable the generation of dependency graphs or execution path although these could in theory be produced by applying dynamic program slicing offline during replay.

This method may therefore be an option in case the runtime, computational cost of dynamic program slicing is unacceptable. Since the aim is to study the use of dynamic dependency graphs and execution paths, this method has not been investigated further.

4.5 Summary

The process of backtracking a signal and its dependencies through an algorithm is very challenging and time consuming. This chapter has shown that the information sought in this process can be formalized and presented as a so-called dependency graph.

Furthermore, such a graph can be extracted automatically by using a method known as dynamic program slicing. Thus, a question like “what inputs affected variable x during this execution of the algorithm?” can be answered automatically.

Dependency graphs can be used in the process of locating the source code responsible for unintended behavior of the algorithm during a maintenance task and this functionality can be integrated in the algorithm development environment. Furthermore, a method of monitoring the algorithm during runtime is proposed, which analyses the collected dependency graphs to raise an alarm once the algorithm behaves abnormally.

The use of program slicing poses no additional requirements to the control algorithm or the control engineer during development and installation which makes the solution attractive for industrial applications.

An implementation of the program slicing technique has been integrated with a commercial high level control system in order to extract dependency graphs from full-scale, running, high level control algorithms. This allows further study of the proposed usage of dependency graphs on real-world control algorithms.

Chapter 5

Assisted Maintenance Demonstrated

When conducting algorithm maintenance, experience shows that performing the actual modifications of the algorithm or identifying the parameter that needs retuning is a simple task, while recognizing the problem, understanding the root cause of the malfunction, and locating the responsible part of the algorithm is time consuming and is what requires all of the expert knowledge.

In this chapter we present a novel concept for alleviating the human expert in performing this task. Despite the kind of control theory being applied in any given control algorithm, it will be implemented by means of source code. Possibly multiple different programming languages are used, and the source code may by organized in special logical structures based on signal flow, execution order etc. dedicated to solve the control problem at hand.

Inevitably, maintenance personnel must navigate within this source code during the process of identifying the problem, which turns out to be challenging for a number of reasons. Foremost, control algorithms tend to be particularly complex by means of signal flow. For example, answering a question like “which variables affect the value of this output?” is a time consuming task of backtracking assignments of said variables through the source code.

The algorithm may be heavily customized for a single process facility, by personnel who are not available for the current maintenance task, and even though documentation of the performed customization exists, it will only cover the intended behavior of the algorithm and will therefore be of little help in investigating the current malfunction.

The concept demonstrated here enables maintenance personnel to quickly diagnose abnormal behavior of a running control algorithm, to explore only the relevant parts of an unfamiliar algorithm and to locate the responsible part of the algorithm that needs modification. The early concept was presented in [Hansen et al. 2008], introducing the term “assisted maintenance” addressing the large segment of maintenance related issues that requires human interaction. Dynamic Program Slicing [Weiser 1982; Korel and Laski 1988], a method known from computer science, is the core of the concept used to record details about the execution of a given algorithm. The recorded data is then processed and presented to the user (maintenance personnel).

The control algorithm is not required to be designed within a dedicated language, it is not needed to model the plant or the algorithm, and nor does the concept need parameters to be tuned. This makes the concept attractive for industrial applications since existing design- and implementation procedures are not affected. The solution can even be applied to existing, implemented algorithms without modifications to their source code.

Thus, the overall aim is to let the human expert do what she or he is good at, being to conduct the actual modifications to the algorithm and tune parameters, and leave everything else to the computer.

5.1 Dynamic Variable Tracking

A special technique have been developed to automatically track dependencies between variables, and record every aspect of the execution of an algorithm, e.g. which code lines that are executed, the value of all variables at the execution of this particular code line, etc.

In the following, we will only consider ordinary, sequential algorithms consisting of statements, equations, conditional branches, loops and methods etc., i.e. algorithms that are implementable in a C-like programming language. We also assume the algorithm to be executed at fixed intervals as part of a sampled system. The technique involves the following steps:

1. Pre-compile analysis of the source code;
2. Automatic insertion of data collecting routines into the source code based on this analysis;

3. Compilation/interpretation of the modified code;
4. Execution of the modified code. Inserted routines record data;
5. Post processing of recorded data.

The initial analysis generates a *code map* which is a data structure representation of the code, by identifying keywords, statements, method calls, equations, etc., and is similar to the initial steps performed by a compiler. Having located all equations and identified the variables used in each equation, routines are inserted in the original code to record the value of these variables. Two routines are inserted, one to record the variables on the right hand side of an equation before it is executed, and one to record the resulting left hand side.

Consider the code snippet listed in Algorithm 5. Algorithm 6 then shows the same code with the recording routines inserted. The location in the original code, in this case the line number is stored as a reference together with the values of the respective variables. Note that the behavior of the algorithm is not affected.

Algorithm 5 Original code.

```
1: $c \leftarrow a + b$
2: **if** $a < b$ **then**
3:     $d \leftarrow c \cdot (a + 1)$
4: **else**
5:     $d \leftarrow c \cdot (e - 1)$
6: **end if**
```

If the algorithm is executed with the initial condition $a = 1$ and $b = 2$ the following record will be produced:

$$
\begin{array}{lr}
\mathbf{R} = & \\
\{1, & (1,2)\} \\
\{1, & (3)\} \\
\{2, & (1,2)\} \\
\{3, & (3,1,1)\} \\
\{3, & (6)\}
\end{array}
$$

with each entry keeping the line number and the values of the now evaluated variables.

In combination with the code map, the record is arranged into a compact table of the lines executed and the recorded variable values as listed in Table 5.1. Since

Algorithm 6 Code with recording routines inserted.

```
1:  $\mathbf{R} \Leftarrow \{1,\ (a,b)\}$
2:  $c \leftarrow a + b$
3:  $\mathbf{R} \Leftarrow \{1,\ (c)\}$
4:  $\mathbf{R} \Leftarrow \{2,\ (a,b)\}$
5:  **if** $a < b$ **then**
6:      $\mathbf{R} \Leftarrow \{3,\ (c,a,1)\}$
7:      $d \leftarrow c \cdot (a+1)$
8:      $\mathbf{R} \Leftarrow \{3,\ (d)\}$
9:  **else**
10:     $\mathbf{R} \Leftarrow \{5,\ (c,e,1)\}$
11:     $d \leftarrow c \cdot (e-1)$
12:     $\mathbf{R} \Leftarrow \{5,\ (d)\}$
13: **end if**
```

the code snippet in Algorithm 5 does not contain loops or method calls, no line is executed more than once.

The code map derived from Algorithm 5 is seen in Table 5.2. For each variable is mapped the line number of equations modifying it, all variables on the right hand side of such an equation, and the list of conditions to be satisfied for the line to be executed.

The combined information of the code map in Table 5.2 and the execution record in Table 5.1 is sufficient to generate a *dependency graph* for any variable during the particular execution. For example, consider the variable d after executing the algorithm: The last assignment to d was on line 3, which, according to the code map depends on c,a and the constant 1. In addition, this line is *execution wise* depending on line 2 which in turn depends on a and b and so forth. The resulting dependency graph for d is shown in Figure 5.1.

Although not shown in the figure, values of all variables are also available. From the graph it can be concluded that at this execution, e.g. d only depends on the inputs a, b and the constant 1. Executing the algorithm with different initial conditions may render a different dependency graph.

Table 5.1: Execution record for Algorithm 6 with $a = 1$ and $b = 2$.

line	dependencies	result
1	$a = 1, b = 2$	$c = 3$
2	$a = 1, b = 2$	
3	$c = 3, a = 1, 1 = 1$	$d = 6$

Table 5.2: Code map of Algorithm 5.

variable	line	dependencies	condition
c	1	a, b	root
$a < b$	2	a, b	root
d	3	$c, a, 1$	root, line 2 is true
	5	$c, e, 1$	root, line 2 is false

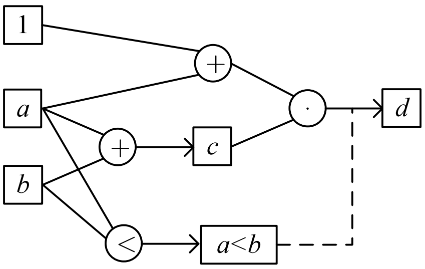


Figure 5.1: Dependency graph for variable d. Dashed line indicates execution wise dependency.

5.1.1 Performance

The presented solution is designed to have a low computational cost at runtime, in order to reduce the effect on the control system. The initial analysis of the source code needs to be done only once at compile time, and its cost is therefore less important. Also the post processing of the record $\mathbf{R}$, in order to generate dependency graphs, is of no concern.

Considering an algorithm with N lines and M variables, the inserted recording statements adds a worst case cost of $O(NM)$ in execution time as well as memory consumed for storage of the record.

The current method of recording the value of any variable x on the right hand side of an equation is redundant, since the value will be available at the latest assignment to x or as the initial value of x. However, the method serves as a consistency check. If the value of a variable recorded just before it is used on the right hand side of an equation is not matching the value last assigned to the same variable earlier during the execution of the algorithm, there must have been a malfunction in the dynamic variable tracking. If this consistency check is omitted, runtime cost of both execution time and memory is reduced to $O(N)$ indicating that the solution is well scalable.

5.1.2 Proof of Concept

The solution has been implemented using the .NET framework and integrated with a commercial platform for high-level control system implementation on which the control algorithms are written in a Visual Basic like language. This has been chosen only because a large number of existing algorithms are available for future testing. In order to work with different languages, a parser capable of generating the initial code map must be implemented for each particular language.

The solution is non-invasive in the sense that existing algorithms needs no modifications. The source code analysis and modification is done automatically as a part of the compilation/interpretation process and it is therefore invisible for the control system developer as well as the rest of the control system. Only the execution time is affected as described above, which should be of less concern as long as it is well below the execution frequency of the control algorithm. The increase in computational delay seen in practice has been insignificant.

5.2 Visualizing Recorded Data

This section demonstrates some of the features of the current proof of concept. The recorded information can be utilized to supply debugging features that are dedicated to control algorithms and are beyond what is available in today's software development environments.

Figure 5.2 shows an example of a small algorithm controlling a SISO system with the control signal `WriteOut` and output from the SISO system `ReadIn` which is controlled to follow the reference `r`. The SISO system is non-linear, so two different gains `k1` and `k2` are used (lines 4-8). Finally, the control signal `u` is limited to $\pm$`umax` on lines 10-15. The algorithm is executed periodically in a sampled system.

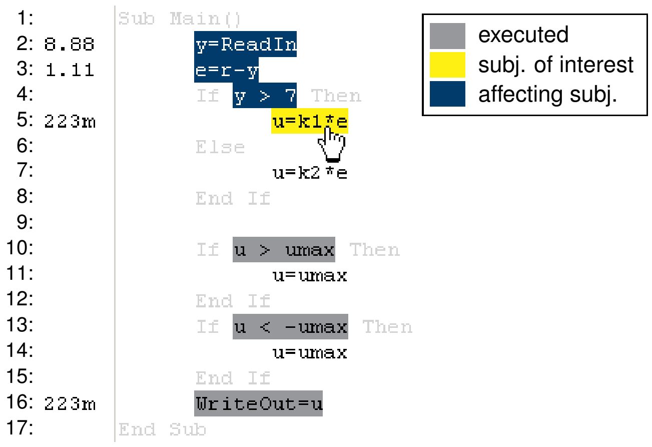
In Figure 5.2 some of the recorded data after a particular execution is visualized. Source code not considered by the mechanism is faded out, while actually executed code is highlighted in light gray. If a line contains an assignment of a value to a variable, its new value is shown to the left. Note the use of decadic prefixes to ensure a fixed-length notation.

This information is helpful e.g. when tracking the sequence of calculations leading to the value of a variable of interest. Similarly, the values of variables and parameters appearing on the right hand side of equations could be displayed. Code statements being executed multiple times e.g. within a loop or in the body of a method called multiple times, may have information about each execution displayed in a similar manner.

If a particular code statement is of interest (yellow line 5 in Figure 5.2) the sequence of statements (blue) affecting its execution can be highlighted. This sequence also have a graphical representation as shown in Figure 5.3: `u` depends on the parameter `k1` and the variable `e` which in turn depend on `r` and `y` and so forth. The execution order is linear from left to right and execution wise dependencies are shown as alternating shaded areas in the background. e.g. the assignment of `u` is execution wise depending on the value of the statement `y > 7`.

If for instance the value of `u` have been found to be incorrect, it is easy to realize that only the input signals `r`, `ReadIn` and the parameter `k1` is to be investigated further.

Figure 5.2 and 5.3 show data at a particular execution of the algorithm. Since the data are recorded at every execution, this log enables both online and offline inspection. Such a log offers an exact replay of the algorithm, and can e.g. easily be send overseas to be analyzed by experts. Trends are available for every variable, as shown in Figure 5.4. The red vertical line represents “now”, corresponding to the particular execution shown in Figure 5.2. Note that in an off line situation future values may be available and are plotted as well.



```
 1:        Sub Main()
 2: 8.88       y=ReadIn
 3: 1.11       e=r-y
 4:            If y > 7 Then
 5: 223m          u=k1*e
 6:            Else
 7:               u=k2*e
 8:            End If
 9:
10:            If u > umax Then
11:               u=umax
12:            End If
13:            If u < -umax Then
14:               u=umax
15:            End If
16: 223m       WriteOut=u
17:        End Sub
```

Figure 5.2: Tracking data shown as highlighting of source code.

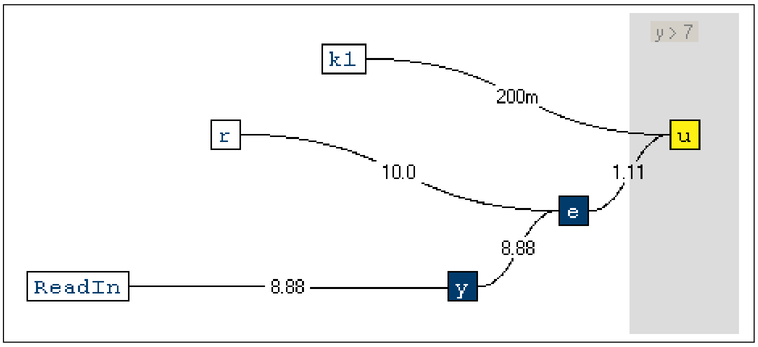


Figure 5.3: Dependency graph for variable u.

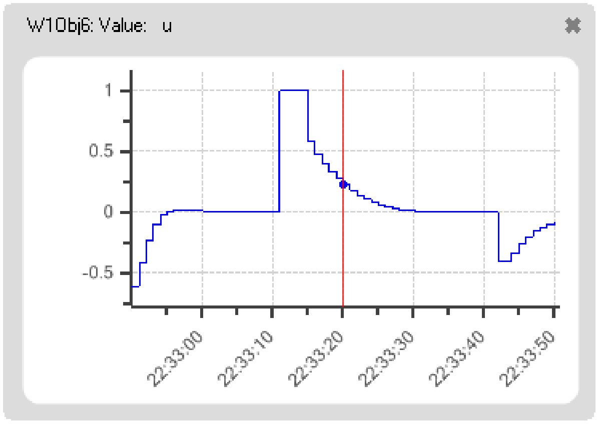


Figure 5.4: Example of trend plot available for any variable.

Naturally, the log can be searched and replayed forwards and backwards, faster than real time or a single step at a time (See Figure 5.5).

While replaying, the code highlighting with variable values in Figure 5.2, and all trends as in Figure 5.4, updates accordingly.

The tools incorporated in the proof of concept are presented here to demonstrate some of the opportunities that are enabled by adding dynamic variable tracking to the control system. It is important to distinguish these tools from those offered by many software development environments such as the ability to "watch" variable values

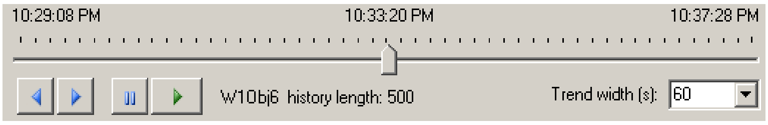


Figure 5.5: Panel for controlling playback of recorded data.

while single stepping through the code. The data recorded here enables an exact replay of every detail of the algorithm, including “watching” all variables, after it has been executed at real time in its natural environment, controlling the physical system. Hence, the system automatically decides what data to record.

Furthermore, the dependency graphs and trend plots for any variable etc. and the ability to go back in time and investigate an anomaly that happened a week ago, are dedicated to the special challenges related to maintenance of control algorithms as described earlier.

The length of the history of recorded data is only limited by the storage capacity available.

5.3 Assisted Maintenance

The one-to-one relation between a point in a trend plot, a node (variable) in a dependency graph, and the source code that produced them provides a powerful tool for locating the code that needs maintenance.

The following example demonstrates some of this potential. Consider the algorithm in Figure 5.2 controlling the throughput of an industrial process, causing the problem shown in Figure 5.6. During installation, the algorithm have been designed, tuned and tested to operate the system output `y` in the interval 6-10. After some time, this must be changed down to the interval 5-8 e.g. due to changed market demands. It is evident from Figure 5.6 that the performance of the algorithm in this new interval is unacceptable, since it causes the system to oscillate and not reach the set point $\mathtt{r} = 5$.

Knowing that the algorithm uses gain scheduling by means of two different gains `k1` and `k2` depending on the value of `y`, it is obvious to suspect that one of the gains is unfit for operating the system in the new interval. However, using dynamic variable tracking, the true responsible part of the algorithm can be located.

Inspecting the trend plot of the control signal in Figure 5.7 around the time frame of the unintended behavior shows a switching pattern. It is therefore interesting to compare the dependency graph for the signal at a sample of intuitively “correct” behavior with one of “incorrect” marked m and n respectively in Figure 5.7. The graphs are shown in Figure 5.8. The difference between the two graphs is evident. The single end node of the graph in the malfunctioning case links directly to line 14 in the algorithm (Figure 5.2) where closer inspection reveals a classic copy-paste error: the line should contain “`u=-umax`” rather than “`u=umax`”. With this error corrected, the algorithm now performs as intended, as shown in Figure 5.9.

Thus, the malfunction was not related to the gain scheduling as would be a natural assumption to investigate initially. Rather, the maintenance system guides the investigation in the right direction.

Note that this means of computer assisted maintenance did not identify the cause of the problem itself, being a missing sign. It merely suggested where in the algorithm to look, which could be deeply within the earlier mentioned 1000-10,000 lines of source code. In practice, it means that the maintenance personnel can be led directly to the root cause of the problem by a few mouse clicks in the trend plot of a signal.

Essentially, the example demonstrates the ideal allocation of tasks between human and machine: The human selects the samples m,n using qualities that are difficult if not impossible to automate, such as intuition, heuristics, or common sense. In our case, sample m seems “correct” since the signal around it is continuous, and sample n seems “incorrect” because it comes after a discontinuity in a period where the signal seems to be in saturation (See Figure 5.7). From here, a machine performs the complex but systematic task of tracking the selected samples back to the source code and leaves the human to realize and fix any errors in the found code.

5.4 Summary

Maintenance personnel must explore and navigate within a possibly unfamiliar control algorithm implementation in order to locate the parameter or piece of source code responsible for the reduced performance. This has shown to be a difficult and time consuming challenge, while readjusting the parameter or modifying the source code once the problem is identified is relatively simple.

A novel concept has been proposed and demonstrated in this chapter to supply computer assisted maintenance. The solution helps by semi-automatically locating the root cause of a malfunction and it supplies a set of tools for enhanced exploration and understanding of the algorithm. This is done by automatically recording all data needed to replay any historical execution of the control algorithm.

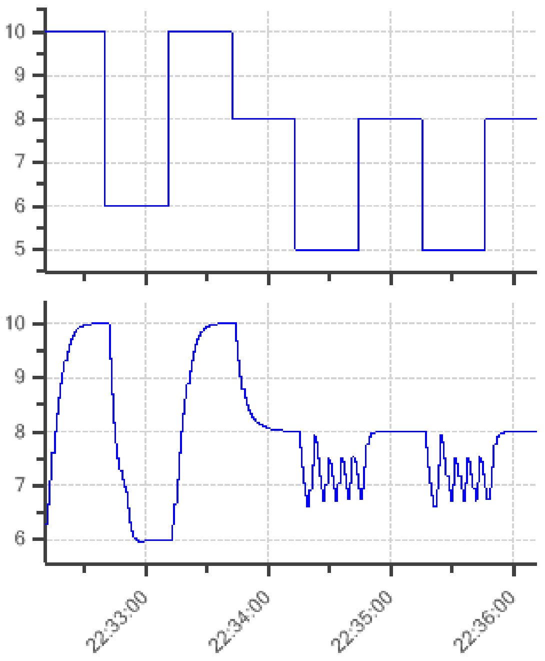


Figure 5.6: Trends of reference `r` (top) vs. process output `y` (bottom).

The only cost of the solution is a slightly increased execution time, and the storage space needed for the record of past executions. The solution is tuning-free, e.g. no model of the process or of the algorithm itself is needed, it does not require the control algorithm to be implemented in a dedicated framework or using special tools and it can be applied to existing algorithms without modification. The solution is therefore well suited for industrial applications.

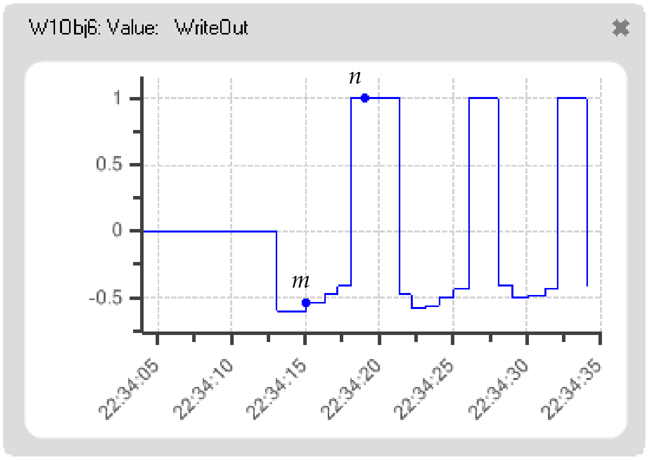


Figure 5.7: Trend of control signal `WriteOut`.

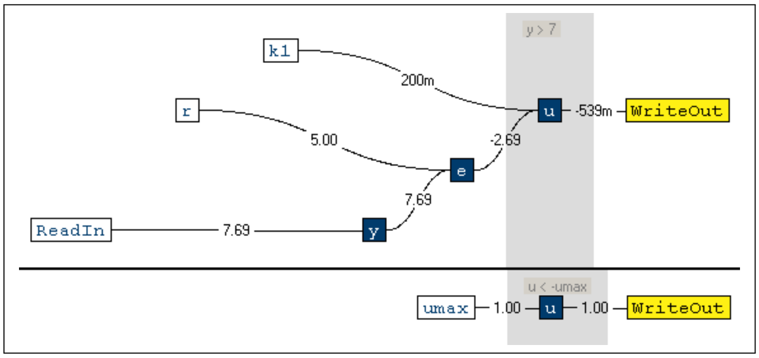


Figure 5.8: Dependency graphs for control signal. Top: correct, sample m. Bottom: incorrect, sample n representing lines 13-16 in Figure 5.2.

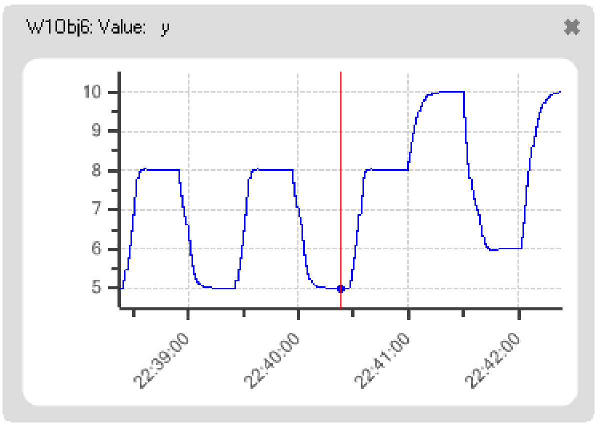


Figure 5.9: Trend of y after correction of algorithm. The system can now be operated in the full interval of $\mathrm{y} = [5; 10]$.

Chapter 6

Supervision of Running Algorithm

As proposed in Section 4.2.1 the execution path of the control algorithm can be interpreted as a state in a finite although large state machine. Hypothetically, the behavior of the algorithm in this state space could hold information about its performance and therefore be used for supervision and diagnostic purposes.

A series of experiments have been conducted to investigate the potential of such measurements.

6.1 Test Setup

FLSmidth has a simulator in its portfolio that is capable of simulating all processes in a cement plant, including process dynamics and logic.

This simulator is utilized in developing new control algorithms, so that every algorithm is first made to run against the standard simulator. Customized algorithms for real world plants are then derived from this basic algorithm. Therefore, control algorithms that fit the simulator exist for all major applications.

A combination of full scale control algorithm and this simulator is therefore the scenario closest possible to the real-life application of an algorithm on a physical facility. DynaTrace have therefore been integrated with the process control framework of FLSmidth in order to collect data from such simulations. Due to practical limitations a simulation cannot run faster than real time, so in order to achieve results quickly, a

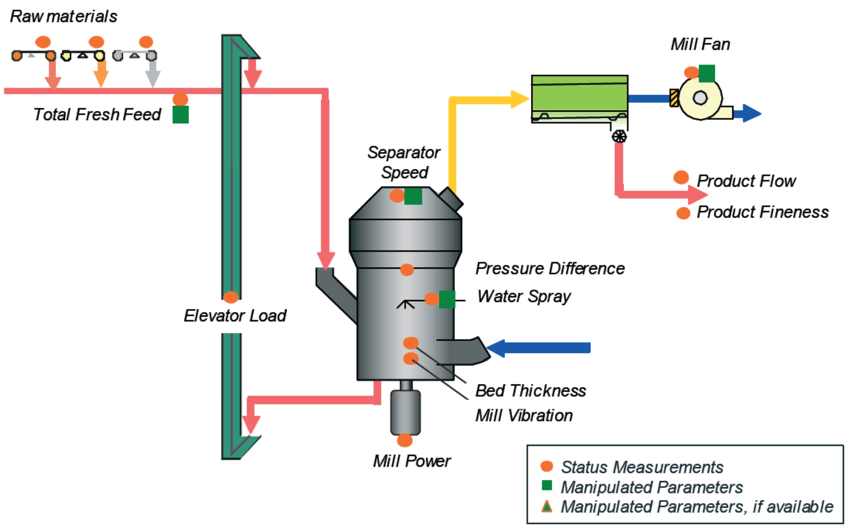


Figure 6.1: Material flow in a vertical raw mill.

vertical raw-mill have been selected, as test subject since this is the system with the fastest time constants available of all processes.

See Figure 6.1. Ground material is air-sweeped from the mill through a separator that returns coarse material back into the mill and allows fine particles to pass. In average, the material is therefore recycled multiple times through the mill before the particle size/mass is low enough to defeat the separator.

Inputs to the system are fresh feed, separator speed and fan power. Observable outputs are differential pressure, power consumption, fineness of end product, magnitude of vibrations, elevator current consumption, a measure of mass flow, and thickness of the layer of material on the grinding table.

The process is controlled by the MPC-algorithm in Figure 3.1 at a sample rate of $0.1Hz$. Implementation details of the algorithm are per definition not relevant for analysis of the execution path. However, we note that the algorithm includes the block “Performance” which is dedicated to generating performance reports, i.e. contains a large amount of source code not related to control or affecting any of the code related to control. Yet, this code is still included in the recorded execution path and may add undesirable noise to the analysis.

6.2 Testing Procedure

The system is started up and allowed to stabilize and left in steady state for an extended period of time. The resulting data record is then considered to represent normal operation. Various disturbances are then applied to the system, forcing it into modes that are deviations from normal operation and cf. the hypothesis in Section 4.2.1, these should somehow be detectable by analyzing records of the execution path.

The disturbances are:

1. Fan speed reduced to 90% for 2min every 4min.
2. Fan speed at 98% constant. This anomaly is considered insignificant and should not differ from normal behavior.
3. Fan speed reduced to 80% for 5min every 10min.
4. Fan speed reduced to 80% for 5min every 10min *and* measurement of separator speed set to zero for 200s every 10min to simulate glitches in signal from sensor.

Each simulation is run overnight or at least several hours.

6.3 Post Processing

The raw data from DynaTrace is stored to disk and accumulates to approximately 0.7Gb for an overnight simulation. Some post processing is applied in order to reduce the amount of data. The algorithm has 79 objects containing source code, and DynaTrace record the execution path by means of the sequence of line numbers executed. Each unique sequence represents one possible state for that particular object. A hash is calculated for each recorded execution path using a standard hashing algorithm and is then considered a unique ID for that execution path, so if e.g. two hashes are identical the corresponding execution paths are also assumed to be identical.

Finally the hash from all objects in a single execution is combined into a hash for the whole worksheet, that is to say a hash of the execution path for the entire algorithm. Thus, at each execution of the algorithm every $10s$, we have a hash for each object, and one hash for the whole worksheet. As an example, there are 6438 samples comprising 270 unique hashes (states) for the worksheet in case of the steady state simulation.

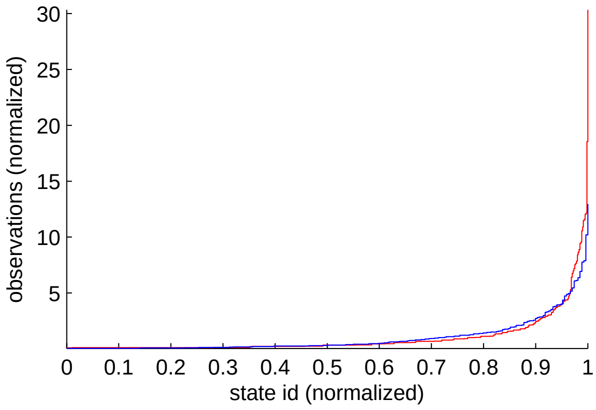


Figure 6.2: Sorted and normalized histograms of observed states. Blue: steady state, red: disturbance no. 1.

6.4 Results

The steady state simulation is compared to disturbance no. 1 in the previous listing and a sorted histogram calculated for the observed states in each simulation:

	Total samples	Common
Steady state	6438	270
Disturbance 1	5374	604

From the observed states listed in the above table it seems like the algorithm stays within a limited set of states when in steady state while more states are covered when disturbance force the system into transient mode.

It turns out that the two simulations have no states in common. This is surprising and should be investigated further.

The sorted histograms from the two simulations are compared in Figure 6.2. The histograms are normalized to denote states in the interval $[0; 1]$ and the area under the curve is 1. The only significant difference in the shape of the histograms is that the algorithm seems to spend more time in a single or few states when disturbed.

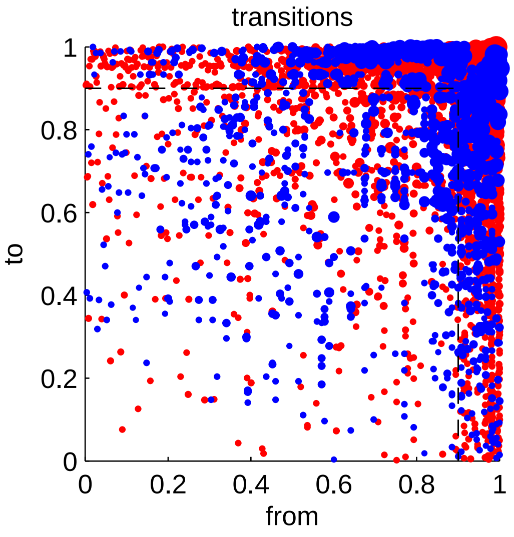


Figure 6.3: All transitions from one state to another. Blue: steady state, red: disturbance no. 1.

6.5 Transitions

As discussed in Section 4.2.1 a possibility is to investigate not only the absolute states but also the transitions between them. The scatter plot of Figure 6.3 shows all observed transitions, where the size of each dot indicate the number of observations. Like in Figure 6.2, the axes show the popularity of the two states involved in the transition, not the popularity of the transition itself. The plot indicates that in both cases, the algorithm tends to go from one popular state to another popular state, while transitions from an unpopular state to another are less frequent. The disturbed system however, seems to cause a slight increase in transitions from popular to unpopular and back.

This tendency is more visible in Figure 6.4, interpolating the number of observations. Note the steep edges of the surface in the disturbed (rightmost) case.

The analysis presented so far has been applied to the other disturbance scenarios with roughly similar results. It seems that the algorithm is indeed behaving differently from steady state in this “state space” when forced into abnormal behavior. However, it is unclear how this should be quantified and measured. We will therefore analyze the execution path behavior on a more detailed level by observing the

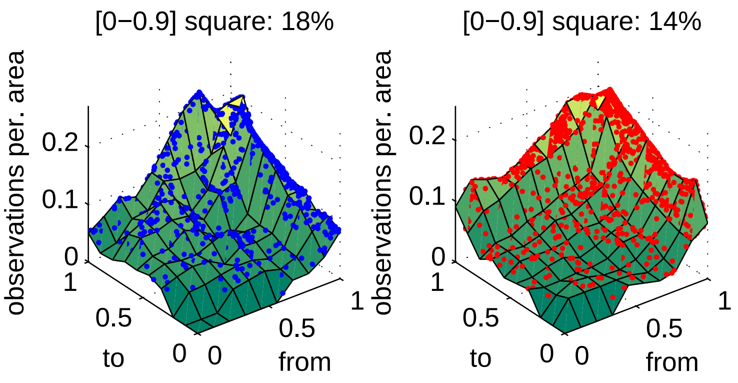


Figure 6.4: Interpolation of observations. The percentage of transitions for which both states are below 0.9 (dashed line in Figure 6.3) is shown. Left: steady state, right: disturbance no. 1.

execution path of each individual object.

Figure 6.5 shows an example for a particular object during the steady state test. It is seen that the object covers 13 different execution paths with different popularity. Here they are sorted and given an integer identifier according to popularity. In this way, the state of the object can be presented by a single integer, the larger the number the more abnormal is the state. By doing so for all objects the state of the whole algorithm can be geometrically presented as a point in the 79 dimensional space $\mathbb{Z}_{79}$.

The immediate question is then how to arrange these dimensions, one for each object, consistently. It must be assumed, that the state of some objects is dependent, e.g. if the same conditional branch exists in multiple objects or if a branch in one object effectively controls the branching of another. Therefore, we chose to perform a singular value decomposition of the steady state data in $\mathbb{Z}_{79}$ to create an eigenspace. We will then map the data from the disturbed simulations into this eigenspace. The hypothesis is then, that the space close to origin can be considered normal operation so that abnormal behavior will be indicated by clusters far away from the origin.

The simulation data is shown in Figure 6.6, projected onto the two most significant

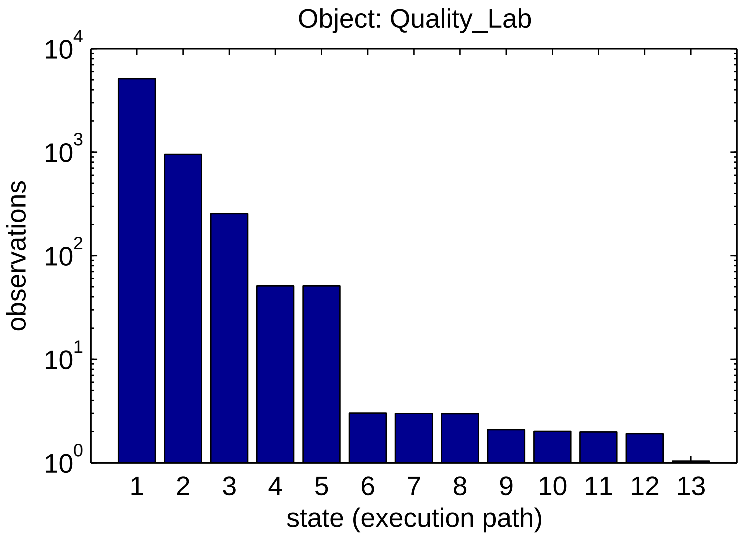


Figure 6.5: Sorted histogram of states for a single object during steady state.

dimensions of eigenspace. It is evident, that the two very abnormal situations caused by 20% disturbance produce states in clusters far from steady state (blue) while the minor, 2% disturbance lies close. It should be noted that Euclidean distance makes no sense in this space. Rather, the abnormality of a state should be measured as the sum of all coordinates - the Manhattan norm.

Figure 6.7 shows this Manhattan norm, or total abnormality for each simulation. Steady state is lowest, as per definition, follwed by 2% constant and 10% altering disturbance as expected. It is not explainable however, why 20% *and* glitches is closer to normal than the 20% disturbance alone.

Finally, the mean contribution to the Manhattan norm over time is depicted in Figure 6.8 for each dimension in eigenspace. Interestingly, it shows that only 15 independent variables are needed to represent the state of the whole algorithm during steady state, including performance reporting. Despite the high number of conditional branches in the algorithm, only 15 of the objects seem to control the execution paths of the others.

Conveniently, the disturbed simulations distinguish themselves from steady state by linearly independent activity of the remaining objects. However, the 2% offset case which is considered very close to normal operation, is surprisingly different from

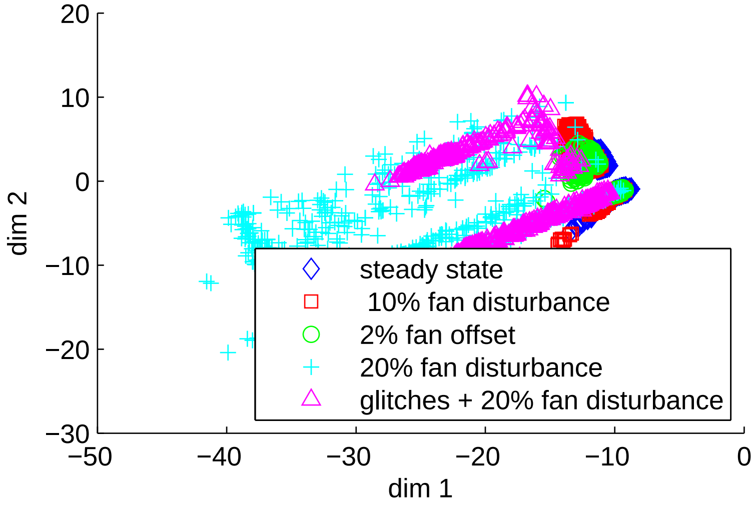


Figure 6.6: Simulation data mapped into the two most significant dimensions of eigenspace.

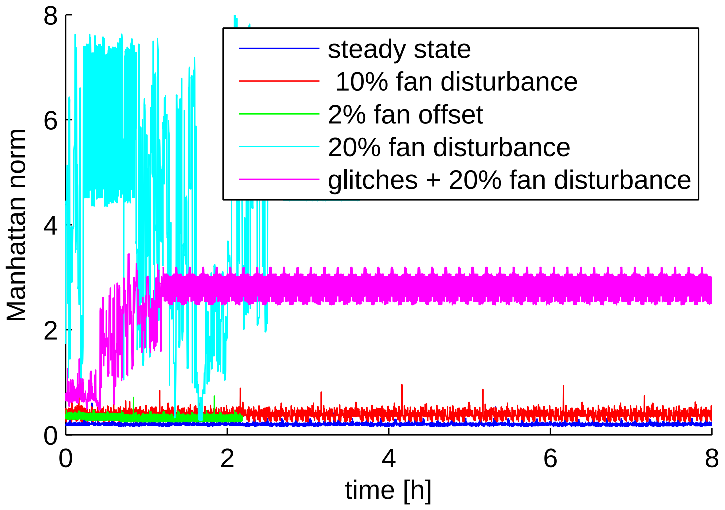


Figure 6.7: Trend of Manhattan norm for each simulation.

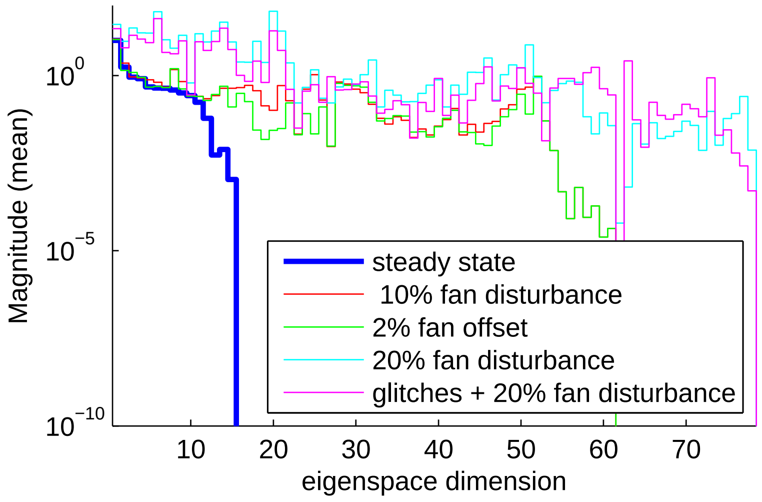


Figure 6.8: Mean value of coordinate. Values below 10^{-10} are effectively zero.

steady state.

6.6 Summary

There are indications that detecting abnormal behavior of the control algorithm by monitoring and analyzing the behavior of its execution path may indeed be possible. The early results presented in this Chapter suggest that statistical difference exist between steady state and situations in which heavy disturbance drives the controlled system and thus the algorithm into abnormal behavior. However, it remains an open question how exactly this can be detected, and how severe the abnormality must be to be detectable in this manner.

Further investigation with data from preferably real world plants is needed to develop an effective detection mechanism.

Chapter 7

Future Work

Dependency graphs extracted from full scale control algorithms can be very large and unsuitable for manual inspection. Nevertheless, they represent information that could be useful for computerized analysis. The initiatives to design a monitoring system for the control algorithm presented in Section 4.2.1 and Chapter 6 utilize only information about the execution path of the algorithm. Proper statistical analysis of not only the execution path but also slicing data, such as the dependency graphs, could possibly improve the ability to raise an early warning in case of abnormal operation. This would be reasonable to investigate further.

The created proof of concept application is matured to be capable of slicing full scale control algorithms running night-over and the next step would be to apply the system on a real world process rather than a simulator in order to produce more realistic data. Such data could be the foundation for further investigation of the mentioned statistical monitoring.

Additionally, the concept of assisted maintenance is to be verified and investigated further with real-world maintenance problems and slicing data.

Continuous logging of execution paths generates potentially large amounts of data, and dynamic program slicing increases the execution time and memory consumption of the control algorithm. Currently no steps have been taken to reduce the amount of logged data.

Finally, the actual benefit by means of reduced man hours used for maintenance is to be confirmed.

Chapter 8

Conclusion

It has been realized that process control algorithms have developed to become complex software systems that need regular maintenance as such. Despite the kind of control paradigm being implemented this software is at some abstraction level defined by source code and this source code is being modified during maintenance.

Analysis of present industrial control algorithms have shown that 90% of the body of these algorithms cannot be classified as classical control theory. While methods exist for reducing or, in theory completely avoiding, the need of maintenance for classical controllers this will only apply to the remaining 10% of the algorithm which can be considered control theory. Furthermore, maintenance scenarios remain for which human interaction is inevitable, such as rebuilding of the plant.

To perform the actual modification of the algorithm is usually simple, whereas understanding the algorithm, realizing the problem, tracking inter-variable dependencies and the flow of signals through the algorithm, and locating the interesting statements which have to be modified are far the most time consuming and demanding tasks.

In this project, attention has been drawn to the issue of maintaining also these 90% of the algorithm, and the kind of maintenance tasks that involve expert maintenance personnel modifying the algorithm. Thus, aim is to reduce the number of man hours used to keep an algorithm running at high performance.

The major contribution of this project is considered to be the introduction of dynamic dependency graphs as a basis for further analysis of control algorithms. A proof of concept has been created to demonstrate the extraction of these data from running, full scale control algorithms by means of program slicing. It has shown that these data can be logged for extended periods of time without consuming extensive amounts of storage, and that it is possible to replay the exact execution of the algorithm including all variable values.

This means that any event of interest in the past, within the logging horizon can be investigated down to the sequence of statements in the source code being executed. At the execution of each line of source code, the value of all variables is available. In contrast, current SCADA systems only offer logging capabilities for selected signals so that important information may be lost. The demonstrated solution on the other hand, guarantees that enough information is stored to perform an exact replay without modifications to the source code. This means that the solution is applicable to existing algorithms and needs no design, configuration, or tuning which is very attractive from an industrial point of view.

Two potential uses of the extracted dependency graphs have been proposed and investigated experimentally. An approach for computer-assisted maintenance by automating the locating of the statements to be modified, the expert is left only to perform the actual modification. Additionally, a concept of monitoring the control algorithm during runtime has been proposed. Aim is to raise an alarm in case of abnormal behavior and to make the system supply an automatic, qualified guess about the location of the responsible part of the algorithm.

Experimental results shows that it is indeed possible to distinguish an (abnormal) heavy disturbance situation from normal operation by analyzing the data extracted from the running algorithm, while the use of dependency graphs in localizing the statements in an algorithm responsible for a fault has been demonstrated on a smaller, artificial example.

Finally, a number of visualizations of dependency graphs have been demonstrated for the purpose of improving navigation and exploration of running algorithms as an additional step to reduce the man hours needed for maintenance.

Bibliography

[Agrawal and Horgan 1990] H. Agrawal and J.R. Horgan. DYNAMIC PROGRAM SLICING. *SIGPLAN Notices*, 25(6):246–256, 1990.

[Åström and Hägglund 1984] Karl Johan Åström and Tore Hägglund. AUTOMATIC TUNING OF SIMPLE REGULATORS WITH SPECIFICATIONS ON PHASE AND AMPLITUDE MARGINS. *Automatica*, 20:645–651, 1984.

[Åström and Wittenmark 1994] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. ISBN: 0201558661.

[Bendtsen et al. 2008] Jan Bendtsen, Klaus Trangbaek, and Jakob Stoustrup. PLUG-AND-PLAY PROCESS CONTROL: IMPROVING CONTROL PERFORMANCE THROUGH SENSOR ADDITION AND PRE-FILTERING. *Proceedings of the 17th World Congress. The International Federation of Automatic Control (IFAC)*, pages 336–341, July 2008.

[Binkley et al. 2006] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, ÁÁkos Kiss, and Bogdan Korel. THEORETICAL FOUNDATIONS OF DYNAMIC PROGRAM SLICING. *Theoretical Computer Science*, 360(1-3):23–41, 2006.

[Bishop 1995] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, November 1995. ISBN: 0198538642.

[Blanke et al. 2003] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki. *Diagnosis and Fault-tolerant Control*. Springer Verlag, 2003. ISBN: 3-540-01056-4.

[Collins et al. 1995] R.E. Collins, R.J. Blue, and M.C. Mound. OPERATING EXPERIENCE AND COST SAVINGS WITH FLSA'S QCX/ONSTREAM ANALYZER INSTALLED AT ST MARYS CEMENT CO., BOWMANVILLE, ONTARIO, CANADA. *1995 IEEE Cement Industry Technical Conference. 37th Conference Record*, pages 103–119, 1995.

[Duda 1977] Walter H. Duda. *Cement-Data-Book*. Macdonald and Evans, 1977. ISBN: 3-7625-0834-8.

[Eccles 2008] L.H. Eccles. THE NEED FOR SMART TRANSDUCERS: AN AEROSPACE TEST AND EVALUATION PERSPECTIVE. *IEEE Instrumentation & Measurement Magazine*, 11(2):23–28, 2008.

[Gamma et al. 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[Hansen et al. 2008] Ole Fink Hansen, Nils Axel Andersen, and Ole Ravn. SUPERVISION AND ASSISTED MAINTENANCE OF CONTROL SYSTEMS. In *The 17. IFAC World Congress*, July 2008.

[Hendricks et al. 2005] Elbert Hendricks, Ole Erik Jannerup, and Paul Haase Sørensen. *Linear Systems Control*. Technical University of Denmark, Ørsted, Automation, DK- Kgs. Lyngby, 3 edition, 2005.
URL: http://server.oersted.dtu.dk/publications/views/publication_details.php?id=2171.

[Holmblad and Østergaard 1995] L.P. Holmblad and J.-J. Østergaard. THE FLS APPLICATION OF FUZZY LOGIC. *Fuzzy Sets and Systems*, 70(2-3):135–146, 1995.

[Keefe and Shenk 2007] B.P. Keefe and R.E. Shenk. THE BEST TOTAL VALUE FOR PYRO TECHNOLOGY. *2007 IEEE Cement Industry Technical Conference Record*, pages 81–89, 2007.

[Kääntee et al. 2004] U. Kääntee, R. Zevenhoven, R. Backman, and M. Hupa. CEMENT MANUFACTURING USING ALTERNATIVE FUELS AND THE ADVANTAGES OF PROCESS MODELLING. *Fuel Processing Technology*, 85(4):293–301, 2004.

[Knuth 1997] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, November 1997. ISBN: 0201896842.
URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&amp;path=ASIN/0201896842.

[Korel and Laski 1988] B. Korel and J. Laski. DYNAMIC PROGRAM SLICING. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[Korel and Rilling 1998] Bogdan Korel and Jurgen Rilling. DYNAMIC PROGRAM SLICING METHODS. *Information and Software Technology*, 40(11-12):647–659, 1998.

[Lau et al. 2008] Nathan Lau, Gyrd Skraaning jr., Greg A. Jamieson, and Catherine M. Burns. ENHANCING OPERATOR TASK PERFORMANCE DURING MONITORING

FOR UNANTICIPATED EVENTS THROUGH ECOLOGICAL INTERFACE DESIGN. *PROCEEDINGS of the HUMAN FACTORS AND ERGONOMICS SOCIETY 52nd ANNUAL MEETING*, pages 448–452, 2008.

[Ljung 1986] Lennart Ljung. *System identification: theory for the user.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN: 0-138-81640-9.

[Maciejowski 2002] J.M. Maciejowski. *Predictive Control with Constraints.* Prentice Hall, England., 2002.

[Mahjoub and Dandachi 2007] A. H. Mahjoub and N. H. Dandachi. POWER SYSTEMS MONITORING & CONTROL CENTERS SHARING SCADA/EMS INFORMATION IN THE AGE OF ENTERPRISE MOBILITY. In *Innovations in Information Technology, 2007. Innovations '07. 4th International Conference on*, pages 312–316, 2007. URL: http://dx.doi.org/10.1109/IIT.2007.4430462.

[Maranzana 2002] M. Maranzana. PLANT AUTOMATION AT SIGNAL MOUNTAIN CEMENT. *IEEE-IAS/PCS 2002 Cement Industry Technical Conference. Conference Record (Cat. No.02CH37282)*, pages 289–306, 2002.

[Martin et al. 2000] G. Martin, T. Lange, and N. Frewin. NEXT GENERATION CONTROLLERS FOR KILN/COOLER AND MILL APPLICATIONS BASED ON MODEL PREDICTIVE CONTROL AND NEURAL NETWORKS. *2000 IEEE-IAS/PCA Cement Industry Technical Conference. Conference Record (Cat. No.00CH37047)*, pages 299–317, 2000.

[Michelsen et al. 2008] Axel G. Michelsen, Roozbeh Izadi-Zamanabadi, and Jakob Stoustrup. TOWARDS AUTOMATIC MODEL BASED CONTROLLER DESIGN FOR RECONFIGURABLE PLANTS. *Proceedings of the 17th World Congress. The International Federation of Automatic Control (IFAC)*, pages 342–346, July 2008.

[Mohapatra et al. 2005] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. COMPUTING DYNAMIC SLICES OF CONCURRENT OBJECT-ORIENTED PROGRAMS. *Information and Software Technology*, 47(12):805–817, 2005.

[Mohapatra et al. 2006] Durga P. Mohapatra, Rajeev Kumar, Rajib Mall, D.S. Kumar, and Mayank Bhasin. DISTRIBUTED DYNAMIC SLICING OF JAVA PROGRAMS. *The Journal of Systems & Software*, 79(12):1661–1678, 2006.

[Mund et al. 2002] G.B. Mund, R. Mall, and S. Sarkar. AN EFFICIENT DYNAMIC PROGRAM SLICING TECHNIQUE. *Information and Software Technology*, 44(2):123–132, 2002.

[Pócza et al. 2005] Krisztián Pócza, Mihály Biczó, and Zoltán Porkoláb. CROSS-LANGUAGE PROGRAM SLICING IN THE .NET FRAMEWORK. *Journal of .NET Technologies*, 3(1-3):141–150, February 2005.

[Pócza et al. 2006] Krisztián Pócza, Mihály Biczó, and Zoltán Porkoláb. TOWARDS EFFECTIVE RUNTIME TRACE GENERATION TECHNIQUES IN THE .NET FRAMEWORK. *Journal of .NET Technologies*, 4(1-3):141–150, 2006.

[Perrier and Kalwa 2005] M. Perrier and J. Kalwa. INTELLIGENT DIAGNOSIS FOR AUTONOMOUS UNDERWATER VEHICLES USING A NEURO-SYMBOLIC SYSTEM IN A DISTRIBUTED ARCHITECTURE. *Europe Oceans 2005*, 1:350–355 Vol. 1, 2005.

[ProcessExpert 2008] FLSmidth A/S Automation. *ECS/ProcessExpert*, 2008. URL: http://attachments.flsmidth.com/FLSA/Brochure/ProcessExpert_BM.pdf.

[Prouty 2006] R. Prouty. THE APPLICATION OF A MODERN CONTROL SYSTEM IN A CEMENT MANUFACTURING FACILITY. *IEEE Cement Industry Technical Conference, 2006. Conference Record.*, page 7 pp., 2006.

[Recke 2006] Bodil Recke. ECONOMICS AND REQUIREMENTS FOR IMPLEMENTING A MPC CONTROLLER IN PRACTICE. *DAU-Bladet*, pages 3–4, August 2006.

[Sheridan and Skjoth 1984] Stephen E. Sheridan and Poul Skjoth. AUTOMATIC KILN CONTROL AT OREGON PORTLAND CEMENT COMPANY'S DURKEE PLANT UTILIZING FUZZY LOGIC. *IEEE Transactions on Industry Applications*, IA-20(3):562–568, 1984.

[Simon et al. 2002] G. Simon, T. Kovacshazy, G. Peceli, T. Szemethy, G. Karsai, and A. Ledeczi. IMPLEMENTATION OF RECONFIGURATION MANAGEMENT IN FAULT-ADAPTIVE CONTROL SYSTEMS. *IMTC/2002. Proceedings of the 19th IEEE Instrumentation and Measurement Technology Conference (IEEE Cat. No.00CH37276)*, 1:123–127 vol.1, 2002.

[Skogestad and Postlethwaite 2005] Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons, 2005. ISBN: 0470011688.

[Sánchez 2003] Enrique Vidal Sánchez. *Robust and Fault Tolerant Control of CD-players*. PhD thesis, Department of Control Engineering, Aalborg University, 2003.

[Song and Lee 2008] E.Y. Song and Kang Lee. UNDERSTANDING IEEE 1451-NETWORKED SMART TRANSDUCER INTERFACE STANDARD - WHAT IS A SMART TRANSDUCER? *IEEE Instrumentation & Measurement Magazine*, 11(2):11–17, 2008.

[Wang and Liao 2004] L. Wang and S. Liao. DISTRIBUTED SUPERVISORY CONTROL AND DATA ACQUISITION (SCADA) SOFTWARE: ISSUES AND STATE OF THE ART. In *IEEE AUTOTESTCON 2004 Proceedings*, pages 332–337, 2004. ISBN: 0780384490. ISSN: 10887725.

[Weiser 1982] Mark Weiser. PROGRAMMERS USE SLICES WHEN DEBUGGING. *Commun. ACM*, 25(7):446–452, 1982.

[Weiser 1984] M. Weiser. PROGRAM SLICING. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[Yuan et al. 2008] Zhugang Yuan, Zhiyuan Liu, and Run Pei. FUZZY CONTROL OF CEMENT RAW MEAL PRODUCTION. *2008 IEEE International Conference on Automation and Logistics*, pages 1619–1624, 2008.

[Zabaniotou and Theofilou 2008] A. Zabaniotou and C. Theofilou. GREEN ENERGY AT CEMENT KILN IN CYPRUS-USE OF SEWAGE SLUDGE AS A CONVENTIONAL FUEL SUBSTITUTE. *Renewable and Sustainable Energy Reviews*, 12(2):531–541, 2008.

www.elektro.dtu.dk

Department of Electrical Engineering

Automation and Control
Technical University of Denmark
Ørsteds Plads
Building 348
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 38 00
Fax: (+45) 45 93 16 34
Email: info@elektro.dtu.dk

ISBN 978-87-92465-09-2