IADIS International Journal on Computer Science and Information Systems
Vol. 3, No. 1, pp. 44-58
ISSN: 1646-3692

DEPENDABILITY OF THE SOFTWARE IMPLEMENTATION OF THE EXPLICIT DMC ALGORITHM

Piotr Gawkowski[1], Maciej Ławryńczuk[2], Piotr Marusak[2], Janusz Sosnowski[1] and Piotr Tatjewski[2]

[1] *Institute of Computer Science*
{P.Gawkowski, J. Sosnowski}@ii.pw.edu.pl

[2] *Institute of Control and Computation Engineering*
Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
{M.Lawrynczuk, P.Marusak, P.Tatjewski}@ia.pw.edu.pl

ABSTRACT

The paper studies dependability of software implementation of the explicit DMC (Dynamic Matrix Control) Model Predictive Control (MPC) algorithm applied for a rectification column. The process with two inputs and two outputs with strong cross-couplings and significant time delays is studied. The algorithm's control law is calculated off-line. Dependability is evaluated experimentally using software implemented fault injection approach. The injected faults influence the quality of rectification process.

KEYWORDS

dependability evaluation, fault injection, DMC algorithm, rectification process, process control

1. INTRODUCTION

Faults appearing during system operation may be critical for implemented applications. They can result in logical errors and application failure (Benso and Prinetto, 2003; Gawkowski and Sosnowski, 2003). It is particularly critical in many reactive systems (e.g. nuclear plants, satellites, aircrafts, chemical industry, medicine). Hence, an important practical issue is to evaluate dependability of software applications in the presence of faults. This paper studies the dependability of software implementation of the explicit version of Dynamic Matrix Control

(DMC) Model Predictive Control (MPC) algorithm applied to a rectification column (Wood and Berry, 1973). In the research the software implemented fault injector adapted to reactive applications is used (Gawkowski and Sosnowski, 2007).

Model Predictive Control is practically the only advanced control technique which have found acceptance in the process industry and is successfully applied in practice (Ławryńczuk et al, 2007; Maciejowski, 2002; Morari and Lee, 1999; Rossiter, 2003; Tatjewski, 2007; Tatjewski et al, 2006). In the MPC algorithms the control action is calculated using a model of the process. If the model is accurate enough, performance offered by MPC algorithms can be better than the one offered by classical control algorithms, especially for processes with difficult dynamics, e.g. with significant time delay. Moreover, thanks to using the process model, the MPC algorithms contain the decoupling mechanism inside. It is very important for Multi-Input Multi-Output (MIMO) processes with strong cross-couplings. Among different MPC techniques, DMC is often applied in practice because it uses a step-response model of the process which is very easy to obtain (Cutler and Ramaker, 1979; Tatjewski, 2007).

The paper is structured as follows. In the next Section the explicit DMC control algorithm is described in a version for MIMO control processes. In Section 3 fault injection test-bed, experiment set-up and some new aspects of experiment conduction according to control algorithms specificity are presented. In Section 4 results of conducted experiments are discussed. Some software-based dependability improvements are also presented. The last section summarises the paper.

2. DMC ALGORITHM FOR MIMO PROCESSES

The distinctive feature of the MPC algorithms is that the on-line calculation of control policy considers the future behaviour of the control process many time-steps ahead. The prediction is made using a dynamic model of the control process. Typically, the future control values are chosen in such a way that the predicted behaviour of the control algorithm minimises a performance function formulated usually as follows:

$$J = \sum_{j=1}^{n_y}\sum_{i=1}^{N}\psi^j \cdot \left(\bar{y}_k^j - y_{k+i|k}^j\right)^2 + \sum_{j=1}^{n_i}\sum_{i=0}^{N_u-1}\lambda^j \cdot \left(\Delta u_{k+i|k}^j\right)^2, \tag{1}$$

where $\bar{y}_k^j$ is a set-point value for the jth output, $y_{k+i|k}^j$ is an output value for the $(k+i)$th sampling instant predicted at kth sampling instant (the way of its calculation depends on a model used for prediction), $\Delta u_{k+i|k}^j$ are future changes in the manipulated variables, $\psi^j \geq 0$ and $\lambda^j \geq 0$ are weighting coefficients for the predicted control errors of the jth output and for the changes of the jth manipulated variable, respectively, N and N_u denote prediction and control horizons, n_y, n_i denote the number of outputs and inputs, respectively.

The performance function (1) can be rewritten in the following way:

$$J = \left(\bar{\boldsymbol{y}} - \boldsymbol{y}\right)^T \cdot \boldsymbol{\Psi} \cdot \left(\bar{\boldsymbol{y}} - \boldsymbol{y}\right) + \Delta\boldsymbol{u}^T \cdot \boldsymbol{\Lambda} \cdot \Delta\boldsymbol{u}, \tag{2}$$

where $\boldsymbol{\Psi}$ is a weighting matrix of dimensionality $n_yN \times n_yN$, $\boldsymbol{\Lambda}$ is a weighting matrix of dimensionality $n_iN_u \times n_iN_u$, $\Delta\boldsymbol{u}$ is a vector of dimensionality n_iN_u composed of the future increments of control values, $\bar{\boldsymbol{y}}$ is a vector of dimensionality n_yN composed of set–point values $\bar{y}_k^j$, $\boldsymbol{y}$ is a vector of dimensionality n_yN composed of output values $y^j_{k+i|k}$ predicted using a control process model. If the prediction is performed using a linear process model then the superposition principle can be used and the vector $\boldsymbol{y}$ can be decomposed as:

$$\boldsymbol{y} = \boldsymbol{y}^0 + \boldsymbol{G} \cdot \Delta\boldsymbol{u}, \tag{3}$$

where $\boldsymbol{G}$ is a matrix of dimensionality $n_yN \times n_iN_u$ called a dynamic matrix composed of the elements of the process step response, $\boldsymbol{y}^0$ is a vector of dimensionality n_yN called a free response because it contains elements equal to the values of the process outputs in the future in the situation if manipulated signals are frozen at the kth sampling instant (Maciejowski, 2002; Rossiter, 2003; Tatjewski, 2007).

If the performance function (2) is minimised without constraints, the optimal unique solution is:

$$\Delta\boldsymbol{u} = \boldsymbol{K} \cdot \left(\bar{\boldsymbol{y}} - \boldsymbol{y}^0\right), \tag{4}$$

where

$$\boldsymbol{K} = \left(\boldsymbol{G}^T \cdot \boldsymbol{\Psi} \cdot \boldsymbol{G} + \boldsymbol{\Lambda}\right)^{-1} \boldsymbol{G}^T \cdot \boldsymbol{\Psi}. \tag{5}$$

Only the elements $\Delta u^j_{k|k}$ of the vector $\Delta\boldsymbol{u}$ are applied to the process and then the procedure is repeated in the next sampling instant. The step-response model of the controlled process used by the DMC algorithm is following:

$$y^j_k = \sum_{m=1}^{n_i} \sum_{i=1}^{D-1} s^{j,m}_i \cdot \Delta u^m_{k-i} + s^{j,m}_D \cdot u^m_{k-D}, \tag{6}$$

where Δu^m_k is a change in the mth manipulated variable at the kth sampling instant, $s^{j,m}_i$ ($i = 1,\dots, D$) are step response coefficients of the controlled process describing influence of the mth input on the jth output, D is equal to the number of time instants after which the coefficients of the step responses can be assumed as settled, u^m_{k-D} is a value of the mth manipulated variable at the $(k–D)$th sampling instant.

The DMC control law (4) can be formulated:

$$\begin{bmatrix} \Delta u^1_{k|k} \\ \Delta u^2_{k|k} \\ \vdots \\ \Delta u^{n_i}_{k|k} \end{bmatrix} = \boldsymbol{K}^e \begin{bmatrix} \bar{y}^1_k - y^1_k \\ \bar{y}^2_k - y^2_k \\ \vdots \\ \bar{y}^{n_y}_k - y^{n_y}_k \end{bmatrix} - \sum_{j=1}^{D-1} \boldsymbol{K}^{\boldsymbol{u}}_j \begin{bmatrix} \Delta u^1_{k-j} \\ \Delta u^2_{k-j} \\ \vdots \\ \Delta u^{n_i}_{k-j} \end{bmatrix}, \tag{7}$$

where $\boldsymbol{K}^e$ is a matrix of dimensionality $n_i \times n_y$ and $\boldsymbol{K}^{\boldsymbol{u}}_j$, $j=1,\dots,D–1$ are matrices of dimensionality $n_i \times n_i$ composed of controller coefficients. Because these matrices and the resulting control law are calculated off-line, the discussed MPC algorithm is named the explicit one. The detailed description of the explicit DMC algorithm derivation can be found in (Pułaczewski, 1998; Tatjewski, 2007).

3. EXPERIMENT SET-UP

The concept of the SoftWare Implemented Fault Injector (SWIFI) is based on the software emulation of a fault during the run-time of the application under test. In this research FITS fault injector is used (Gawkowski and Sosnowski, 2006; Sosnowski et al, 2003a). It uses standard Win32 *Debugging API* to control the execution of the software application under tests. It captures all events (e.g. exceptions generated by the tested program), can set single step execution, trap on specific conditions breaking tested program execution and then perform required actions in the context of that program (e.g. read and change program's memory content, get and change states of registers). All of that allow building complex disturbance scenarios, which can emulate fault appearance of a given type (e.g. execution of some extra instructions in the tested program or modifications of the memory/CPU registers states at specified conditions). It is worth to note that the application source code does not require any instrumentation. However, some simple instrumentation can help to obtain more detailed knowledge on the fault effects and future improvement of the application. It will be discussed in section 3.2.

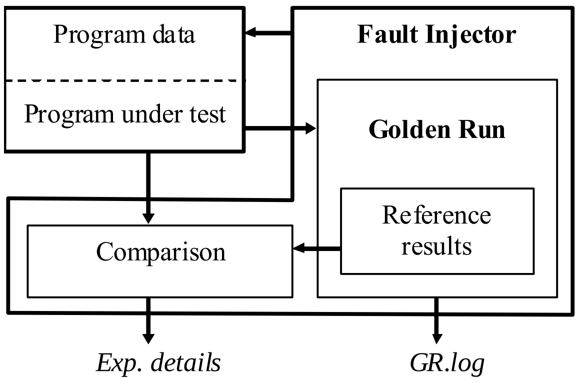

Figure 1. Fault injection scheme

The general idea of fault injection experiments is shown in Fig 1. For the analysed application (program and input data) we generate so called *golden run* which provides us with reference results and the application profile (GR.log). During this step the preliminary statistical analysis is made in parallel to better characterize the tested application and prepare proper fault disturbance profile for the experiments. This is a time consuming task performed at the machine instruction level (in CPU's single-step mode). We distinguish two views on the statistics: the static one describes static code analysis of the application while the dynamic one relates to the executed instruction stream. The analysis includes the following statistics:

- mnemonics (number and percentage of different mnemonics and their operand variants),
- binary code analysis: number of zero and one bits, size of the code,
- opcode length and type analysis,

- CPU resources usage profile: number of reads, writes, state changes of the given CPU register, minimal and maximal values, bit position usage and the activity of the register (Gawkowski et al, 2005).

Such analysis is helpful in further experiment profiling (helps to point out which resources are used and which are not), interpretation of obtained experimental results (e.g. correlation with resource activity), normalization of experimental results (Gawkowski et al, 2005).

After the Golden Run, in the series of subsequent executions called *tests*, FITS injects faults into the running application and compares its behaviour with the reference one. As the result we get general statistics of test results and some more detailed information (exp. details). The whole fault injection process is controlled by the operator by means of a graphical interface but it is fully automated so no human interaction is needed during experiment run.

In the following the process controlled by the analysed control algorithm (DMC) is described, instrumentation (facilitating further analysis) of the tested application is introduced, then the fault insertion policy, and finally, applied result qualification is given.

3.1 MIMO process description

The process is a rectification column with two manipulated (inputs) and two controlled (outputs) variables shown in Fig. 2. It is described by the continuous-time transfer function model (Pułaczewski, 1998; Wood and Berry, 1973) (time constants in minutes):

$$\begin{bmatrix} Y^1(s) \\ Y^2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{12{,}8}{16{,}7s+1} & \dfrac{-18{,}9e^{-4s}}{21{,}0s+1} \\ \dfrac{6{,}6e^{-8s}}{10{,}9s+1} & \dfrac{-19{,}4e^{-4s}}{14{,}4s+1} \end{bmatrix} \cdot \begin{bmatrix} U^1(s) \\ U^2(s) \end{bmatrix} + \begin{bmatrix} \dfrac{3{,}8e^{-4s}}{14{,}9s+1} \\ \dfrac{4{,}9}{13{,}2s+1} \end{bmatrix} \cdot U^3(s), \qquad (8)$$

where the controlled variables are: y^1 – methanol concentration in the distillate (the top product), y^2 – methanol concentration in the effluent (the bottom product), the manipulated variables are: u^1 – flow rate of the reflux, u^2 – flow rate of the steam into a boiler, u^3 is feed flow rate (a disturbance). All process variables are scaled.

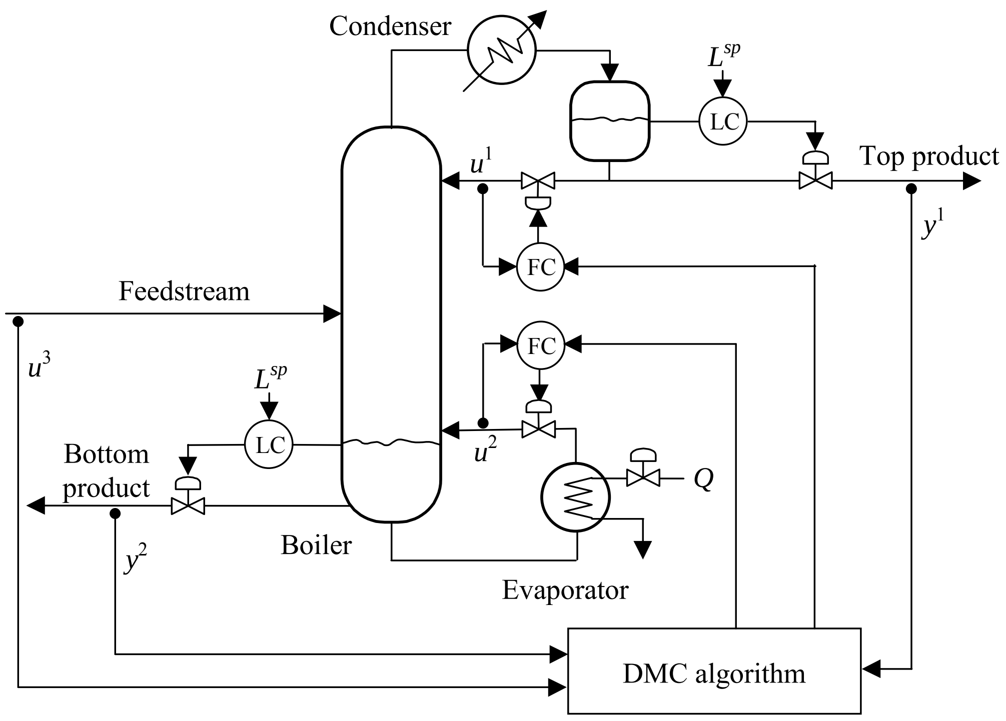

Figure 2. Rectification column control system structure

For the considered rectification process the explicit DMC algorithm is designed: the sampling period T_p=1 min is assumed, the dynamics horizon is equal to the prediction horizon D=N=100, the control horizon N_u=50, the values of weighting coefficients are: $\psi^1=\psi^2=1$, $\lambda^1=\lambda^2=10$. The simulation horizon is 300 discrete time-steps. Structure of the control system with the explicit DMC algorithm is shown in Fig. 3.

3.2 Code instrumentation

FITS disturbs directly the tested application only within so-called testing areas (Gawkowski and Sosnowski, 2007). Testing areas limit the scope of disturbances only to the selected parts of the application. Here, the application also contains the mathematical model of the controlled process for simulation. The parts of the tested application disturbed during the experiments (dashed box) as well as process models (not disturbed) are marked in Fig. 3.

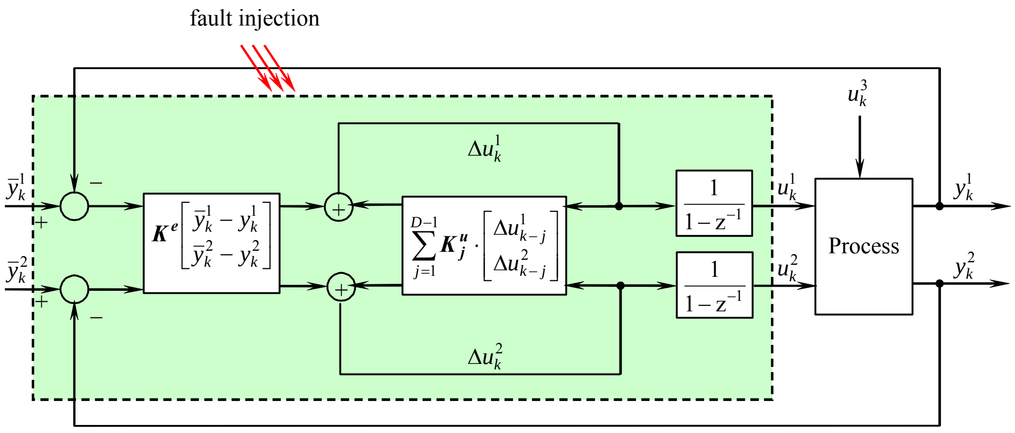
fault injection
$\bar{y}_k^1$
$\bar{y}_k^2$
$K^e \begin{bmatrix} \bar{y}_k^1 - y_k^1 \\ \bar{y}_k^2 - y_k^2 \end{bmatrix}$
$\sum_{j=1}^{D-1} K_j^u \cdot \begin{bmatrix} \Delta u_{k-j}^1 \\ \Delta u_{k-j}^2 \end{bmatrix}$
Δu_k^1
Δu_k^2
$\frac{1}{1-z^{-1}}$
u_k^1
u_k^2
u_k^3
Process
y_k^1
y_k^2

Figure 3. Structure of the control system with the explicit DMC controller

The controlled process is not fault-affected during the experiments as shown. The tested application is also instrumented in order to calculate some measures (e.g. related to internal variables values, output signal deviations, process outputs), save them into the output file (separate for each test) and send to the fault injector using user-defined messages (Gawkowski and Sosnowski, 2007). Those messages are collected by FITS and reported within the experiment summary. It is helpful in further analysis of test cases (e.g. by finding assigned test cases and correlating them with the saved test outputs). Here, the distribution of the rectification column product deviation over the tests is build (Section 4). The output files (unique for each test – filenames are assigned by FITS) consist the set of control variables' values as well as the column's products values over the simulated time period.

3.3 Fault injection policy

FITS simulates faults by disturbing the running application. The following fault types are possible: bit inversion, bit setting, bit resetting, bridging (logical AND or OR of coupled bits). It is also possible to select pseudorandom generation of fault types (single, multiple). Duration of the fault is specified in number of instructions for which the fault must be active starting from the triggering moment. This mechanism gives the possibility of setting transient and permanent faults. The assured fault specifications make it possible to model physical faults in various functional blocks of the system, in particular, processor sequencer, processor ALUs, FPU, general purpose and control registers, bus control unit, RAM memory. FITS provides high flexibility in specifying the moment of fault injection - fault triggering point. The set of faults to be inserted can be specified explicitly or generated in a pseudorandom way.

In this study the single bit-flip faults within CPU and FPU registers, application's data and machine instruction code are considered. Faults are injected pseudorandomly in time (program execution) and space (bit position within disturbed resource, distribution over application's memory). Such a fault model well mimics Single Event Upset (SEU) effects (Gawkowski et al, 2005; Gawkowski and Sosnowski, 2003; Sosnowski et al, 2005).

3.4 Qualification of experimental results

The correctness qualification of results produced by the application under test is more complicated in case of applications related to control systems (Sosnowski et al, 2005) than in case of simple calculation-oriented ones. Control algorithms require complex analysis of the controlled process behaviour. The standard factor SSE (Sum of Squared Error) is used as a measure of result (y^1, y^2) correctness. The reference SSE value (obtained during referential execution – non-faulty) is 2.53 (due to delayed response and feed stream disturbances). The whole experiment is conducted by FITS automatically. At the end of the experiment synthetic (aggregated) results for each fault location are given. In general, 4 classes of test results are distinguished:

- C: correct behaviour (SSE<5),
- INC: incorrect (unacceptable) behaviour (SSE≥5),
- S: test terminated by the system due to un-handled exception,
- T: timed-out test.

Analysis of fault effects requires detailed information upon the faults injected and the application behaviour. FITS provides details about every test (simulated fault injection). Hence, manual replay of the whole test execution can be done. Moreover, all the events and user messages occurring during the test are recorded. The tested application is instrumented to save its outputs (here simulation results, i.e. a set of control signals in subsequent sampling instants) into separate files for each test (file names are managed by FITS). This gives a possibility for post-experiment analysis of fault effects in the correlation with the injected fault and observed behaviour for each test.

4. SIMULATION RESULTS

Three versions of software implementations are considered. All of them are written in C language and compiled using Microsoft Visual C++ 2005. The first, the rudimentary version, is based on direct implementation of DMC algorithm. Two remaining implementations contain some hardening techniques.

4.1 Rudimentary implementation

The first examined implementation of the control algorithm takes 124 machine instructions (405 bytes of the static code). The algorithm needs execution of 1020000 instructions for the whole simulation horizon (300 discrete sampling instants). As the static and dynamic profile is different, the distribution of mnemonics in the static code as well as in the executed stream (dynamic) is presented in Fig. 4.

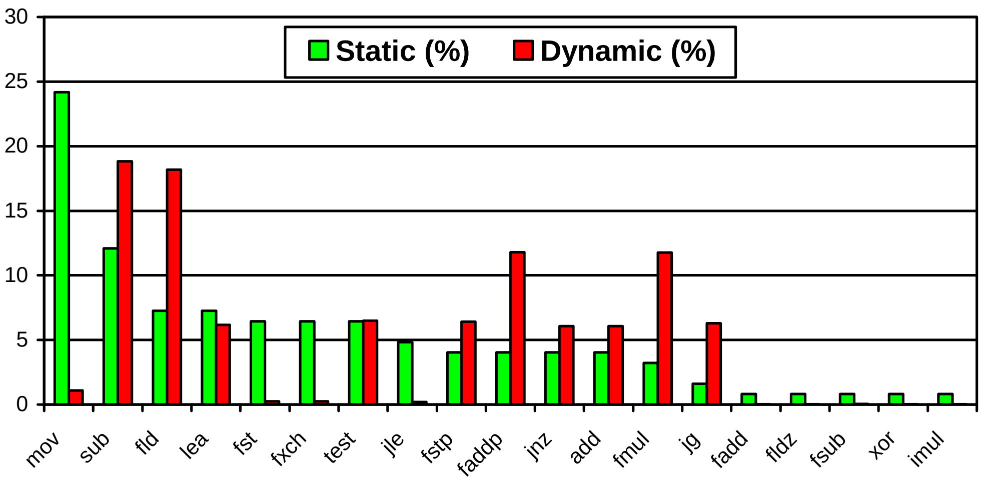

Figure 4. Distribution of DMC mnemonics in static and dynamic profile

It is worth noting that floating point instructions constitute 38% of the code in the static code while dynamically they are executed in 54,8% of time. Moreover, instructions organising computational loops in the DMC implementation (*sub*, *test*, *add*, *jnz*, *jg*) take another 38%. Hence, the application is strongly computational with high degree of FPU utilisation. Nevertheless, the instruction set used is rather limited. On the other hand, the activity ratio for CPU resources is high (98, 94, 80, 98, 97, 81 % for EAX, EBX, ECX, EDX, ESI and EDI, respectively) (Sosnowski et al, 2005).

Faults in the CPU and FPU registers, data area of the application, executed instruction stream, and static code image are considered. For each fault location approximately 1000 disturbed executions are investigated (single fault injected in each application execution). The summary of results (according to categories described in Section 3) is presented in Fig. 5. As the algorithm uses many parameters (400) it is very robust to fault located in the data area – there are only few data memory locations critical for the algorithm – most of the data correspond to the algorithm's parameters. The high degree of FPU robustness could be astonishing. Past experience shows that the FPU is rarely used hard (Gawkowski and Sosnowski, 2003) (e.g. only few FPU stack locations used simultaneously). This results in overall low fault sensitivity of the FPU. Nevertheless, there are some very sensitive locations within the FPU (e.g. control registers).

The most fault sensitive resource of the DMC controller is its code. Fortunately, there are software techniques (e.g. exception handling, duplication of critical data and code) that can be applied at the source code level to provide fault robustness (Gawkowski and Sosnowski, 2005a; 2005b; Gawkowski et al, 2005; Gawkowski and Sosnowski, 2002). They have already proved their advantages; nevertheless, their effectiveness in the considered algorithm is investigated in Section 4.2. One can expect, that in the DMC case some simplifications in the fault hardening implementations can be applied without noticeable aggravation of dependability and performance (i.e. rare errors on the controlled process inputs are not critical for its behaviour).

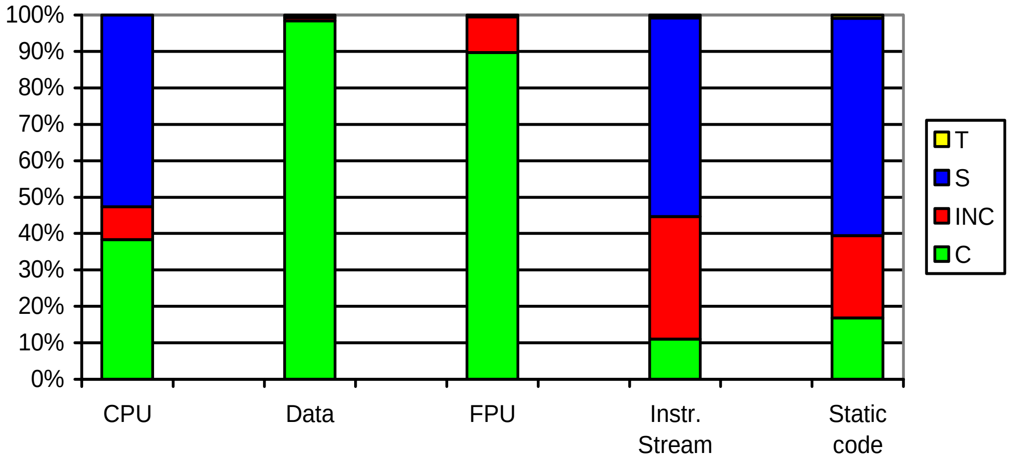

Figure 5. Experimental results of rudimentary DMC implementation

Analysing fault susceptibility it is worth correlating the observed effects (simulated process behaviour) with the injected fault details in accordance to the source and machine code of the disturbed DMC. Fig. 6 and 7 present plots of the application outputs (y^1, y^2 – left plots and u^1, u^2 – right plots) over the DMC iteration number in case of sample fault disturbed executions. For reference the undisturbed simulation results are shown, i.e. the golden run (blue lines). The simulation scenario is as follows (blue lines). At the beginning, the process is driven to a given set-point. Then, at sampling instant 30 the change in the feed stream flow rate (u^3) is introduced (from 0 to 0.1). Another change in u^3 is made at the instant 140 (from 0.1 to –0.05).

In the case considered in top plots in Fig 6, the fault is injected at the sampling instant 28. It results in the change of the *faddp* instruction into the *fmulp* (operands remained unchanged). The instruction disturbed is used to calculate the *du1* variable of the DMC algorithm (corresponding to the Δu_k^1 in Fig. 3). As a result the source code statement `du1+=r1[i]*vektup[i]` is changed to `du1*=r1[i]*vektup[i]`. The result of this disturbance varies on the control signal and process states. For instance, the considered fault injected at the 28th sampling instant results in SSE=3.39 (Fig. 6 top), at the 101st sampling instant in SSE=4.25 (Fig. 6 middle), at the 224th – SSE=2.53 (Fig. 6 bottom) (the same as the reference SSE value). Hence, it disturbs the top product composition.

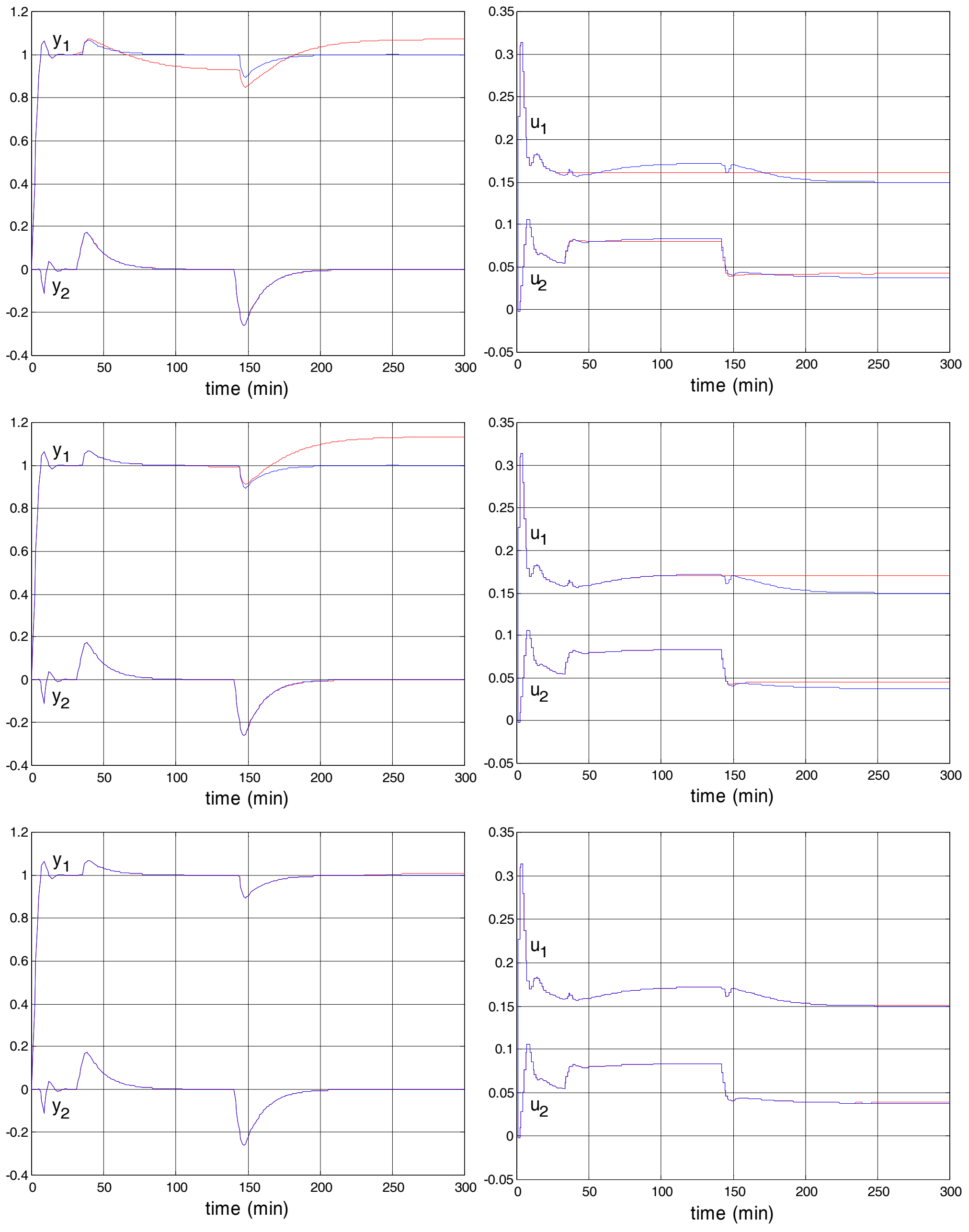

Figure 6. Single bit inversion within the *faddp* instruction disturbs the top product composition. Faults injected at sampling instants: *top panels:* 28th, SSE=3.39, *middle panels:* 101st, SSE=4.25, *bottom panels:* 224th SSE=2.53; golden run responses in blue, fault injected responses in red

More critical situation is illustrated in Fig. 7. Single bit inversion (at 61st sampling instant) within the same instruction as described above destabilises the process. In this case the instruction mnemonic remain unchanged while its first operand changed from *st(2)* to *st(0)*. Observed SSE is 4.40.

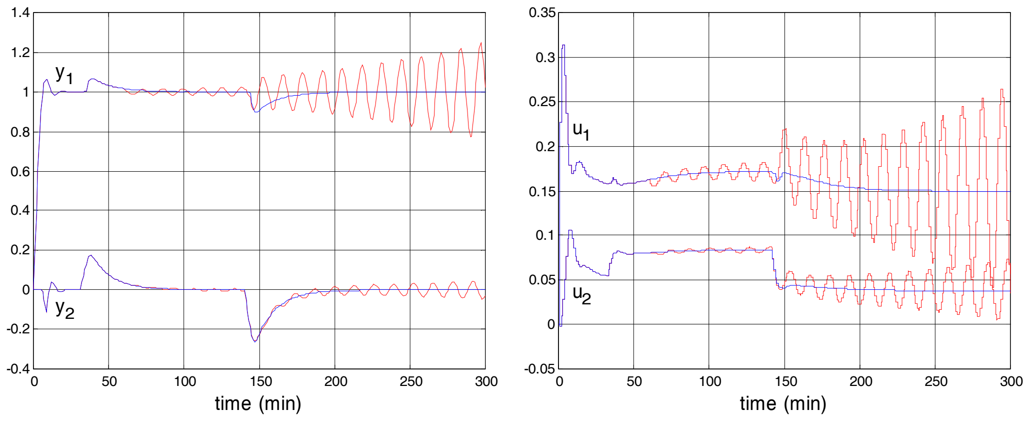

Figure 7. Single bit inversion within the *faddp* instruction destabilises the process. Fault injected at the 44th sampling instant, SSE=4.40; golden run responses in blue, fault injected responses in red

It is worth noting that overwhelming majority of incorrect behaviour relates to very high SSE values (higher than 1000). It means that the values of control errors cumulate. At the other hand it gives the possibility to easily detect such big deviations using additional diagnostic subroutines (e.g. assertions on DMC variables).

4.2 Dependability improved implementations

Two software-hardened versions are analyzed. Different parameters for exception handling of the used compiler are also examined. In rudimentary implementation in C language (not fault hardened) three parts can be distinguished (referenced in further pseudocode listings – the whole sequence of these parts is referenced as DMC):

- **part_A**: calculation of increments of manipulated control variables values (according to the equation 4, e.g. control errors calculation) – the major part,
- **part_B**: saving the calculated values of increments into the historical vector of increments (needed for further processing),
- **part_C**: calculation of manipulated control variables values for the next iteration (integration of previous values with the increments calculated in current iteration).

The idea of the first hardened implementation is to triplicate the main (i.e. DMC algorithm calculation) part and to vote over calculated propositions. It is worth to note that no exception handling is used in this implementation. The summary of the first hardened version (H1) can be expressed in the form of the following pseudocode:

```
part_A1;  // triplicated part A
part_A2;
part_A3;
```

```
voting;   // voting on part A results propositions
part_B;
part_C;
```

The compiler's flag /EHsc is used. Static code size is 463 machine instructions which corresponds to 1278 bytes. Number of executed instructions for considered simulation horizon is 2 289 794.

The idea of the second hardened implementation is to utilize C++ exception handling statements to recover from system-detected errors (e.g. memory access violations, FPU exceptions) by switching the DMC code to its backup. Moreover, the FPU state is reset as the DMC is FPU intensive. Previous research showed that FPU resetting is crucial for further calculations as no further exceptions will be signalled by the FPU and provided results might be erroneous. The summary of the second hardened version (H2) can be expressed in the form of the following pseudocode:

```
try{
   if(modules_switched){
      DMC; }
   else
      throw excp;}
catch(…) {
   if(!already_switched)
      switch_modules;
   _fpreset();
   DMC;}
```

The compiler's flag /EHa /fp:except is used in this implementation. Static code size is 130 machine instructions, which corresponds to 485 bytes. Number of executed instructions is 1 941 900.

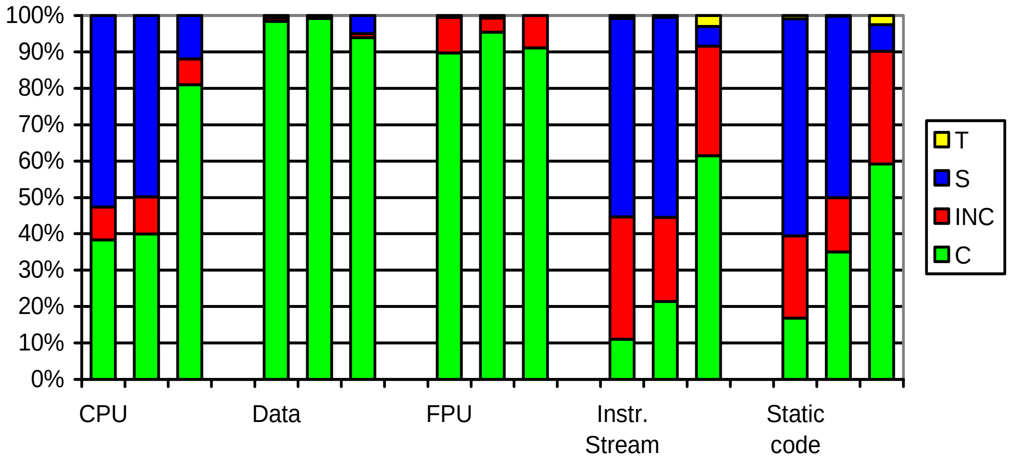

Figure 8. Comparison of experimental results. For each fault location: *left bars:* the rudimentary implementation, *middle bars:* H1 - the first hardened implementation, *right bars:* H2 - the second hardened implementation.

Fig. 8 compares experimental results obtained in three studied software implementations for faults located in CPU registers, data memory area, FPU registers, executed instructions stream and static code image. These locations are marked on x-axis.

It is worth noting, that for rudimental and H1 versions the sum of C and INC categories is almost the same. The poor dependability improvement is a consequence of not integrated exception handling within the application. System built-in fault detectors (signaled as exceptions to the application) are very efficient. H2 version uses alternative code in passive configuration and integrates exception handling mechanisms. The disadvantage of this version is that only exception occurrence activates the switch to the backup code. The rise of correct behaviour (C category) percentage comparing to the rudimentary version shows how many tests can be recovered with the secondary code. On the other hand, existence of tests within the INC category for H2 denotes that injected faults may result in major deviations without exception occurrence. They may lead to undetected errors which can influence further execution (not a temporary value within the iteration boundary). So, further dependability improvements of the DMC source code will be focused on building the detectors for such errors. The considered faults in the code locations (executed instruction stream as well as the static code image) were designed to mimic mainly memory location faults. Faulty code remains unchanged as long as it is reloaded from persistent storage (e.g. disk, EPROM, flash RAM). It is worth to note, that the solution H2 would be very efficient in case of faults located in the instruction cache as the faulty code could be recovered while flushing the cache or reading the code from fault robust location.

5. CONCLUSION

The paper studies dependability of software implementation of the explicit DMC algorithm applied to a rectification process. The process has two manipulated variables, two controlled variables and significant time-delays. Dependability is examined using software implemented fault injector. Its functionality towards better fault effect traceability and the whole fault injection process is described.

Results of the experiments carried out clearly indicate that the well-known and widely used formulation of the DMC algorithm is susceptible to software faults. It is particularly important in case of industrial processes such as the considered rectification column because faults are likely to lead to undesirable behaviour of the process. More specifically, in the least difficult situation a fault can disturb composition of the products whereas in the most dangerous case it can destabilise the controlled process. Technological and financial consequences of faults are of fundamental importance. Energy losses and unacceptable compositions of the products (which means that the product cannot be sold) are just two examples of such situations.

In order to increase fault robustness of software implementation of the DMC algorithm, a few techniques can be applied. The first choice techniques should be those making possible creative usage of fault detection. In the case of application considered in the paper, exception handling brought the significant improvement of the algorithm dependability.

REFERENCES

Benso, A. and Prinetto, P., 2003. *Fault injection techniques and tools for embedded systems reliability evaluation*. Kluwer Academic Publishers.

Cutler, R. and Ramaker, B., 1979. Dynamic matrix control – a computer control algorithm. *Proceedings of AIChE National Meeting*. Houston, USA.

Gawkowski, P. and Sosnowski, J., 2007. Experiences with software implemented fault injection. *Proceedings of International Conference on Architecture of Computing Systems*. VDE Verlag GMBH, pp. 73–80.

Gawkowski, P. and Sosnowski, J., 2006. Analysing system susceptibility to faults with simulation tools. *Annales UMCS Informatica AI*. Vol. 4, pp. 123–134.

Gawkowski, P. and Sosnowski, J., 2005a. Software implemented fault detection and fault tolerance mechanisms – part I: Concepts and algorithms. *Kwartalnik Elektroniki i Telekomunikacji*. Vol. 51, pp. 291–303.

Gawkowski, P. and Sosnowski, J., 2005b. Software implemented fault detection and fault tolerance mechanisms -- part II: Experimental evaluation of error coverage. *Kwartalnik Elektroniki i Telekomunikacji*. Vol. 51, pp. 495–508.

Gawkowski, P. et al, 2005. Analyzing the effectiveness of fault hardening procedures. *Proceedings of 11th IEEE Int. On-Line Testing Symposium*. pp. 14–19.

Gawkowski, P. and Sosnowski, J., 2003. Dependability evaluation with fault injection experiments. *IEICE Transactions on Information & System*. Vol. E86-D. pp. 2642–2649.

Gawkowski, P. and Sosnowski, J., 2002. Experimental Validation Of Fault Detection And Fault Tolerance Mechanisms. *Proceedings of 7th IEEE Int. Workshop on High Level Design Validation And Test*. Cannes.

Ławryńczuk, M. et al 2007. Multilayer and integrated structures for predictive control and economic optimisation. *Proceedings of IFAC Symposium on Large Scale Systems*. Gdańsk, Poland.

Maciejowski, J.M., 2002. *Predictive control with constraints*. Prentice Hall. Harlow.

Morari, M. and Lee, J.H., 1999. Model predictive control: past, present and future. *Computers and Chemical Engineering*. Vol. 23, pp. 667–682.

Qin, S.J. and Badgwell, T.A., 2003. A survey of industrial model predictive control technology. *Control Engineering Practice*. Vol. 11, pp. 733–764.

Pułaczewski, J., 1998. *Multidimensional DMC algorithm*. Report of ICCE WUT no 98–11, Warsaw, Poland (in Polish).

Rossiter, J.A., 2003. *Model-based predictive control*. CRC Press, Boca Raton.

Sosnowski, J. et al, 2005. Fault injection stress strategies in dependability analysis. *Control and Cybernetics*. Vol. 33, pp. 679–699.

Sosnowski, J. et al, 2003a. Software implemented fault inserters. *Proceedings of IFAC Workshop on Programmable Devices and Systems*. Ostrava, pp. 293–298.

Sosnowski, J. et al, 2003b. Fault injection stress strategies. *Proceedings of 4th IEEE LATW 2003 Workshop*. pp. 258-263.

Tatjewski, P., 2007. *Advanced control of industrial processes, Structures and algorithms*. Springer. London.

Tatjewski, P. et al, 2006. Linking nonlinear steady-state and target set-point optimisation for model predictive control. *Proceedings of IEE International Control Conference ICC 2006*. Glasgow, UK.

Wood, R.K. and Berry, M.W., 1973. Terminal Composition Control of a Binary Distillation Column. *Chemical Engineering Science*. Vol. 28, pp. 1707–1717.