Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

IFAC

Assisted Monitoring and Maintenance of Control Systems

Ole Fink Hansen, Nils Axel Andersen and Ole Ravn

Automation, Department of Electrical Engineering
Technical University of Denmark
(e-mail: ofh@elektro.dtu.dk)

Abstract: Current process control algorithms are complex software systems that need regular maintenance in order to keep a high uptime. Experience shows that the process of understanding an algorithm by tracking signal dependencies throughout the system and recognizing a problem is a major challenge for the maintenance personnel while actually performing the needed modifications to the algorithm is relatively simple. This problem is addressed by a computer science approach to automate the tracking of signals and supply tools for monitoring running control systems, identifying failing parts of the control algorithm and alleviating the task of exploring complex control systems.

Keywords: Monitoring; Signal analysis for FDI; Fault diagnosis; Maintenance; Program slicing.

1. INTRODUCTION

Current process control systems need regular maintenance in order to keep a constant, high performance. Although the control system alone may be time invariant, the controlled process is subject to changes such as wear of machinery, variations in raw materials, modifications to the plant or altered working points due to changed operation mode or market demands.

Thus, a control system designed and tuned for optimal performance at install time will need modifications to cope with the changing tasks.

Performance may be defined using metrics such as up time, process throughput, quality of end product and power consumption but also aspects related to the interaction with the operator such as usability and alarm count.

System identification (Ljung [1986]) ideally fits an accurate model to the process, thus allowing changes in the process to be recognized. Adaptive control (Åström and Wittenmark [1994]) and auto tuning (Åström and Hägglund [1984]) tune the control algorithm to match parametric changes in the process such as drifting gains and time constants.

Fault detection and fault tolerant control (Blanke et al. [2003]) detects structural changes and faulty sensors etc. in the process and continue working possibly with reduced performance.

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 or simply fixing a bug.

Since the time is not ripe for large scale, self maintaining software/algorithms, these modifications must be conducted by expert maintenance personnel. During this process it has turned out that conducting the actual modification is relatively simple while recognizing the malfunction, understanding, and locating the exact part of the algorithm that needs modification is the most time consuming task. Therefore, maintenance personnel would benefit from a system that could somehow:

- Monitor the control system and raise an alarm in case of abnormal behavior.
- Locate the part(s) of the algorithm being responsible for the behavior.
- Track cause-effect dependencies within the algorithm to aid the personnel in investigating only relevant parts of the algorithm.

In this paper we will present a novel concept for such a system and introduce the term *assisted maintenance* in the context of process control to meet the above needs. The concept is based on Dynamic *Program Slicing* (Weiser [1982], Korel and Laski [1988]), a well-established research topic in computer science. Dynamic program slicing allows the backtracking of dependencies among signals in the algorithm and monitoring of how the algorithm is executing.

The use of program slicing means that the algorithm is not required to be designed within a dedicated language nor is it needed to model the plant or the algorithm. This makes the concept attractive for industrial applications since design- and implementation procedures are not affected and the concept can be applied to existing, implemented algorithms without modifications to their source code.

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

978-1-1234-7890-2/08/$20.00 © 2008 IFAC

10.3182/20080706-5-KR-1001.3183

2. BACKGROUND

In this paper we will only consider ordinary sequential algorithms consisting of statements, conditional branches, loops and methods, etc. i.e. implementable in any C-like programming language.

A number of high level control algorithms commissioned for chemical process facilities have been available to study for the caurse of this paper. It has turned out that each consists of an assembly of 1000-10000 source code lines customizing the algorithm for the particular facility.

This assembly may employ sub-algorithms and external utilities such as the solver for a Model Predictive Control problem comprising hundreds of thousands of source code lines. However, these are normally considered black boxes, and all maintenance work is focused on the assembly. Ultimately, a workaround may be included in the assembly to compensate for unintended behavior in such a black box.

One of these algorithms that is considered to be a typical high level controller, has been installed on a large chemical processing facility and some of its characteristics are presented in this section.

Since many chemical engineering processes are nonlinear by nature and most common control techniques only operate on linear approximations of these, switching of control strategy is often time needed when operating in the areas where the nonlinearities are dominant (Berber and Kravaris [1998]).

The above-mentioned algorithm uses not just approx. 30 analog inputs but as many boolean inputs like status flags that must be included in the control decisions. Finally, the industry's demand for high uptime pushes the control algorithms to handle not just normal operation but also interlocks and upset situations such as a failing or saturating actuator etc.

These demands require the control algorithm to incorporate not just traditional control theory, but also a large number of conditional branches and other functionality not directly related to control theory. Consequently, the implementation of control theory will account for only a small amount of the total number of code lines.

A statistical analysis of the assembly of the above-mentioned algorithm (see Table 1) has shown that less than 10% of the lines contains an equation, indicating that only a small fraction of an implementation is related to traditional control theory by means of difference equations, matrix operations etc. The remaining majority of code lines handles initialization, upset recovery, signal validation etc. related to the above mentioned demands.

These parts of the algorithm require maintenance as much as the implementation of actual control theory, and moreover there is a lack of formalism and scientific methods to classify and handle this bulk of the algorithm. Additionally, the number of conditional branches relative to the total lines of source code (Table 1, row 2) indicates a high level of complexity.

2.1 Example

Fig. 1 shows a moving average of the uptime from the mentioned high level control algorithm. When a set of criteria are satisfied to ensure that the operational conditions are within the design limits of the control algorithm, the operators may engage the algorithm to control the process fully autonomously.

Should the process fail to satisfy the criteria once the algorithm is engaged, an alarm is sounded and the algorithm disables itself, leaving the operators to handle situations the algorithm was not designed for. Once the criteria are fulfilled again, the operators may reengage the algorithm manually.

The light shading shows when the criteria are met allowing the algorithm to be enabled, and the dark shading when the algorithm was actually enabled. The black line is the ratio between the two, i.e. the real uptime which will be 100% if the algorithm is running all the time it could run.

The relatively high uptime immediately after installation indicates not only high performance but also the operator's high motivation to enable the algorithm whenever possible.

However, this tendency is decreasing slowly until the algorithm is no longer used, and the process is controlled manually 100% of the time only approx. 200 days after installation. The slow declining of the uptime after 125 days indicates a gradual increase in the number of general problems associated with the use of the algorithm.

Eventually, the problem turned out to be caused by glitches starting to appear in an input signal months after

Table 1. Content of a typical control system assembly. Subsystems handle encapsulated procedures such as a single control loop.

	Count	Example
Lines, total	3487	
Conditional branches	889	**if** a **then**
Equations	300	$a \leftarrow b + c$
Assignments, no equation	985	$a \leftarrow b$
Assignments to 0 or 1	377	$a \leftarrow 0$
Subsystems	82	"Control core pressure"

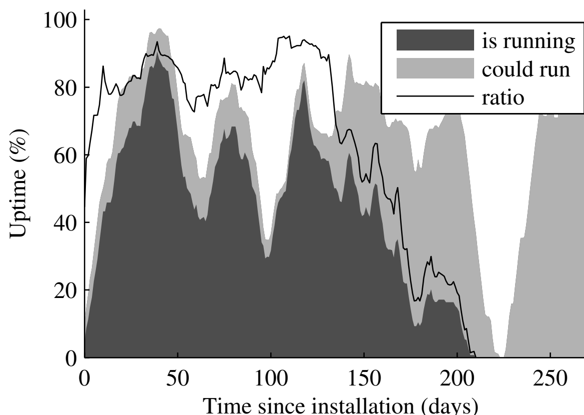

Fig. 1. Average uptime of a high level control system on a process plant since install-time.

installation. A condition the control algorithm was not intended to handle although it is harmless for the overall process control.

Locating the cause of this malfunction was very time consuming for the expert control engineer, while the actual correction needed in the algorithm to make it robust toward such glitches is very simple.

3. INTER-VARIABLE DEPENDENCY TRACKING

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.

The algorithm may be hierarchically structured from well-defined subsystems with explicit inputs and outputs and containing lower level subsystems within higher level systems as illustrated in Fig. 2. 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 being 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 a certain signal e.g., locating the part of the algorithm that modifies a signal, or simply understanding the implementation of an unfamiliar algorithm.

Fig. 3 shows a simplified example. Consider the output identified by x of the entire control system defining the name space **A** in Fig. 2. Backtracking the signal leads to the output y and input f in **B** to input b back in name space **A** as illustrated in Fig. 3. That is to say the process

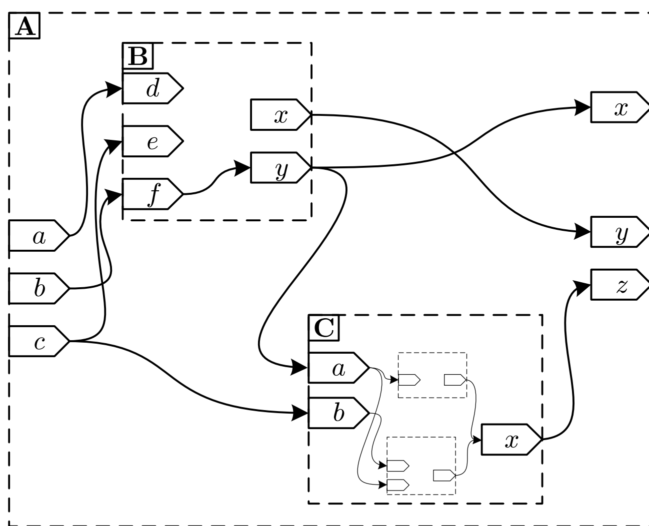

Fig. 2. Example of signal flow through a control algorithm, with inputs a,b,c and outputs x,y,z. It consists of subsystems **B** and **C**. **C** is further comprising two lower level subsystems.

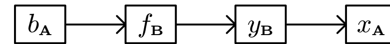

Fig. 3. Backtracking output x in Fig. 2.

of realizing that x is merely an untouched copy of the input b is complex.

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 is particularly complex in the case of control algorithms due to their special structure.

3.1 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 Fig. 4.

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 Fig. 5. The assignment of x in line 2 is control-dependent on the condition in line 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 fixed set of possible dependency graphs limited by the number of branching combinations implicitly affecting y by control-dependency.

3.2 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*.

Algorithm 1

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

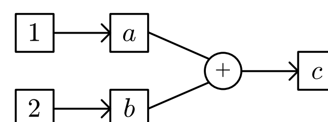

Fig. 4. Dependency graph for variable c after execution of Algorithm 1.

Algorithm 2

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

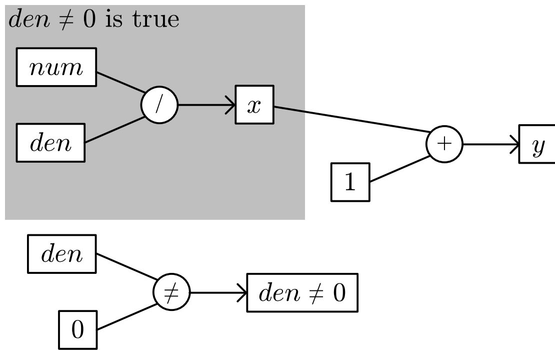

Fig. 5. Dependency graph for variable y in Algorithm 2 for $den \neq 0$. Shaded area denotes control-dependency on the condition $den \neq 0$.

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, dynamic graphs can be used for comparison of executions.

3.3 Program Slicing

A framework has been developed to generate dynamic dependency graphs for variables in existing algorithms based on a computer science technique known as *Program Slicing*. Static program slicing was introduced by Weiser [1982, 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.

In addition to a dependency graph for all variables at every execution, the framework records the *execution path* being the sequence of statements executed. Algorithm 3 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. ASSISTED MAINTENANCE

It is our 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.

4.1 Monitoring and Detection

Any algorithm will have a limited although potentially large set of possible execution paths. Now assume the algorithm to be executed periodically and 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 illustrated in Fig. 6 by 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}.

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 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 by the maintenance crew.

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

4.2 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 Fig. 3, if generated automatically, may be used to highlight the corresponding signal path in Fig. 2 instantly visualizing the signal flow after a particular execution.

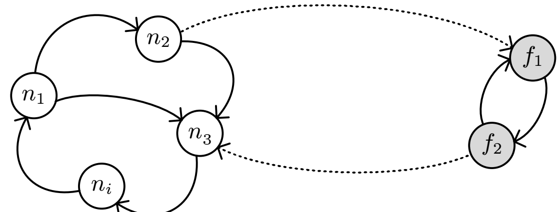

Fig. 6. State machine representation of execution paths for algorithm during normal operation $n_{1...i}$ and abnormal situation f_{1-2}.

The raw dependency graph may be utilized with any level of detail, e.g. the graph of Fig. 5 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. E.g. if output y is predominantly affected by subsystems classified as "upset recovery" and rarely by "normal operation" subsystems alone, 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.

4.3 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 the last two weeks by a running algorithm, see Fig. 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 Fig. 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 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 Fig. 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.

The sequence of tasks from recognizing an interesting feature in the history of a signal in Fig. 7, generating the dependency graph comparison in Fig. 8 and 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 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.

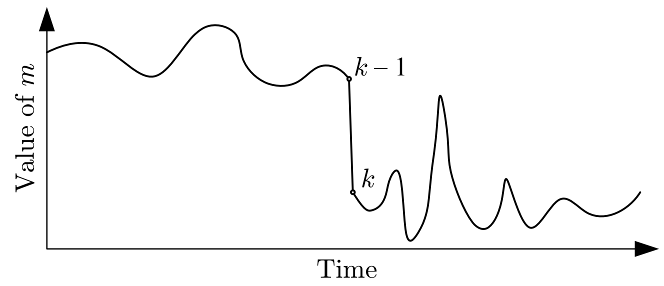

Fig. 7. Discontinuity in the output m between two executions of the algorithm at sample $k-1$ and k.

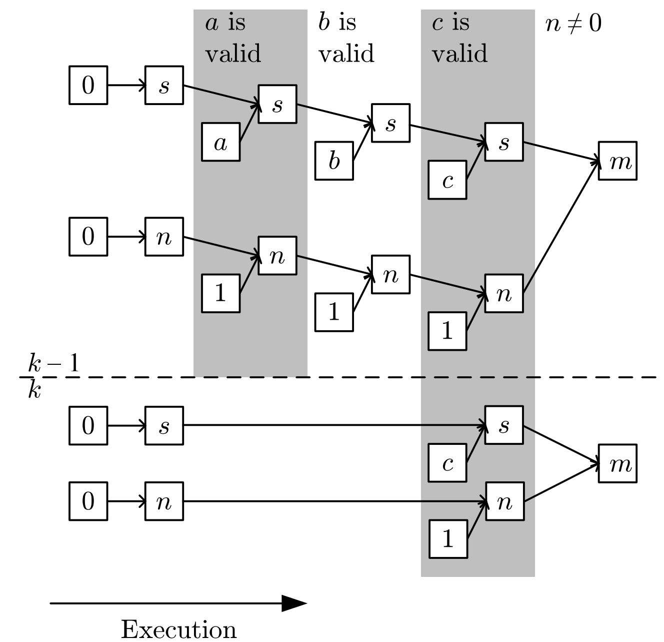

Fig. 8. Comparison of dependency graphs of m for executions $k-1$ and k. Operators are omitted for the sake of readability.

Algorithm 4 Algorithm-snippet associated with the dependency graphs in Fig. 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**

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

5. CONTROL ALGORITHMS AS SOFTWARE SYSTEMS

The concept proposed in this paper is based on the approach that control algorithms are recognized as software systems. Consequently, it must be assumed that general tools known from common software development such as certain design guide lines and validation procedures have a positive influence on control system maintainability. However, the special properties of control algorithms as presented in Sections 2-4 represent unique challenges that are to be targeted by tools specialized for maintenance of control algorithms.

Many control system related issues may be addressed by dedicated solutions. E.g. consider an algorithm containing a P-controller:

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 in this paper 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 to be applied orthogonally to the algorithm, disregarding the kind of control theory implemented.

6. FUTURE WORK

A dynamic program slicing system has been developed and can generate dynamic dependency graphs and execution paths from running algorithms. In order to develop and verify the statistical analysis of the state machine to detect abnormal behavior, data by means of execution paths must be collected from a realistic, running control algorithm, during both normal and abnormal situations. Furthermore, realistic data is needed in order to evaluate methods for comparison of dependency graphs for the location of statements of interest. Therefore, the developed system must be expanded to handle entire, realistic control algorithms.

Continuous logging of execution paths generate potentially large amounts of data, and dynamic program slicing increases the execution time and memory consumption of the control algorithm. The overall performance cost and/or methods for reducing the amount of data are to be investigated.

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

7. CONCLUSION

It has been realized that process control algorithms need regular maintenance by means of modifications in the source code, and that more than 90% of the body of these algorithms cannot be classified as classical control theory. Since the time is not ripe for fully automating such operations, the maintenance task is to be conducted by human experts.

While performing the actual modification of the algorithm is usually simple, understanding the algorithm, realizing the problem, tracking inter-variable dependencies and the flow of signals through the algorithm and locating the interesting statements that are to be modified is far the most time consuming and demanding task.

A novel approach for computer-assisted maintenance based on program slicing has been presented to alleviate this task. By automating the locating of the statements to be modified, the expert is left only to perform the actual modification. Additionally, a method for monitoring the control algorithm during runtime has been proposed. In case of abnormal behavior an alarm is raised, and the system makes a qualified guess on the responsible part of the algorithm.

Finally, a number of proposals involving the use of dependency graphs to enhance navigation and exploration of running algorithms have been presented.

REFERENCES

Karl Johan Åström and Tore Hägglund. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20:645–651, 1984.

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

Ridvan Berber and Costas Kravaris. *Nonlinear Model Based Process Control*. Kluwer Academic Publishers, August 1998. ISBN 0792352203.

M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki. *Diagnosis and Fault-tolerant Control*. Springer Verlag, 2003.

B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988. ISSN 0020-0190.

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

Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982. ISSN 0001-0782.